NSO Modulo 4 version 2008

Sistemas Operativos (Universidad Nacional de La Matanza)

MODULO: 4

SINCRONIZACIÓN Y COMUNICACIÓN ENTRE PROCESOS

CONTENIDO:

Conceptos de Concurrencia.
Conceptos de Sincronización entre Procesos.
Algoritmos de Sincronización y Comunicación entre procesos
Conceptos sobre Abrazo Mortal (Deadlocks)

OBJETIVO DEL MÓDULO: Presentar las soluciones algorítmicas de los problemas planteados en la sincronización y comunicación entre procesos.

OBJETIVOS DEL APRENDIZAJE: Después de estudiar este módulo, el alumno deberá estar en condiciones de:

- Explicar y reconocer los distintos algoritmos de sincronización y comunicación entre procesos.
- Comprender los conceptos y características de la concurrencia de los procesos.
- Explicar la situación de Deadlock y como evitarla.
- Comprender la terminología específica desarrollada en este módulo.

Metas de este módulo

En este Módulo se tratará detallar los mecanismos que provee el sistema operativo para que varios procesos puedan acceder simultáneamente a un recurso. En particular Hay concurrencia tanto en la multiprogramación como en el multirproceso. Desde que se empezó a usar la multiprogramación, se choco con que 'la velocidad relativa de ejecución no puede ser predecida'.
Para la multiprogramación se plantean 3 dificultades derivadas de que la ejecución no pueda ser predicha:

- Compartir recursos globales es peligroso
- Es difícil para el sistema operativo asignar óptimamente los recursos.
- Es muy difícil encontrar un error de programación, generalmente las condiciones y resultados del error no son reproducibles.

Para el multiproceso también se plantean estos problemas, mas otros derivados del procesamiento simultaneo.
La solución al primer problema es el acceso controlado a los recursos compartidos, no deberían ser accedidos por más de un proceso al mismo tiempo. Esto se conoce como sincronización de procesos.

También se explicarán los conceptos de comunicación entre procesos, condiciones de competencia, regiones o secciones críticas, los distintos algoritmos para la solución por software y hardware la mutua exclusión. Además, se Introducirán las definiciones de semáforo, monitores, y las soluciones para los problemas clásicos de la comunicación entre procesos. Por último se plantearán los conceptos sobre Abrazo Mortal (deadlock).

4. Sincronización y comunicación entre procesos

En los sistemas operativos, en general, los procesos que trabajan juntos comparten con frecuencia un espacio común para almacenamiento, en el que cada uno puede leer o escribir, o también comparten un recurso. El espacio compartido puede estar en la memoria central o ser un archivo, una estructura de datos o una variable global de un programa en ejecución.

El acceso a estos recursos compartidos o la localización de áreas compartidas en la memoria generan problemas de uso y de comunicación entre los procesos. Para resolver estos problemas de competencia entre procesos, se utilizan dos mecanismos ***: la sincronización y la comunicación.***

La sincronización y la comunicación entre procesos son temas muy importantes en los S.O. modernos, sobre todo si la ejecución se efectúa concurrentemente. En la figura 4.01 presentamos dos procesos Px y Py que comparten recursos comunes y que requieren sincronizarse en el uso de los mismos y también se intercambian datos como mensajes.

Lo explicado genéricamente como aplicado a procesos también es válido para la sincronización de los threads (procesos livianos).

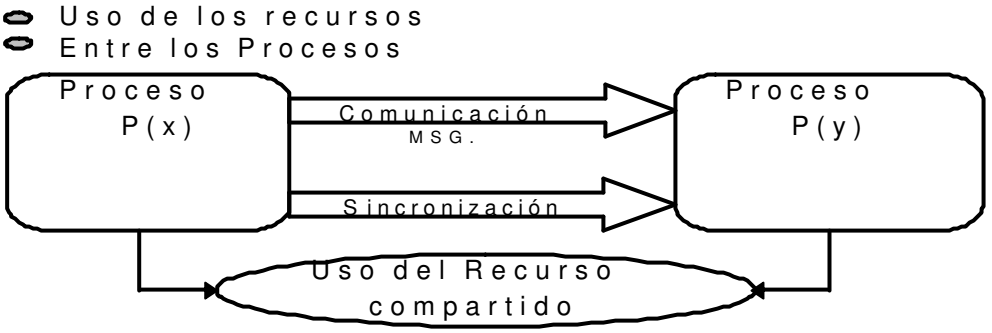


Fig. 4.01 Relaciones entre procesos

Definimos a ambos términos como:

SINCRONIZACIÓN ENTRE PROCESOS: Ordenamiento de las operaciones en el tiempo debido a las condiciones de carrera (acceder a los diversos recursos asincrónicamente. Ejemplo, dos procesos seleccionan a la misma impresora simultáneamente lo que daría listados mezclados. La solución a este problema es colocar una valla con cerradura que la llamaremos Lockeo).

COMUNICACIÓN ENTRE PROCESOS: Intercambio de Datos. La comunicación permite que los procesos cooperen entre sí en la ejecución de un objetivo global, mientras que la sincronización permite que un proceso continúe su ejecución después de la ocurrencia de un determinado evento.

4.0. Principios generales de concurrencia entre procesos

En un sistema multiprogramado con un único procesador, los procesos se intercalan en el tiempo para dar la apariencia de ejecución simultánea. Aunque no se consigue un proceso paralelo real y se produce una cierta sobrecarga en los intercambios de procesos, esta ejecución produce beneficios importantes en eficiencia y estructuración de programas. En un sistema con varios procesadores, no sólo es posible intercalar los procesos, sino también superponerlos. En ambos casos, los problemas de concurrencia parten del hecho de que la velocidad relativa de ejecución de los procesos no puede predecirse, ya que depende de la actividad de otros procesos, del tratamiento de las interrupciones y de las políticas de planificación. Así surgen las siguientes dificultades:

1. Compartir entre procesos los recursos globales está llena de riesgos.
2. Para el S.O. resulta difícil asignar los recursos de forma óptima.
3. Resulta difícil localizar un error de programación porque los resultados no son normalmente reproducibles.

Pero también, El sistema operativo debe poder seguir el rastro de varios procesos activos, debe poder asignar o desasignar recursos a cada proceso activo y debe proteger la información y los recursos físicos de cada proceso, contra la interferencia intencional o no de otro proceso. Los resultados de un proceso deben ser independientes de su velocidad relativa.

Un sistema multiprocesador debe solucionar además los problemas originados por el hecho de que varios procesos puedan estar ejecutando a la vez. Un ejemplo del problema de concurrencia se plantea en los procedimientos o funciones que pueden ser llamados por cualquier programa en cualquier momento. En el caso de un sistema monoprocesador, el problema se presenta cuando una interrupción detiene la ejecución de instrucciones en cualquier punto de un proceso, y otro programa puede invocar entonces al mismo procedimiento o función. En el caso de un sistema multiprocesador, se tiene la misma

condición y, además, el problema puede ser causado por dos procesos que estén ejecutando simultáneamente el mismo procedimiento. La solución a este conflicto es controlar el acceso al recurso compartido, es decir mutua exclusión.

4.1. Problemas Concurrentes:

Los programas pueden ser clasificados en secuénciales y en concurrentes. Un programa secuencial especifica una secuencia de instrucciones que se ejecutan sobre un procesador que definimos como proceso o tarea. Un programa concurrente especifica dos o más procesos secuénciales que pueden ejecutarse concurrentemente como tareas paralelas.

Un proceso secuencial se caracteriza por no ser dependiente de la velocidad de ejecución y de producir el mismo resultado para un mismo conjunto de datos de entrada, mientras que en un proceso concurrente (o lógicamente paralelo) las actividades están superpuestas en el tiempo (una operación puede ser comenzada en función de la ocurrencia de algún evento, antes de que termine la operación que se estaba ejecutando).

La programación concurrente requiere de mecanismos de sincronización y comunicación entre los procesos.

La **mutua exclusión** es uno de los problemas más importantes que presenta la ejecución de los procesos concurrentes debido al hecho de ser la abstracción de muchos problemas de sincronización. Ejemplo: leer o escribir variables globales, modificar tablas, sobreescribir un archivo, etc.

Para manipular o realizar operaciones sobre objetos, éstos deben ser adecuada y cuidadosamente tratados, de forma tal que no se produzcan conflictos. Si las tareas son independientes entre ellas (solo tienen variables disjuntas) entonces su ejecución no presenta mayores inconvenientes, pero si tienen estructuras de datos o variables compartidas (datos o variables globales que se modifican entre ellos) o se envían mensajes entre sí, se producen conflictos ya sea en el resultado de la ejecución o en la secuencia o cooperación de su ejecución. Es fácil demostrar que si las variables de una tarea son inaccesibles por otras tareas en un instante de su ejecución, el resultado final de la misma será una función independiente del tiempo, o sea, no existe ninguna interferencia. En cambio si una tarea utiliza las variables de otra tarea, el resultado obtenido por esta última dependerá de las velocidades relativas entre las tareas.

4.1.1. Grafos de precedencia.

Un grafo de precedencia es un grafo sin ciclos donde cada nodo representa una única sentencia o un conjunto secuencial de instrucciones agrupadas. Un arco que sale del nodo S1 hacia S2 indica que S2 puede ser ejecutado sólo si S1 ha completado su ejecución.

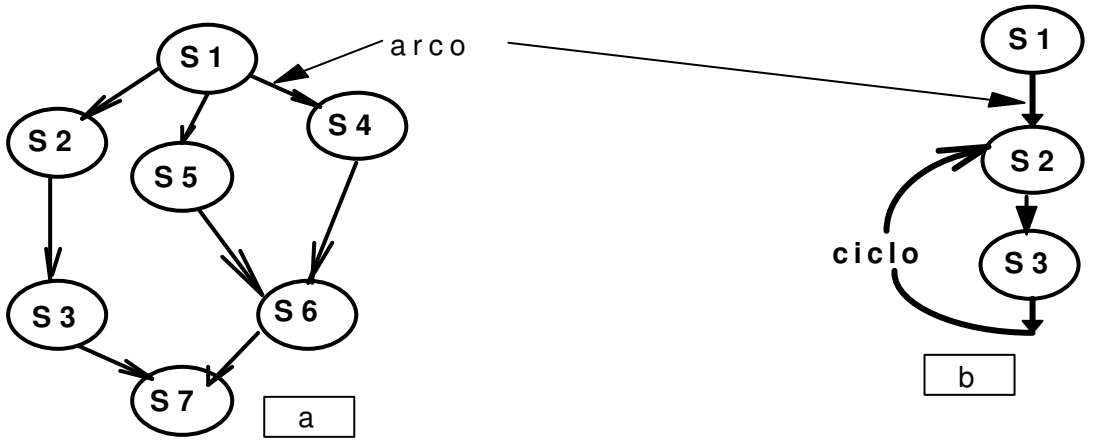


Fig. 4.02 Grafos de precedencia

En el grafo de la Figura 4.02a se ejemplifican las precedencias de siete sentencias (ejemplo: S7 solo puede seguir cuando terminen S3 y S6). El grafo 4.02b tiene un ciclo de la finalización de S3 a S2 por lo que no hay precedencia.

4.1.2. Condiciones de concurrencia (Bernstein)

Debe darse un conjunto de condiciones para que se puedan ejecutar varios procesos a la vez. Si las sentencias cumplen con ellas, podrán ser ejecutadas concurrentemente o sea en paralelo. La ejecución

paralela resultará si no hay dependencia en los resultados o en los requerimientos de entrada de alguna de las dos sentencias. Esto es válido si se verifica la relación de variables o algún dispositivo de Entrada / Salida.
Establecemos la siguiente notación:

Conjunto de Lectura: R (S_i) = (a_1, a_2,... , a_m)

El conjunto de lectura de la sentencia S_i es aquel formado por todas las variables que son **referenciadas** por la sentencia S_i durante su ejecución sin sufrir cambios.

Conjunto de escritura: W (S_i) = (b_1, b_2,... , b_n)

El conjunto de escritura de la sentencia S_i es aquel formado por todas las variables cuyos valores son **modificados** durante la ejecución de S_i.

Ejemplos:

R (Read (a)) = (ø) W (Read (a)) = (a)
R (Read (b)) = (ø) W (Read (b)) = (b)
R (c= a+b) = (a,b) W (c= a+b) = (c)

Definición: Dos sentencias cualesquiera S_i y S_j pueden ejecutarse concurrentemente produciendo el mismo resultado que si se ejecutaran secuencialmente sí sólo sí se cumplen las siguientes condiciones:

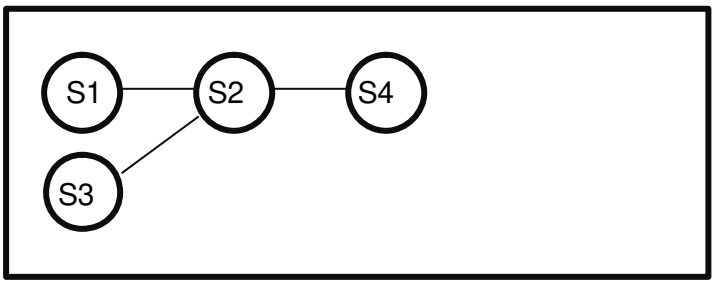
1. **R (S_i) ∩ W (S_j) = (ø)**
2. **W(S_i) ∩ R (S_j) = (ø)**
3. **W(S_i) ∩ W (S_j) = (ø)**

Si las tres condiciones producen conjunto vacío, podemos asegurar que no hay dependencia entre las sentencias.

Siempre se evalúan pares de sentencias por ejemplo S_i y S_j.
Sea: S_1 : a = b+ c; S_2: c = b+d; S_3: e = d + f; y S_4: g = a*b;

Sentencia	R	W
S_1	{b,c}	{a}
S_2	{b,d}	{c}
S_3	{d,f}	{e}
S_4	{a,b}	{g}

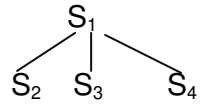
Analizamos:
S_1 y S_2 ⇒ { c } = 0
S_1 y S_3 ⇒ ok (ok significa que pueden ejecutarse en forma concurrente)
S_1 y S_4 ⇒ { a } = 0
S_2 y S_3 ⇒ ok
S_2 y S_4 ⇒ ok
S_3 y S_4 ⇒ ok
Ejemplo1: se elige S_1



1t se ejecuta S_1 y S_3 (concurrentemente)
2t se ejecuta S_2
3t se ejecuta S_4

Fig. 4.03 Ejecución de 4 procesos

Ejemplo2: se elige S_1



Ejemplo3: se elige S_3

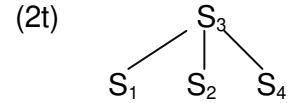


Fig. 4.04 Estratégias de ejecución

Se aplican sobre Readers (R) y Writers (W) y requiere de múltiples comparaciones por lo que vuelven a consideración por los threads cuya ejecución es relativamente simple, pero no son aplicables a procesos pesados por los excesivos parámetros que lo componen y el costo de comparar cada uno de ellos si cumplen con las condiciones de Bernstein.

4.1.3. Especificaciones concurrentes:

- Existen diversas notaciones para especificar actividades concurrentes. Básicamente veremos dos: fork y join (no estructurados) y cobegin y coend (estructurados).

A) Instrucciones FORK - JOIN (Conway -1963, Dennis y Van Horn -1966)

La instrucción **fork** (tenedor, horqueta, separador) indica el comienzo de la concurrencia (crea e inicia dos instrucciones concurrentes, una en la rotulada etiq (etiqueta) o Label y la otra es la continuación de la ejecución siguiente instrucción al fork) y **join** recombina (junta) la concurrencia en una sola instrucción indicando que ha concluido la concurrencia. La instrucción ***fork etiq*** produce dos ejecuciones concurrentes en un programa. Una ejecución comienza a partir del rótulo etiq mientras que la otra prosigue con la ejecución de la sentencia que está a continuación de la instrucción fork, como se observa en la figura 4.05. Cada rama tiene que solicitar ser unida a la otra. Puesto que se ejecutan a distintas velocidades, una puede ejecutar join antes que la otra, en este caso, la que ejecuta primero el join termina, mientras que se permite continuar la segunda. Si hubiera tres ramas, las primeras dos en ejecutar la unión terminarían, mientras que se permitiría continuar la tercera. Esto se ejemplifica en la figura 4.03.

```
sent1;
sent2;
fork etiq;
...
join;
...
end;

etiq:
        sent1;
          sent2;
          ...
          sentn;
quit;
```

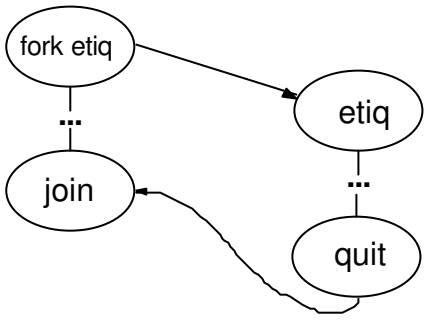


Fig. 4.05 Notación de concurrencia: fork - join

Join necesita saber el número de ramas a unir para terminar todas excepto la última. Para ello se tiene que especificar como parámetro el número de computaciones a reunir.

Ejemplo: para dos ramas count=2

count = count - 1;
if count = 0 then quit;

(instrucción de la que resulta la terminación de la ejecución). Ejemplifiquemos esto en el grafo de la figura 4.02a.

```
     .....
     S1;
     cont1 = 2;
     fork L1;
     fork L2;
     S2;
     S3;
     go to L4;
     cont2 = 2;
L1:  S4;
     go to L3;
L2:  S4;
     go to L3;
L3:  join cont2;
     S6;
L4:  join cont1;
     S7;
     ...
```

Resumen de fork y join
1. fork es, en esencia, un goto que simultáneamente bifurca y continúa la ejecución.
2. quit y join permiten que dos ramas de actividad vuelvan a juntarse.
3. quit: Lo ejecuta el proceso hijo cuando terminó su tarea.
4. join: Lo ejecuta el proceso padre para esperar a que termine el hijo.
5. No es estructurado por su estructura de control.
6. La instrucción join tiene un parámetro que especifica el número de procesos a ser juntados. Join tiene que ejecutarse indivisiblemente, esto es, la ejecución de dos join consecutivos es equivalente a la ejecución secuencial de dos instrucciones, en algún orden no definido.

B) COBEGIN / COEND

- Década 80 Hoare lo llamó parbegin y parend por parallel-begin.
- Década 90 se designa concurent en lugar de parallel.

Todas las instrucciones insertadas entre cobegin y coend pueden ejecutarse concurrentemente como se observa en la figura 4.06.

```
COBEGIN
    sent1;
    sent2;
    ...
    sentn;
COEND
```

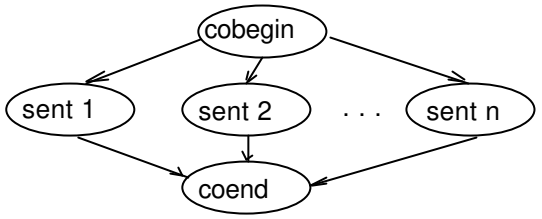


Fig. 4.06 Notación de concurrencia: cobegin - coend

Cada $sent_i$ puede ser un grupo de sentencias con esta construcción se están creando *n* procesos concurrentes cada uno de los cuales debe ejecutarse completamente antes que el proceso creador pueda continuar.
La sent *n+1* solo puede ejecutarse solo si S_i = 1,...,*n* han terminado.

Obsérvese que el grafo de la figura 4.02a no puede ser resuelto mediante estas sentencias.

- Ejemplo: Programa para encontrar el máximo entre 4 números.

```
cobegin
    m1=max(a,b);
    m2=max(c,d);
coend
m=max(m1, m2);
return m;
```

```
fork max1;
fork max2;
join;
join;
m=max(m1, m2)
return m;

max1:
    m1=max(a,b);
    quit;

max2:
    m2=max(c,d);
    quit;
```

C) Implementación

- Notación COBEGIN/COEND debe ser una construcción del lenguaje utilizado. Debe estar en el compilador.
- Notación FORK puede ser implementada mediante system calls.
- El Proceso creador es llamado **padre**.
- El Nuevo proceso es llamado **hijo**, a su vez, puede crear sus propios hijos.
- Cada proceso nuevo necesitará recursos (tiempo de CPU, memoria, archivos, dispositivos, etc.). Puede obtenerlos:
  1. Tomándolos del S.O.
  2. Obligando al padre a dividir recursos entre sus hijos o compartirlos (memoria o archivos), esta restricción impide sobrecargas en el sistema.
- Existen dos formas de activar el nuevo proceso.
  1. El padre ejecuta en forma concurrente con el hijo.
  2. El padre espera hasta que todos los hijos terminen.

4.2. Relaciones entre procesos concurrentes y sus conflictos

- Un proceso es **independiente** si no puede afectar o ser afectado por otros procesos corriendo en el sistema:
  1. Su estado no es compartido con ningún otro proceso.
  2. La ejecución es determinística: el resultado de la ejecución depende sólo de las entradas.
  3. Su ejecución puede detenerse y reasumirse sin por eso causar efectos laterales al resto del sistema.
- Un proceso es **interactuante** si puede afectar o ser afectado por otros procesos.
  1. Su estado es compartido con otros procesos.
  2. El resultado de su ejecución no puede ser predicho ya que depende de la ejecución de otros procesos.
- Ejemplo:

Independientes

```
COBEGIN
      M1 = MAX (A, B)
      M2 = MAX (C, D)
COEND
M = MAX (M1, M2)
```

Interactuantes

```
J = 10
COBEGIN
     PRINT J
     J = 1000
COEND
```

⇒ El resultado del ejemplo de la derecha depende de las velocidades relativas de los dos procesos. Uno ejecuta sobre CPU (ejecuta en nanosegundos) y el otro sobre E/S (lo hace en milisegundos).
⇒ El Kernel que multiplexa el procesador no controlan con precisión las señales de interrupción.
⇒ Como los procesos no ejecutan exactamente a la misma velocidad aparece una **condición de concurso** o condición de carrera (**race condition**).
⇒ **race condition**: Es la Situación en la cual el resultado de la ejecución de 2 o más procesos interactuantes depende del orden de ejecución de los mismos.

- Ejemplo:

Productor (observador)

```
for(;;)

      {
      observarEvento();
      nroEventos++;
    }
```

Consumidor (reportero)

```
for(;;)
     {
     printf("%d",nroEventos);
     nroEventos = 0;
     }
```

Algunas situaciones de concurso:
1. La interrupción del reportero justo después del printf() por lo que vuelve al productor y después pone nro = 0 por lo tanto se pierde un evento.
2. En la sentencia nroEvento++. La solución queda como ejercicio para el lector. (Sugerencia: Considere como traduce el compilador la sentencia nroEvento++ a código de máquina).

¿Cómo solucionar una condición de concurso?. La solución para evitar los problemas de concurso o competencia en el uso de la memoria compartida, archivos compartidos y cualquier otro recurso compartido, es determinar una forma de prohibir que más de un proceso **lea o escriba** los datos compartidos a la vez. En otra palabra, lo que necesitamos es la **mutua exclusión** (una forma de garantizar que si un proceso utiliza una variable o archivo compartido para modificarlo, los demás procesos no puedan utilizarlos), es decir un mecanismo que permita que los procesos accedan en forma ***ordenada*** al recurso compartido.

Dentro de la ejecución de un programa se presentan otros conflictos con el uso de recursos que el S.O. deberá resolver, a saber:

- **Inanición, o Postergación o Aplazamiento Indefinido (Starvation)**: Consiste en el hecho de que uno o varios procesos nunca reciban el suficiente tiempo de ejecución para terminar su tarea. Por ejemplo, que un proceso ocupe un recurso y lo marque como 'ocupado' y que termine sin marcarlo como 'desocupado'. Si algún otro proceso pide ese recurso, lo verá 'ocupado' y esperará indefinidamente a que se 'desocupe'.

- **Condición de Espera Circular**: Esto ocurre cuando dos o más procesos forman una cadena de espera que los involucra a todos. Por ejemplo, suponer que el proceso A tiene asignado el recurso 'cinta' y el proceso B tiene asignado el recurso 'disco'. En ese momento al proceso A se le ocurre pedir el recurso 'disco' y al proceso B el recurso 'cinta'. Ahí se forma una espera circular

entre esos dos procesos que se puede evitar quitándole a la fuerza un recurso a cualquiera de los dos procesos.

- **Condición de No expropiación:** Esta condición no resulta precisamente de la concurrencia, pero juega un papel importante en este ambiente. Esta condición especifica que si un proceso tiene asignado un recurso, dicho recurso no puede arrebatársele por ningún motivo, y estará disponible hasta que el proceso lo 'libere' por su voluntad.

- **Condición de Espera Ocupada**: Esta condición consiste en que un proceso pide un recurso que ya está asignado a otro proceso y la condición de no expropiación se debe cumplir. Entonces el proceso estará gastando el resto de su porción de tiempo (time slice) verificando si el recurso fue liberado. Es decir, desperdicia su tiempo ejecución en esperar. La solución más común a este problema consiste en que el sistema operativo se dé cuenta de esta situación y mande a una cola de espera al proceso, otorgándole inmediatamente el turno de ejecución a otro proceso.

- **Condición de Mutua Exclusión**: Cuando un proceso usa un recurso del sistema realiza una serie de operaciones sobre el recurso y después lo deja de usar. A la sección de código que usa ese recurso se le **l**ama **'región crítica'**. La condición de mutua exclusión establece que solamente se permite a un proceso estar dentro de la misma región crítica. Esto es, que en cualquier momento solamente un proceso puede usar un recurso a la vez. Para lograr la mutua exclusión se ideo también el concepto de 'región crítica'. Para lograr la mutua exclusión generalmente se usan algunas técnicas para entrar a la región crítica como ser: semáforos, monitores, el algoritmo de Dekker y Peterson, los 'candados'.

- **Condición de Ocupar y Esperar un Recurso:** Consiste en que un proceso pide un recurso y se le asigna. Antes de soltarlo, pide otro recurso que otro proceso ya tiene asignado.

Los problemas descriptos son todos importantes para el Sistema Operativo, ya que debe ser capaz de prevenir o corregirlos. Tal vez él más serio que se puede presentar en un ambiente de concurrencia es el 'abrazo mortal', también llamado en inglés **deadlock**. El deadlock involucra algunos procesos, éstos quedarán bloqueados en la ejecución para siempre. Este punto lo estudiaremos mas adelante en este módulo.

- **Labores del S.O**.: los elementos de gestión y diseño que surgen por causa de la concurrencia son:
  1. El S.O. debe ser capaz de seguir la pista de los distintos procesos activos, para esto utiliza los PCB.
  2. El S.O. debe asignar y quitar los distintos recursos a cada proceso activo, estos recursos incluyen tiempo de procesador, memoria, archivos y dispositivos de E/S.
  3. El S.O. debe proteger los datos y los recursos físicos de cada proceso contra injerencias no intencionadas de otros procesos.
  4. Los resultados de un proceso deben ser independientes de la velocidad relativa a la que se realiza la ejecución con respecto a otros procesos concurrentes.
- **Interacción entre procesos:** hay distintos niveles de conocimiento que cada proceso tiene de la existencia de los demás. Los tres niveles de conocimiento son:

Grado de Conocimiento	Relación	Influencia que un Proceso tiene sobre otro	Potenciales Problemas de Control
Los procesos **no se conocen**	Competencia	• Los resultados de un proceso son independientes de la acción. • La duración del proceso puede ser afectada	• Mutua. Exclusión • Deadlock (recursos reusables). • Starvation (Inanición) .
Los procesos **se conocen indirectamente**	Cooperación por compartir	• Los resultados de un proceso pueden depender de la información obtenida de otro. • La duración del proceso puede ser afectada	• Mutua. Exclusión • Deadlock (recursos reusables). • Starvation. • Coherencia de datos
Los procesos **se conocen directamente**	Cooperación por comunicación	• Los resultados de un proceso pueden depender de la información obtenida de otro. • La duración del proceso puede ser afectada	• Deadlock (recursos consumibles). • Starvation.

Tabla 4.1 Relación entre procesos

Estas condiciones no son siempre tan terminantes, muchas veces los procesos compiten y cooperan al mismo tiempo.

Beneficios de la concurrencia

- Trata de evitar los tiempos muertos del procesador
- Comparte y optimiza el uso de recursos
- Permite la modularidad en las diferentes etapas del proceso
- Acelera los cálculos
- Da mayor comodidad

Desventajas de la concurrencia

- Inanición e interrupción de procesos
- Ocurrencia de interbloqueos (deadlocks)
- Que dos o más procesos compitan por el mismo recurso (No Apropiativo) complica su tratamiento.

4.3. Introducción al problema de la Región Crítica (R.C.)

El problema de evitar las condiciones de concurrencia también se puede formular de forma más abstracta. Durante cierta parte del tiempo, un proceso está ocupado realizando cálculos internos y otras tareas que no conducen a condiciones de competencia. Sin embargo, en algunas ocasiones, un proceso puede tener acceso a un recurso compartido o realizando tareas críticas que pueden llevar a conflictos. Esa parte del programa, en la cual se tiene acceso a la memoria compartida se llama la **Región o sección Crítica**. Si se puede programar para que no ocurra que dos procesos están al mismo tiempo en su Región Critica, se puede evitar las condiciones de competencia.

Aunque esta condición evita los conflictos, no es suficiente para que los procesos paralelos cooperen en forma correcta y usen de modo eficaz los datos compartidos. Se necesita de varias condiciones para una buena solución. Recordemos los siguientes conceptos:

⇒ Los **puntos de entrada** de un recurso indican la cantidad de procesos que pueden utilizarlo *simultáneamente* al mismo.

⇒ Un recurso con 1 punto de entrada se lo denomina **recurso crítico** o **no compartible**.

Región crítica de un proceso es la fase o etapa en la vida de ese proceso concurrente en la cual accede a un **recurso crítico** para modificarlo o alterarlo. Es Un trozo de código en el que se utiliza un recurso compartido y que se ejecuta de forma exclusiva. Algunos autores la llaman **sección crítica**.

⇒ Las condiciones de concurso se presentan debido a la presencia de regiones críticas dentro de los procesos.

Definimos el problema de **mutua exclusión** como el de desarrollar mecanismos que garanticen que solo un proceso está operando sobre un objeto a la vez. Éste objeto podría ser un espacio de memoria, por ejemplo. Utilizamos cuatro criterios para evaluar un mecanismo de mutua exclusión.

1. ¿Asegura el mecanismo la mutua exclusión?
2. ¿Se hacen suposiciones acerca de la velocidad relativa de los procesos?
3. ¿Se garantiza que la terminación de un proceso fuera se su sección crítica no afecta la habilidad de otros procesos en competir por el uso de un recurso compartido?
4. ¿Cuando más de un proceso desea entrar a la región crítica a la vez se concede la entrada a ésta a uno de ellos en tiempo finito?

Las respuestas a estas cuestiones permiten usar la Región Critica mediante un protocolo de sincronización que garantice la mutua exclusión entre procesos.

Propiedades para usar la Región Crítica

- El problema de la región crítica consiste en sincronizar los procesos de forma tal que se cumpla el siguiente ***protocolo de sincronización***.
  1. **Mutua exclusión**: sólo un proceso a la vez puede estar ejecutando en su región crítica (lo accede y la usa).
  2. **Progreso**: Un proceso fuera de su Región Critica no debe impedir la entrada de otro proceso a la misma. Sólo los procesos que quieren entrar a la Región Critica deben participar en la decisión. (La solución de estricta alternancia viola este requisito).
  3. **Espera limitada**: Un proceso debe poder entrar a la Región Crítica después de un número limitado de intentos.
  4. **Abandono** (Tiempo limitado): debe dejar a la Región Critica en un tiempo finito. (Es "casi" una consecuencia de la 3).
  5. **Penalidad:** Un Proceso no puede consumir tiempo de ejecución mientras espera por un recurso.
  6. **Privilegio**: No debe haber ningún proceso privilegiado que monopolice la Región Crítica.
- Este protocolo de sincronización se implementa de forma tal que cada proceso debe ejecutar el código de la figura 4.07 para usar un recurso crítico.

Las soluciones comunes al problema de mutua exclusión siguen un protocolo de tres pasos mostrado en la figura 4.07. La región crítica es aquella sección de código que a lo más un proceso puede ejecutar a la vez. Si lo que se quiere asegurar es que no más de un proceso haga referencia a un espacio de memoria a la misma vez, éste contendrá todo aquel código manipule a éste espacio.

/* (RC) designa una serie de instrucciones que utilizan el recurso crítico y cumplen con el protocolo y éstas instrucciones se descomponen en tres etapas. */

- Las funciones entradaRC() y salidaRC() se deben implementar de forma tal que cumplan el protocolo de sincronización.
- Estas dos funciones determinan el ***algoritmo de sincronización*** utilizado.
- En general se resuelven mediante un hardware cableado o un mecanismo explícito de ejecución atómico (Master - Slave) que impide los tratamientos de las interrupciones.

Se pueden implementar soluciones por software para los procesos en condiciones de competencia que se ejecutan en maquinas monoprocesador o multiprocesador con una memoria compartida, estas soluciones suponen que existe una mutua exclusión elemental en el acceso a memoria es decir que los accesos a memoria se hacen en una secuencia ordenada. Veamos algunos algoritmos que permiten sincronizar los procesos entre sí.

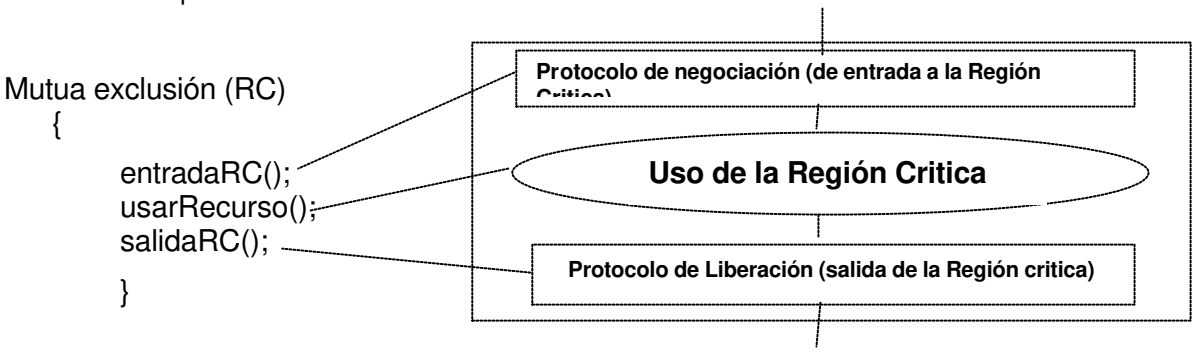


Fig. 4.07 Mutua Exclusión

4.4. Algoritmos de sincronización con espera activa.

- **Espera activa** o **busy waiting** significa que el proceso sigue ocupando la CPU mientras no puede entrar a su región crítica (el proceso no se bloquea).
- El algoritmo queda realizando un **spinlock** chequeando el recurso hasta que éste es liberado.
- El gran problema con estos algoritmos es que desperdician tiempo de CPU.
- Se declara una variable pública booleana en que v = 0 indica recurso desocupado y v = 1 recurso ocupado (es arbitrario si v es 0 o 1)
- Se consulta a v para poder entrar. Si es cero entra y lo coloca en “1” y cuando sale lo pone en “0”

4.4.1. Solución simple

Este algoritmo utiliza un flag. Al entrar a la región crítica se fija si es uno o cero si es cero lo pone en uno y entra a la sección crítica; si es uno espera hasta que valga cero. Antes de salir de la sección crítica iguala el flag a cero. Ejemplo:

- Se utiliza una variable global “ocupada”, que indica el estado de la región crítica.
- Se consulta a ocupada para poder entrar.

```
entradaRC()
{
    do
        ;
    while(ocupada);
    ocupada = 1;
}
```

```
salidaRC()
{
    ocupada = 0;
}
```

A esta solución también se la conoce como **Algoritmo de Variables de Cierre.**

- No funciona. Considere una interrupción inmediatamente antes de que entradaRC() pueda poner ocupada en 1.

4.4.2. Espera ocupada por turnos (alternancia)

Utiliza una variable global *turno*. Si *turno* es igual a i entonces un proceso P_i puede entrar. De esta manera se asegura que haya sólo un proceso en la región crítica. Exige **alternancia estricta**. Si *turno* = 0 y P_1 está listo para entrar entonces debe esperar aunque P_0 no éste en la región crítica.

- Este método es usado en esquemas de multiprocesadores para acceder a memoria compartida.
- Utiliza una variable global turno. Si turno es igual a i entonces proceso P_i puede entrar.
- Asumimos la existencia de dos procesos 0 y 1.

```
entradaRC(int i)
{
    while(turno != i);
}
```

```
salidaRC(int i)
{
    turno = 1 - i;
}
```

- Asegura que haya sólo un proceso en la Región Critica.
- Exige alternancia estricta. Si turno = 0 y P_1 está listo para entrar entonces debe esperar, aunque P_0 no esté en Región Critica.
- Usado en esquemas multiprocesadores para acceder a zonas de memoria compartida.

El problema aquí radica en que si un proceso tiene una sección no crítica muy larga el otro tendrá que esperar para entrar en su sección crítica a pesar que no se este ejecutando nada que la afecte.

4.4.3. Solución de Peterson

Esta solución es igual a la de alternancia, nada más que agrega un vector flag en donde cada proceso indica si esta interesado en entrar a la región crítica.

Se basa en una variable global señal (interesado[]) que indica la posición de cada proceso con respecto a la mutua exclusión y la variable global turno resuelve los conflictos de simultaneidad. Cada procedimiento realiza lo siguiente: pone su señal en cierto, y cede el turno al otro proceso, luego mientras la señal del otro proceso esté activado y él tenga el tuno se queda esperando, una vez que esto sea falso entrará en su sección crítica, y al salir pondrá su señal en falso.

- Asumimos la existencia de dos procesos 0 y 1.
- Utiliza dos variables

1. vector interesado[], donde cada proceso indica si está interesado en entrar.
2. variable turno igual que antes.

```
entradaRC(int i)
{
  interesado[i] = 1;
  turno = i;
  while(turno==i && interesado[1-i] == 1);
}
```

```
salidaRC(int i)
{
    interesado[i] = 0;
}
```

4.4.4. Algoritmo de Dekker

Resuelve el problema mediante una tabla unidimensional de dos elementos lógicos (switches). Es un algoritmo para la mutua exclusión de 2 procesos.

La idea de este algoritmo, son dos variables booleanas, una para c/proceso, que indican la voluntad de acceder a la sección critica y una variable extra que indica que proceso tiene derecho (de quien es el turno) a tratar de entrar (las tres variables son compartidas).

Un proceso que quiere entrar a la sección critica pone su bandera en TRUE y se fija en la bandera del otro, si esta en FALSE, quiere decir que puede ejecutar su sección critica. Si no, se fija en la 3° variable y si dice que es su turno volverá a revisar la bandera del otro. Si no fuese su turno queda atrapado en un ciclo hasta que lo sea.

El algoritmo escrito en Pascal es el siguiente:

```
program algoritmo_dekker;
var proceso_seleccionado: (primero, segundo);
   p1quiereentrar, p2quiereentrar: boolean;
procedure proceso_uno;
    begin
      while true do
         begin
           p1quiereentrar := true;
```

```
            while p2quiereentrar do
              if proceso_seleccionado = segundo then
                begin
                  p1quiereentrar := false;
                  while proceso_seleccionado = segundo do;
                  p1quiereentrar = true;
                end;
              región_crítica_uno;
              proceso_seleccionado := segundo;
              p1quiereentrar := false;
              otras_tareas_uno
            end
    end;

procedure proceso_dos;
    begin
      while true do
        begin
          p2quiereentrar := true;
            while p1quiereentrar do
              if proceso_seleccionado = primero then
                begin
                  p2quiereentrar := false;
                  while proceso_seleccionado = primero do;
                  p2quiereentrar = true;
                end;
              región_crítica_dos;
              proceso_seleccionado :=primero;
              otras_tareas_dos
            end
    end;
 begin
     p1quiereentrar := false;
     p2quiereentrar := false;
    proceso_seleccionado := primero;
    parbegin
      proceso_uno;
      proceso_dos
    parend
end.
```

4.4.5. Algoritmo de Lamport o de la panadería.

Lamport desarrolló una solución que es particularmente aplicable a los sistemas de procesamiento distribuido. El algoritmo usa un sistema de "toma de boleto de turno ", como el usado en las panaderías muy concurridas, y ha sido apodado el *algoritmo de la panadería de Lamport*. Nos centraremos únicamente en los aspectos del algoritmo relacionados con un entorno centralizado.

Al entrar en la tienda cada cliente recibe un número, y se atiende primero al que tenga el número menor. Por desgracia, el algoritmo de la panadería no puede garantizar que dos procesos (clientes) no reciban el mismo número. En el caso de un empate, primero se atiende el proceso con el nombre menor. Es decir, si Pi y Pj reciben el mismo número y si i < j, entonces primero se servirá a Pi. Como los nombres de procesos son únicos y ordenados, nuestro algoritmo es completamente determinista.
Las estructuras de datos comunes son:

```
var
  boleto: array[1..numero_procesos] of integer;
  eleccion: array[1.. numero_procesos] of boolean;
```

Al principio, a estas estructuras se les asigna un valor inicial 0 y *false*, respectivamente. Por conveniencia, definimos la siguiente notación:

n (a,b) < (c,d) si (a < c) o si (a = c) y b < d (1)

condición que determina si el proceso b con el boleto a es favorecido o no para entrar a la región crítica con respecto al proceso d con el boleto c. Nos permitimos dejar al lector la programación de este algoritmo que deberá tener al menos una función para los favorecidos que cumpla con la condición (1) y otra función que determine el número máximo de acuerdo a:

$$n \max(a_1, \ldots, a_n)$$

es un número, k, tal que $k \geq a_i$ para i=1, ...,n

4.4.6. Mecanismos provistos por el hardware:

Son instrucciones que se incorporan en el juego de instrucciones del procesador que aseguran la mutua exclusión.

1) Deshabilitar interrupciones.

En una máquina monoprocesador la ejecución de procesos concurrentes sólo puede intercalarse, por lo tanto un proceso continuará ejecutando hasta que solicite un servicio del S.O. o sea interrumpido. Como consecuencia para garantizar la mutua exclusión es suficiente con impedir que un proceso sea interrumpido, lo cual puede ofrecerse en forma de primitivas definidas por el kernel para inhabilitar o habilitar las interrupciones. El precio de esta solución es alto ya que se limita la capacidad del procesador para intercalar programas. Esta técnica no funciona en arquitecturas de multiprocesadores.

- Si se produce una interrupción por dispositivos u otra causa y ésta está deshabilitada, su tratamiento se retrasará hasta que se rehabiliten de nuevo.
- La mutua exclusión se logra ejecutando explícitamente la deshabilitación y cuando se deje la Región Critica rehabilitando las interrupciones.

2) Instrucciones especiales del procesador:

Se han propuesto varias instrucciones de máquina que realizan dos acciones atómicamente, tales como leer y escribir o leer y examinar sobre una misma posición de memoria en un único ciclo de lectura de instrucción. Puesto que estas acciones se realizan en un único ciclo de instrucción no están sujetas a injerencias por parte de otras instrucciones. Dos de las más habituales son:

a) ***Instrucción comparar y fijar (TAS o TSL)[1]:*** la instrucción examina el valor de su argumento, si es cero lo cambia por uno y devuelve cierto, en caso contrario el valor no se modifica y devuelve falso. El protocolo de mutua exclusión basado en esta instrucción es el siguiente, se da un valor inicial cero a una variable compartida cerrojo, el único proceso que puede entrar en su sección crítica es el que encuentre cerrojo en cero y los demás procesos que intenten entrar pasan a modo espera activa. Cuando un proceso abandona su región crítica vuelve a poner cerrojo en cero.

```
entradaRC()
{
    while (tsl(ocupada));
}

tsl(int flag)
{
    int x = flag;
    *flag = 1;
    return x;
}
```

```
salidaRC:
{
    ocupada = 0;
}
```

- TSL es una instrucción perteneciente al instruction set del procesador que en una operación atómica e indivisible lee y almacena un valor en una dirección de memoria.
- Generalmente utilizada en casos de múltiples procesadores.
- Puede ser utilizada con un solo procesador pero con duración corta en la Región Crítica y con baja frecuencia de ocurrencia.
- Es aplicable a cualquier número de procesos con memoria compartida tanto de monoprocesador como multiprocesador.
- Puede producir inanición y Deadlock.
- Es simple y fácil de verificar.
- Sirve para varias secciones críticas, cada una con su propia variable.

b) ***Instrucción intercambiar (CAS:* Compare And Swap*):*** esta instrucción intercambia el contenido de un registro con el de una posición de memoria. El protocolo de mutua exclusión basado en esta instrucción es el siguiente: se da un valor inicial cero a una variable compartida cerrojo, el proceso intercambia el valor de cerrojo por el de una variable con valor uno, y repite

[1] TAS es el acrónimo de **T**est **A**nd **S**et y TSL es **T**est and **S**et **L**ock.

esta operación hasta que el valor de la segunda variable sea cero, es decir que cerrojo valía cero antes del intercambio. En este caso el proceso entra en su región crítica y al salir de ésta vuelve a realizar el intercambio para poner cerrojo en cero.

- Existen algunas otras implementaciones para la instrucción CAS (Compare And Swap) que no utiliza un bloqueo por cerradura ni semáforos. Es eficaz en la actualización simple de variables globales. Lo que hace es copiar el valor de la variable global al espacio local (generalmente un registro de la CPU) y utiliza ese valor para calcular el valor nuevo (actualizado) que lo guarda en otro registro. Luego compara el valor de la variable guardada en el primer registro con el de memoria. Si son iguales actualiza la memoria con el valor del segundo registro y si son distintos vuelve a empezar toda la secuencia. Esta secuencia se hace atómicamente.

El uso de instrucciones especiales de máquina tiene las siguientes ventajas: es aplicable a cualquier número de procesos en sistemas con memoria compartida, tanto de monoprocesador como de multiprocesador, es simple y fácil de verificar y puede usarse para disponer de varias regiones críticas.

Las desventajas son: a) que se emplea espera activa, b) que puede producirse inanición o deadlock, ya que cuando un proceso abandona la sección crítica y hay más de un proceso esperando la selección es arbitraria y puede producirse deadlock si el proceso que está adentro de la sección crítica es interrumpido y comienza a ejecutarse otro proceso que también desea entrar en la región crítica este quedará bloqueado indefinidamente.

4.4.7. Cola de espera ($q_{(v)}$)

- Es un mecanismo de sincronización que provee un ordenamiento explícito (secuenciamiento) en el uso de la Región Critica.

Si varios procesos requieren usar un recurso crítico, el S.O. los coloca en una cola de espera.

- Reglas de uso:

1. Si un proceso no pudiera entrar en la Región Crítica se pondría en una cola de espera asociada a esa Región Critica.
2. Si un proceso sale de una Región Critica activaría a uno de los procesos de la cola de espera si la cola no esta vacía.
3. El valor inicial de la cola es vacía.

Fig. 4.08 Elementos de una cola

Se operan mediante dos primitivas del S.O. poner() y sacar(). Ambas funciones pueden ejecutarse concurrentemente por lo que la cola es un recurso crítico.

Primitivas son procedimientos o funciones estándar del S.O. (System Calls) que se ejecutan atómicamente como si fueran una sola instrucción (no son divisibles ni interrumpibles) y su mutua exclusión no aparece explícitamente en la ejecución.

4.4.8. Semáforos.

Un semáforo es una herramienta genérica de sincronización de procesos, o sea, permite el ordenamiento de las operaciones que realizan los procesos en el tiempo. Es una especie de bandera (señal o ***flag***) que indica la posibilidad de acceder o no a un recurso.

- Es una función o un arreglo dentro del Kernel.
- Un semáforo es una *variable protegida* cuyo valor puede ser accedido y alterado tan sólo por las dos **primitivas independientes y atómicas** y una operación de inicialización del semáforo.
- Un semáforo S es una variable entera $e_{(s)}$ llamada ***valor del semáforo*** con las siguientes propiedades:
  1. La variable puede tomar cualquier valor entero (positivo, negativo o nulo) en $\mathbf{e_{(s)}}$ (valor del semáforo).
  2. Se **crea** mediante un system call o una declaración, donde se especifica el valor inicial del semáforo que, por definición, debe ser un entero no negativo.

4. Sólo es accesible mediante dos operadores primitivos atómicos **down()** y **up()** (así los llama Tanenbaum)**.** Estas primitivas fueron creadas por Dijkstra y los llamó $\mathbf{P_{(s)}}$ (Proberen = comprobar en holandés) y $\mathbf{V_{(s)}}$ (Verhogen = soltar) respectivamente. Estas primitivas pueden tener distintos nombres. Por ejemplo **wait()** y **signal()** (Silberschatz), **pedir() y soltar()** en el CAOS, etc.

Nosotros utilizaremos la notación **P() y V)** en honor a su creador.

P()

```
P(int s)
{
    while (s <= 0);
    s--;
}
```

V()

```
V(int s)
{
    s++;
}
```

- La característica fundamental de estos operadores: **su ejecución es indivisible o sea atómica**.

Hay varios tipos de semáforos. Se destacan los ***semáforos binarios*** sólo pueden tomar los valores 0 y negativos. Los ***mutex*** que pueden tomar valores 1, 0 y negativos y los ***semáforos contadores o generales*** pueden tomar valores enteros no negativos, cero y negativos.
La operación **P** en el semáforo S, (se escribe P(S)), opera de la siguiente manera:

```
if S > 0
        then S=S - 1
        else (espera en S)
```

La operación **V** en el semáforo S, (se escribe V(S)), opera de la siguiente manera:

```
if (uno o más procesos están en espera en S)
        then (deja proseguir a uno de estos procesos) y
        else (sigue en secuencia) luego hace S=S+1
```

Supondremos que hay una disciplina de colas del primero en entrar - primero en salir (FIFO) para los procesos que esperan a completar un P(S). La ejecución de las instrucciones P y V son indivisibles. La mutua exclusión en el semáforo S, es aplicada en P(S) y V(S). Si varios procesos intentan ejecutar P(S) al mismo tiempo, sólo uno podrá proseguir; los otros permanecerán en espera. Los semáforos y las operaciones de semáforos pueden implementarse en software o hardware. En general, se implementan en el Kernel del sistema operativo, donde se controlan los cambios de estado de un proceso.

4.5. Algoritmos sin espera activa

- Para evitar el problema de la espera activa diseñamos algoritmos que además del protocolo de sincronización cumplan con las siguientes reglas:
  1. Si un proceso no puede entrar en la Región Critica, pasa a una cola de espera asociada al semáforo, ejecutando una primitiva poner()como la indicada en la Figura 4.08.
  2. Si un proceso sale de su Región Critica, activa a uno de los procesos de la cola de espera asociada al semáforo ejecutando sacar(), si es que no está vacía.
  3. La cola está vacía inicialmente.
- Dos nuevas funciones se necesitan,
  1. **block**(): añade el proceso actual a la cola y lo pone en estado inactivo.
  2. **wakeup**(): Activa o despierta un proceso de la Q; lo pasa de detenido a listo. La elección de que proceso activar no es responsabilidad de la función sino del Kernel.

Estas primitivas también se ejecutan atómicamente

4.5.1. Semáforos sin espera activa

- En lugar de una variable entera simple, utilizamos una estructura con dos elementos: La variable que indica el valor del semáforo y una cola de procesos asociada al semáforo.

```
typedef struct
{
        int   valor;
        colaP Q;
}
semáforo;
```

```
P(semáforo s)
{
    s.valor--;
    if (s.valor < 0)
        {
        enqueue(s.Q);
        block();
        }
}
```

```
V(s)
{
    s.valor++;
    if (s.valor <= 0)
        {
        p = dequeue(s.Q);
        wakeup(p);
        }
}
```

- Propiedades:
  1. Valor inicial ≥ 0, puede hacerse negativo luego de un número determinado de **V().**
  2. En monoprocesador, para asegurar ejecución atómica, el S.O. inhibe interrupciones antes de **P().** En un esquema multiprocesador esto no se puede hacer ¿por qué? (por las características de las interrupciones en multiprocesamiento).
  3. Sea np(s): cantidad de instrucciones **P()** ejecutadas en el semáforo.
     nv (s): cantidad de instrucciones **V()** ejecutadas.
     e_i (s): estado inicial del semáforo.
     e(s): estado actual del semáforo
     Entonces siempre se cumple **e(s) = e_i(s) - np(s) + nv(s)**
  4. Si e(s) < 0 entonces |e(s)| es el número de procesos que están esperando para usar el recurso (ver Figura 4.09).
  5. Si e(s) ≥ 0 luego e(s): cantidad de procesos que pueden franquear el semáforo sin ser bloqueados.

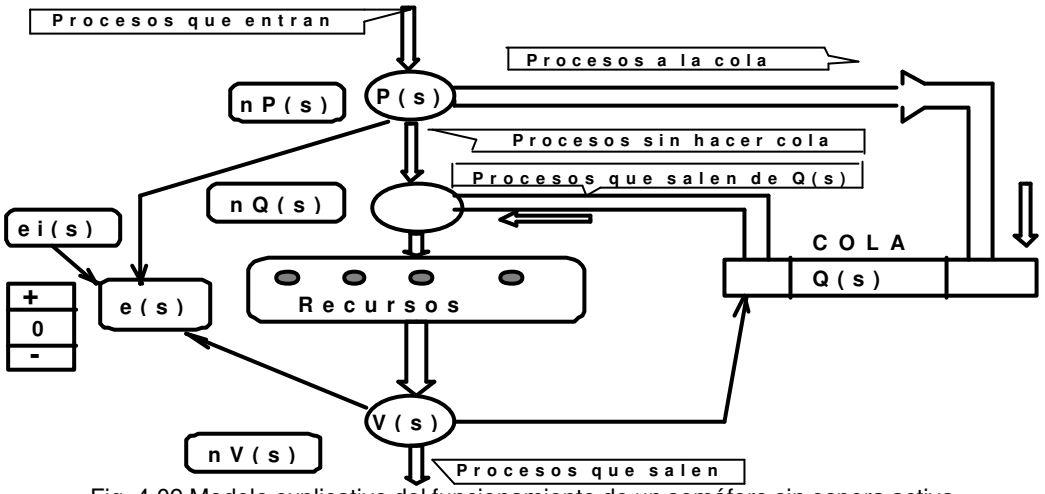


Fig. 4.09 Modelo explicativo del funcionamiento de un semáforo sin espera activa

  6. Sea nQ(s): cantidad de procesos que han ejecutado la primitiva **P()** (es decir que no se han bloqueado por ella o se han bloqueado primero y después desbloqueado). En todo momento tendremos:

     nQ(s) ≤ nP(s)

- El valor del semáforo indica la cantidad de puntos de entrada que tiene el recurso.
- Un semáforo con valor 1 indica un recurso crítico, por lo tanto se lo llama semáforo **mutex** o de mutua exclusión y si su valor es cero es un semáforo **binario** (sirve para contar mensajes u otros elementos dado por el valor absoluto del semáforo).
  Los problemas de la mutua exclusión son resueltos por medio de una operación **P()** antes de entrar y una operación **V()** al salir y el valor del semáforo inicializado en 1.

Disciplinas en el manejo de las colas de procesos bloqueados

FIFO: cuando se ejecuta un signal, se desbloquea el primer proceso de la cola (el más antiguo).
RANDOM: Se desbloqueará cualquier proceso que esté en la cola de bloqueados, independientemente de su antigüedad.
Con la disciplina FIFO, se sabe que, independientemente de la longitud de la cola de procesos en espera, le legará su turno. En cambio, con la disciplina RANDOM, no se sabe cuándo llegará su turno.

Construcciones del lenguaje

- Desventajas o problemas del uso de las primitivas anteriores:
  1. Intercambiar una **P()** y una **V()**
  2. Poner 2 **P()**
  3. No poner una **P()** y, sí, una **V()**
- Algunos lenguajes incluyen construcciones especiales para sincronizar procesos. Uno de ellos es Concurrent Euclid (Holt, 1983).

Ejemplo de uso de semáforos con n procesos concurrentes:

```
Valor = 1;
Pi ()
 {
    P (semáforo (s));
      Región Critica;
    V(s);
  }

cobegin
   P1; P2; .....; Pn
Coend
```

Implementación de los semáforos: al ser el semáforo un recurso crítico sólo un proceso lo puede manipular, ya sea con una operación P() o con V(). Para asegurar la mutua exclusión se pueden implementar las operaciones P() y V() como primitivas atómicas. Una forma obvia es realizarlo por hardware o por firmware, si no se puede realizarlo utilizando las técnicas de software o hardware para la mutua exclusión: los algoritmos de Dekker o Peterson, o utilizar la instrucción comparar y fijar. Para un monoprocesador se puede utilizar la inhibición de interrupciones.

4.5.2. Regiones críticas condicionales (CCR) (Brinch Hansen 1972)

Brinch Hansen dio este nombre a una solución al problema de la mutua exclusión. La idea es crear una estructura de datos compartida que se implemente a nivel de compilador. El compilador va a generar un código que garantice la mutua exclusión en los accesos a los recursos compartidos. Para ello:

- Se declara una variable indicando que va ser de uso compartido
  var v: shared Type;
- La variable v sólo puede ser accedida dentro de la sentencia region, o sea, que las instrucciones que utilizan la variable dentro de "regiones" asociadas a esa variable serán ejecutadas bajo mutua exclusión
  region v do S;
- Mientras S se ejecuta, nadie puede acceder a v. Si un proceso pretende entrar a una región crítica ocupada, se bloquearía y se pondría en la cola correspondiente. Las regiones críticas no disponen de sincronización de condiciones, lo que limita su uso.
- S es ejecutado como una región crítica. El compilador se encarga de insertar una **P()** y una **V()** alrededor de S.

La región crítica condicional, en realidad, es una extensión de la anterior en que un proceso puede esperar fuera de la región crítica hasta que se satisfaga una condición dada.

La variable compartida se declara de la siguiente forma:

region v when E do S;

Si además de estar v disponible, se cumple E (una condición arbitraria) entonces entra a la región y ejecuta S. region se usa para controlar el acceso.

Brinch Hansen sugirió que para implementar las CCR se emplearán dos colas. La principal contendría los procesos que han encontrado ocupada la región crítica, mientras que la secundaria se ocuparía de los procesos que habiendo superado la condición de exclusión, no cumplen la condición.

Al liberarse la región crítica, los procesos de la cola secundaria pasarían a la principal y competirían para entrar en la región crítica.

Se podría optimizar el procedimiento limitando la verificación de la condición solamente cuando se hayan producido cambios que pueden hacer que se cumpla la condición.

Aunque aportan ventajas respecto de los semáforos, presentan algunos inconvenientes:

- Pueden estar dispersas por el texto del programa. Solución: agrupar los accesos a variables compartidas

- La integridad de una estructura compartida puede ser dañada, ya que no hay control sobre las operaciones llevadas a cabo sobre ellas. Solución: limitar el acceso mediante una serie de operaciones
- Difíciles de implementar

4.5.3. Monitor (Hoare 1974)

Los monitores son un mecanismo de sincronización implementada por una primitiva de alto nivel (Mediante compilador) dentro de la cual se dispone de un conjunto de procedimientos, variables y estructuras de datos agrupados en un módulo.

Los recursos globales compartidos pueden ser declarados como pertenecientes a un monitor y ningún proceso tendrá permiso para accederlo. En cambio sí pueden accederlo a través de un (y solo un) procedimiento público proporcionado por el monitor y suministrar datos producidos como argumento.

El monitor manipula los datos, o sea los encapsula para que los procesos del usuario no pueda accederlos en forma directa.

La variable monitor debe estar implementada en el compilador.

- Propiedad: sólo un proceso puede estar activo en un monitor en cualquier instante.
- Un monitor se escribe como:

a) un conjunto de declaraciones de variables
b) un conjunto de procedimientos
c) un cuerpo de comandos que se ejecutan inmediatamente después de la inicialización del programa que contiene el monitor.

- La sintaxis de un monitor es la siguiente:

```
type nombre = monitor
  begin
    ...declaración de variables...

    procedure P1(parámetros); /* procedimientos públicos o privados */
      begin
                ...          /*cuerpo del procedimiento*/
      end;
    .
    .
    .
    procedure Pn();
      begin
      ...
      end;

    begin
      código de inicialización del monitor
    end.
```

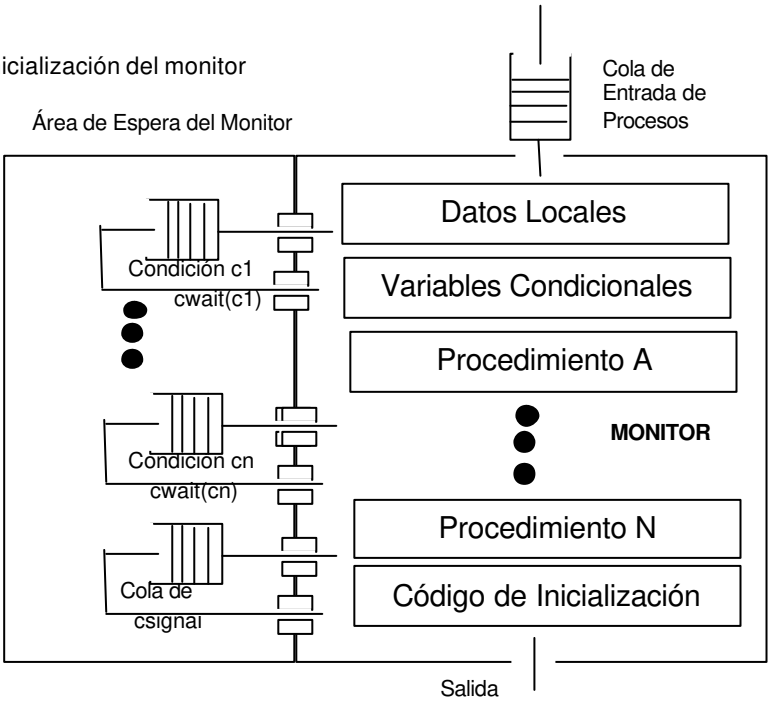


Fig. 4.10. Estructura del monitor de señales

- Los monitores son estructuras estáticas que solo se activan cuando uno de sus procedimientos es invocado desde el exterior.
- Cuando uno de los procedimientos del monitor está ejecutando, todos los otros están bloqueados encolados. Esto garantiza la mutua exclusión.
- El monitor, internamente puede tener un conjunto de variables globales, pero externamente estas variables son locales a monitor.
- Las variables declaradas dentro del monitor sólo pueden ser accedidas por los procedimientos del monitor.
- La comunicación entre el monitor y el mundo exterior se realiza sólo mediante los parámetros de los procedimientos.

- **Monitores con señales:** un monitor es un módulo de software que consta de uno o más procedimientos, una secuencia de inicialización y unos datos locales. Las características básicas son:
  1. Las variables de datos locales están sólo accesibles para los procedimientos del monitor y no para procedimientos externos.
  2. Un proceso entra en el monitor invocando a uno de sus procedimientos.
  3. Sólo un proceso puede estar ejecutando en el monitor en un instante dado; cualquier otro proceso que haya invocado al monitor quedará suspendido (en una cola de entrada) mientras espera que el monitor esté disponible.

     Si se cumple la norma de un proceso cada vez, el monitor puede ofrecer un servicio de mutua exclusión. Las variables de datos del monitor pueden ser accedidas sólo por un proceso cada vez. Una estructura de datos compartida puede protegerse situándola dentro del monitor. Si los datos del monitor representan algún recurso, el monitor ofrecerá un servicio de mutua exclusión en el acceso a este recurso. Los monitores deben incluir herramientas de sincronización para que resulten útiles en el proceso concurrente. La sincronización se produce por medio de las variables de condición que se incluyen dentro del monitor y que sólo son accesibles desde adentro.

     Existen dos funciones:
     1. cwait(c): suspende la ejecución del proceso llamado bajo la condición c. El monitor está ahora disponible para ser usado por otro proceso.
     2. csignal(c): reanuda la ejecución de algún proceso suspendido después de un cwait bajo la misma condición. Si hay varios procesos, elige uno de ellos; si no hay ninguno no hace nada.

     El monitor tiene un único punto de entrada que está custodiado para que sólo un proceso pueda entrar en él en un instante. El resto de los procesos que intente entrar al monitor se añadirán a una cola de procesos suspendidos a la entrada mientras esperan a que el monitor esté disponible. Una vez que un proceso está dentro del monitor, puede suspenderse a sí mismo bajo cierta condición, entonces se sitúa en una cola de procesos que esperan en un Área de Espera del Monitor para volver a entrar al monitor cuando cambie la condición (cwait). Si un proceso que está ejecutando en el monitor detecta un cambio en una variable de condición, avisa a la cola de condición correspondiente de que la condición ha cambiado (csignal). La misma estructura del monitor garantiza la mutua exclusión, sin embargo se deben colocar correctamente las primitivas cwait y csignal. Una ventaja del monitor con respecto al semáforo es que en el caso del semáforo tanto la mutua exclusión como la sincronización son responsabilidades del programador, en el monitor en cambio todas las funciones de sincronización están confinadas dentro de éste. Además una vez que un monitor está correctamente programado, el acceso al recurso protegido es correcto para todos los procesos. Con los semáforos en cambio el acceso sólo será correcto si todos los procesos están correctamente programados.
- **Monitores con notificación y difusión[2] o Mesa:** los monitores anteriores presentan un defecto, éste es el siguiente cuando un proceso ejecutaba un csignal debía salir inmediatamente del monitor o suspenderse en el mismo. Así pues se tenían dos inconvenientes:
  1. Si el proceso que ejecuta el csignal no abandona el monitor hacen falta dos cambios de contexto adicionales: uno para suspender el proceso y otro para reanudarlo cuando el monitor quede disponible.
  2. La planificación de procesos asociada con las señales debe ser muy fiable. Cuando se ejecuta un csignal, debe activarse automáticamente un proceso de la cola de condición correspondiente y el planificador debe asegurarse de que ningún otro proceso entre al monitor antes de la activación. Si no es así, la condición bajo la que se ha activado el proceso podría cambiar.

[2] Versión de Lampson & Redell

El nuevo monitor provee un método diferente que supera los problemas mencionados. La primitiva csignal se reemplaza por cnotify. Cuando un proceso que está en el monitor ejecuta un cnotify, origina una notificación hacia la cola de cierta condición, pero el proceso que dio la señal puede continuar ejecutando. El resultado de la notificación es que el proceso de la cabeza de la cola de esa condición será reanudado en el futuro cercano, cuando el monitor esté disponible. Sin embargo, esto no garantiza que ningún otro proceso entre al monitor antes que el proceso que espera, por lo tanto este proceso debe volver a comprobar la condición. Una modificación útil que se puede asociar a la primitiva cnotify es establecer un temporizador de guarda. Un proceso que ha estado esperando durante el intervalo máximo de guarda será situado en el estado de listo independientemente de si se ha notificado la condición. Cuando el proceso se active, éste comprueba la condición y continua ejecutando si es que ésta se cumple. El temporizador entonces lo que impide es la inanición indefinida de un proceso. Siguiendo la norma de notificar a los procesos en lugar de reactivarlos a la fuerza, se puede añadir una primitiva de difusión cbroadcast. Esta primitiva provoca que todos los procesos que están esperando por una condición se sitúen en el estado listo. Las dos ventajas que presenta este tipo de monitores con respecto al anterior son que el segundo es menos propenso a errores y además presenta un método más modular de construcción de programas.

Desventajas:

- Son un concepto del lenguaje de programación.
- No sirven para procesos distribuidos.

4.6. Comunicaciones entre procesos (IPC – Inter Process Communication)

Para que los procesos puedan cooperar entre si, para intercambiar datos, es fundamental la comunicación. La forma mas eficiente de implementar esto es establecer una comunicación cliente servidor, de modo que cada mensaje llegue al receptor y no tenga que verificar cada proceso si el mensaje es para el o no. La forma más simple de implementar esta comunicación es en forma sincrónica, en la que tanto el receptor como el emisor se bloquean mientras se realiza la transmisión del mensaje. Pero esto desperdicia tiempo de CPU. La solución a esto es utilizar un buffer como intermediario, en un sistema productor – consumidor, pero esto aumenta la complejidad del proceso ya que el buffer se convierte en una región critica.

4.6.1. Mensajes

La comunicación entre dos procesos puede ser realizada de dos maneras:

1. Comunicación a través de un área común de memoria.
2. Comunicación mediante el intercambio de mensajes llamado paso de mensajes.

En el primer caso se requiere contar con mecanismos de sincronización (semáforos, monitores, etc.) para garantizar la consistencia de los datos almacenados. Ejemplo Consumidor - Productor que se explica en el punto 4.6.4.

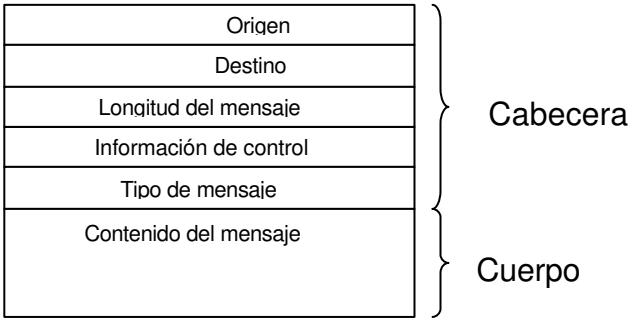
En el segundo caso las variables son locales.

- El propósito es permitir que dos procesos se sincronicen o se envíen datos mediante un mecanismo explícito.

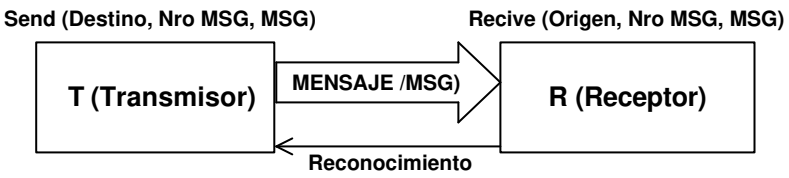
Definimos como **mensaje** a una porción discreta de datos (generalmente compuesto por un conjunto de bits). Los mensajes pueden ser de tamaño fijo o variable. En el segundo caso, suelen tener una cabecera de tamaño fijo y un cuerpo, de tamaño variable

Los mensajes tienen una cabecera (header) y un cuerpo. El header tiene el identificador del transmisor, el identificador del receptor, la longitud, y el tipo. El cuerpo del mensaje es el contenido generalmente expresado en bytes.

- Se usan dos primitivas:
  1. send(destino, mensaje, N° MSG)
  2. receive(fuente, &mensaje, N° MSG)



Formato típico de mensaje



Velocidades distintas (generalmente T>R) se requiere un buffer para amortiguar las fluctuaciones de velocidad relativa entre procesos T y R.

Fig. 4.11 Modelo de comunicación entre dos entidades T y R

Observación: el Nº MSG se requiere por el extravío de los mensajes. Una solución es disponer, como retorno, un mensaje de reconocimiento que devuelve el receptor al transmisor. Pero este mensaje también se puede perder por lo que se puede resolverlo de la siguiente manera:

1. Si al cabo de un tiempo T no se recibe el reconocimiento, se retransmite el mensaje.
2. Entonces R tiene repetido dos mensajes iguales, por lo que R debe distinguir entre estos dos mensajes iguales, por eso se requiere Nº MSG. Consecutivos, en cada MSG original.

- Si T y R se quieren comunicar entonces necesitan un vínculo de comunicación (**communication link**).

  Cada vínculo tendrá las siguientes características:
  1. Nombre de los procesos asociados (Origen T y Destino R)
  2. Capacidad del vínculo (buffer space).
  3. Tamaño de cada mensaje (variable o fijo).
  4. Dirección (bi o unidireccional).

Existen dos tipos de comunicaciones entre procesos: las directas y las indirectas.

* *Comunicación directa*

Los procesos envían y reciben los mensajes entre sí. Dependen de las velocidades relativas entre sí (si son distintas requieren un ***buffer de mensajes*** para su sincronización).

send(P, mensaje): envía mensaje a P
receive(Q, mensaje): recibe mensaje de Q

Propiedades:

1. Se establece un vínculo automáticamente entre los procesos. Sólo deben conocerse mutuamente.
2. El vínculo se establece con 2 procesos exactamente.
3. El vínculo es bidireccional.

```
p()                          |  c()
{                            |  {
    while(1)                 |      while(1)
        {                    |          {
        x = producir();      |          recibir(c, &x);
        send(c, x);          |          consumir(x);
        }                    |          }
}                            |  }
```

Desventajas:

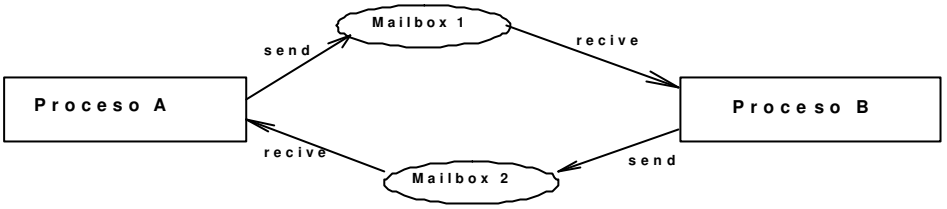
1. Poca modularidad (rígido).
2. Si se cambia la identificación de un proceso (PID), puede ser necesario revisar el código de los otros.

* *Comunicación indirecta*

Los mensajes son enviados a un ***buzón o mailbox*** y se retiran del buzón como se indica en la Figura 4.12.

1. Los mensajes van y vienen a **buzones (mailbox)** o **puertos**.
2. Cada buzón tiene su propia identificación, luego los procesos se comunican compartiendo una determinada cantidad de buzones.
3. Para comunicarse deben tener un buzón compartido.
   send(A, msg): envía mensaje al buzón A.
   receive(A, &msg): busca mensaje del buzón A.
4. Propiedades
   a) El vínculo sólo se establece si ambos procesos comparten un mailbox.
   b) Link puede ser asociado con más de 2 procesos.

c) Entre cada par de procesos puede haber varios links distintos cada uno correspondiente a un mailbox.
d) Un link puede ser uni o bidireccional.
e) La disciplina del mailbox es generalmente FIFO o de prioridad basada en el tipo de mensaje. Otra alternativa es permitir al receptor inspeccionar la cola y elegir el mensaje que va a levantar.



Mailbox: interfase entre procesos y S.O.. Se crea y quita facilmente.

Procesos: A Pide un mailbox y envía MSG, B se activa y recibe MSG

Fig. 4.12 comunicación indirecta

4.6.2 IPC: Inter Process Communication

Paso de mensajes:

La sincronización y la comunicación son los principios que se deben satisfacer cuando los procesos interactúan. El paso de mensajes cumple esto y además puede ser implementado en sistemas distribuidos, así como en sistemas multiprocesador y monoprocesador de memoria compartida.
El paso de mensajes generalmente se realiza a través de dos primitivas **send(destino, mensaje, Nº de Mensaje)** y **receive(origen, mensaje, Nº de Mensaje)** en vez de la lectura o escritura de variables compartidas.

La comunicación se debe a que un proceso al recibir un mensaje, obtiene los datos enviados por otro proceso y la sincronización se produce porque un mensaje solo puede ser recibido después de ser enviado. Generalmente después de recibido el mensaje se destruye. Si se desea conservarlo se debe tomar los recados adecuados.

Los envíos de mensajes pueden ser realizados por distintas primitivas send en que se debe contemplar la situación en que permanecerá el transmisor. Por ejemplo:

- **send asincrónico**: El proceso transmisor envía un mensaje y continúa con otra tarea.
- **send sincrónico:** El transmisor envía el mensaje y se bloquea hasta que el receptor le indique que lo ha recibido.
- **send condicional:** El proceso transmisor envía un mensaje y continúa con otra tarea. Si el receptor no está disponible devuelve un código de retorno al transmisor para indicar que el mensaje no ha llegado a destino; caso contrario el retorno indicara que sí.
- **receive (incondicional o bloqueante):** El receptor queda bloqueado hasta recibir el mensaje.
- **receive condicional (polling):** El receptor pregunta a los canales de comunicaciones si existe algún mensaje para él. En caso negativo, la primitiva receive devuelve un código de retorno que indica la no existencia de mensajes en ese canal para él.

Existen cuestiones de diseño que son relativas al paso de mensajes:
Sincronización: el receptor no puede recibir un mensaje hasta que sea enviado por otro proceso. Tanto el emisor como el receptor pueden ser bloqueantes o no bloqueantes. En base a esto se pueden generar las siguientes combinaciones:

- Envío bloqueante, recepción bloqueante (**Rendez Vouz** que se describirá más adelante): esta combinación permite una fuerte sincronización entre procesos ya que ambos, el emisor y el receptor, se bloquean hasta que se entrega el mensaje.
- Envío no bloqueante, recepción bloqueante: permite que un proceso envíe uno o más mensajes a varios destinos tan rápido como sea posible. El receptor se bloquea hasta que llega el mensaje solicitado. Esta es la combinación más útil.
- Envío no bloqueante, recepción no bloqueante: nadie debe esperar. Como no hay bloqueo para hacer entrar en disciplina al proceso, esos mensajes pueden consumir recursos del sistema, incluido tiempo del procesador y espacio en buffer, en detrimento de otros procesos y del S.O..

Direccionamiento: es necesario especificar en la primitiva send qué proceso va a recibir el mensaje. De forma similar, la mayoría de las implementaciones permiten indicar a los procesos receptores el origen del mensaje que va a recibir.

- **Direccionamiento directo:** la primitiva send incluye una identificación específica del proceso destino, la primitiva receive puede gestionar de dos formas. Una posibilidad requiere que el

proceso designe explícitamente un proceso emisor, por lo tanto el proceso debe conocer de antemano de que proceso espera un mensaje. En otros casos, es imposible el proceso dé el origen por anticipado. El parámetro origen de la primitiva receive tendrá un valor de retorno cuando se haya realizado la recepción.

- **Direccionamiento indirecto:** los mensajes no se envían directamente del emisor al receptor, sino a una estructura de datos compartida formada por colas que guardan los mensajes temporalmente. Esta se llama buzón (**mailbox**). Para que dos procesos se puedan comunicar deben compartir un buzón. Una ventaja de este direccionamiento es que permite mayor flexibilidad en el uso de los mensajes y la relación entre el emisor y el receptor puede ser de uno a uno, de uno a muchos, de muchos a uno y de muchos a muchos. Una relación uno a uno permite establecer un enlace privado de comunicaciones entre dos procesos. Una relación de muchos a uno resulta útil cuando un proceso ofrece un servicio a un conjunto de procesos. En este caso el buzón se denomina **puerto**. Una relación uno a muchos permite un emisor y muchos receptores, es útil para aplicaciones en que un mensaje se debe difundir a un conjunto de procesos. La asociación de procesos a buzones puede ser estática o dinámica. Los **puertos son estáticos**. El puerto se crea y se asigna al proceso permanentemente. En una relación uno a uno, el buzón también es permanente. Cuando hay varios emisores, la asociación de un emisor a un buzón puede realizarse dinámicamente. Pueden usar primitivas como conectar y desconectar para la asociación dinámica. En el caso de un puerto, pertenece y es creado por el proceso receptor, entonces cuando se destruye el proceso también se destruirá el puerto. Para el caso general de los buzones el S.O. puede ofrecer un servicio de creación de buzones. El buzón puede ser propiedad del proceso creador, en cuyo caso se destruye junto con el proceso creador, o pueden ser considerados como propiedad del S.O. en cuyo caso se necesita una orden explícita para destruir el buzón.

Formato de mensajes: varía según los objetivos del servicio de mensajería. Pueden ser mensajes cortos y de tamaño fijo, minimizando el procesamiento y el costo de almacenamiento. Si se va a pasar una gran cantidad de datos, los datos pueden ponerse en un archivo y el mensaje hará referencia al archivo. Una solución más flexible es permitir mensajes de longitud variable. Así el mensaje, se divide en dos partes: una cabecera que contiene origen, destino, longitud del mensaje, información de control, y el tipo de mensaje, y un cuerpo que contiene el contenido del mensaje.

Disciplina de cola: la alternativa más simple es la cola FIFO. Sin embargo si hay mensajes que son más urgentes que otros una alternativa es asignarles prioridad, y otra alternativa es permitir al receptor examinar la cola de mensajes y designar cual quiere sacar.

Mutua Exclusión: se usan un receive bloqueante y un send no bloqueante. Un conjunto de procesos concurrentes comparten un buzón. El buzón contiene inicialmente un único mensaje de contenido nulo. Un proceso que desea entrar en su sección crítica intenta primero recibir el mensaje, si el buzón está vacío el proceso se bloquea; una vez que un proceso ha conseguido el mensaje, ejecuta su sección crítica y devuelve el mensaje al buzón. El mensaje funciona como testigo que se pasa de un proceso a otro. Si varios procesos ejecutan el receive concurrentemente, si hay un mensaje se entrega a un solo proceso y los otros se bloquean, y si no hay mensajes todos los procesos se bloquean y cuando haya un mensaje sólo uno de los procesos bloqueados se activa y toma el mensaje. El problema del productor/consumidor con buffer acotado se puede resolver utilizando dos buzones. A medida que el productor genera datos los envía como mensajes al buzón "puede consumir", con tal de que haya un mensaje en ese buzón el consumidor podrá consumir. Este buzón hace de buffer. Inicialmente el buzón "puede producir" se llena con un número de mensajes nulos igual a la capacidad del buffer, el número de mensajes de puede producir se reduce con cada producción y crece con cada consumo. Puede haber varios productores y consumidores siempre que todos tengan acceso a ambos buzones.

4.6.3. Tipos de sincronizaciones mediante mensajes.

Las primitivas de comunicaciones vistas en el punto anterior permiten clasificarlas en sincrónicas o asincrónicas según se bloqueen o no a los procesos que ejecutan la emisión o recepción de un mensaje.

- El lenguaje ADA propone las siguientes soluciones:

a) Comunicación Sincrónica

Rendez-Vous:

Ejecutar a ciertas sentencias dentro de procedimientos remotos o no. Como el emisor y el receptor *se reúnen* para una **comunicación síncrona**, a menudo se le llama ***rendezvous* o *Rendez-vouz***.

El proceso emisor es bloqueado hasta que el receptor esté listo para recibir el mensaje. Cuando el proceso receptor ejecuta el receive y el mensaje no se encuentra disponible queda bloqueado hasta la

llegada del mismo. Una vez que se ha producido el intercambio de mensajes ambos procesos continúan su ejecución concurrentemente.

Con semáforos puede ser para **t** transmisor y **r** receptor:

```
V(t);                                   P(t);
P(r);                                   V(r)
send(msg);  ____________________________  receive(msg);
```

Rendez-Vous extendido:

Es una extensión del mecanismo anterior, con la diferencia de que el proceso receptor solamente envía una respuesta al transmisor después de la ejecución de un cuerpo de comandos que operan sobre el mensaje recibido. Mientras tanto el transmisor queda bloqueado a la espera hasta que el cuerpo de comandos haya terminado sus tareas. La respuesta puede poseer parámetros que contengan resultados de los cálculos efectuados por el receptor.

Con semáforos puede ser para **t** transmisor y **r** receptor:

```
V (t);                                  P(t);
send();  ______________________________  receive();
P (r);                                  if  ok then V(r);
```

La **invocación remota** se conoce como ***rendezvous extendido*** ya que se pueden realizar operaciones arbitrarias antes de que se envíe la respuesta (es decir, durante el *rendez-vous*).

Rendez-Vous asimétrico:

Aquí solamente el transmisor (que oficia de cliente) nombra al proceso receptor (server). La primitiva receive es reemplazada por el comando **accept**. Ambos procesos quedan en rendez-vouz hasta que se complete la ejecución de todo el cuerpo del comando accept. El accept no nombra al transmisor **t** debido a que la comunicación ya fue establecida.

Los parámetros pueden ser de entrada, de salida o de entrada/salida (es semejante a la llamada de un monitor). Ejemplo:

```
t()                                     r()
{                                       {
send(r,msg);  ____________________________  accept send(msg);
}                                       y = msg;
                                        }
```

b) ***Comunicación Asincrónica:*** Las primitivas de este tipo de comunicación se caracterizan por no bloquear a los procesos que las ejecutan. Así cada uno sigue su ejecución. Esto es importante en el caso del receptor ya que sigue ejecutando aunque no le llegue ningún mensaje. Depende de la implementación si los mensajes siguientes serán atendidos o no.

c) *Comunicación semi-sincrónica:* Se usa un send no bloqueante y receive bloqueantes. Esto es riesgoso pues se pueden acumular una gran cantidad de mensajes en colas.

d) *otros modos*

Existen otros mecanismos en que se pasan mensajes y parámetros a procedimientos remotos (***RPC Remote Procedure Calls)***. Pueden usar memoria compartida o paso de mensajes. En la siguiente tabla se resume las características de cada uno.

En resumen:

PASO DE MENSAJES	MEMORIA COMPARTIDA
-Medio de comunicación entre procesos que ofrece el S.O. -Se debe abrir una conexión y conocer el nombre del comunicador (proceso en la misma CPU o no) -En red cada computador tiene el **nombre del anfitrión** y cada proceso tiene **nombre de proceso.** -La fuente de la comunicación conocida como **cliente** y el demonio (daemons) que recibe **servidor**, intercambian mensajes mediante llamadas al sistema. -Para intercambiar pequeñas cantidades de datos. -Evita conflictos. -Más fácil de usar para comunicación entre computadoras.	-Utiliza llamadas para **correspondencia de memoria** con el propósito de obtener acceso a los registros de memoria que pertenecen a otros procesos. -S.O. evita que un proceso tenga acceso a la memoria de otro. Para compartir la memoria se requiere que dos o más procesos estén de acuerdo en eliminar esta restricción y así intercambiar información leyendo y escribiendo datos en estas áreas compartidas. -Forma de datos y ubicación están determinadas por esos procesos y no por el S.O., los cuales también se encargan que no se escriba en posiciones iguales de memoria a la vez. -Máxima velocidad y conveniencia en la comunicación (puede efectuarse a la velocidad de la memoria) -Problemas en el área de protección y sincronización.

Tabla 4.01 comparación de paso de mensajes con memoria compartida

4.6.4. Modelo productor-consumidor (bounded buffer)

- Usado para describir dos procesos ejecutando en forma concurrente:
  1. **Productor**: Genera un conjunto de datos necesarios para la ejecución de otro proceso.
  2. **Consumidor**: Toma los datos generados por el productor y los utiliza para su procesamiento.
- Ejemplos:
  1. ls | more
  2. El spooler del S.O. y los procesos que quieren imprimir.

Productor: Genera elementos mediante **producir()** y los ingresa en el buffer mediante **depositar().**
Buffer:Zona de memoria utilizada para amortiguar las diferencias de velocidad entre dos procesos. Almacena temporalmente los elementos generados por **p**.
Consumidor: Retira los elementos del buffer mediante **recuperar()** y los consume con **consumir().**

- Restricciones:
  1. El productor no puede sobrepasar, el envío de mensajes en uno, la capacidad del buffer
  2. El consumidor no puede consumir mensajes más rápido de lo que se produce.

- Se utiliza el siguiente ***protocolo de sincronización:***
  1. Si el productor quiere poner un elemento en un buffer lleno entonces es demorado hasta que el consumidor saque uno.
  2. Si el consumidor intenta tomar mensajes de buffer vacío entonces es demorado hasta que el productor ingrese uno.

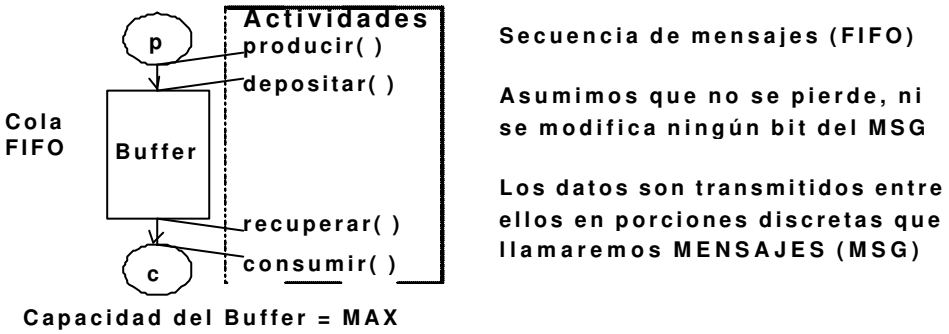


Fig. 4.13 Modelo productor consumidor

- La secuencia de mensajes enviados y recibidos son dos arreglos infinitos **R** (mensajes recibidos) y **S** (mensajes enviados) con índices **r** y **s**.
  Deben cumplir con las siguientes condiciones:
  1. El número de mensajes recibido no puede exceder el número de mensajes enviados:
     $0 \le r \le s \le MAX$
  2. El N° de Mensajes enviados, pero aún no recibidos, no pueden exceder la capacidad máxima del Buffer (MAX):
     $0 \le s - r \le MAX$
  3. Los mensajes deben recibirse exactamente en el orden en que son enviados:
     $\text{for } \forall\, i: 1 \le i \le s,\ S[i] = R[i]$
     Se puede plantear el siguiente invariante de comunicaciones:
     $0 \le r \le s \le r + MAX$ (**Invariante de comunicación**)
     $\text{for } \forall\, i: 1 \le i \le s,\ S[i] = R[i]$
  4. El consumidor deberá sacar todo los mensajes de una sola vez.

4.6.5. Algunos algoritmos para el modelo productor - consumidor

- Los algoritmos modelan el comportamiento del productor (**p**) y del consumidor (**c**) de forma tal que se cumpla el protocolo de sincronización.

Observación: utilizamos **p** (minúscula) para productor y P (mayúscula) para la primitiva del semáforo.

A. Con sleep() & wakeup()

- **sleep():** Llamada al sistema que bloquea al proceso solicitante.
- **wakeup():** tiene como parámetro el PID del proceso a desbloquear.

```
p()
{
    while (1)
    {
        x = producir();
        if (cont == N)
            sleep();
        cont++;
        ingresar(x);
        if (cont == 1)
            wakeup(c);
    }
}
```

```
c()
{
    while (1)
    {
        if (cont == 0)
            sleep();
        x = sacar();
        cont--;
        if (cont == N - 1)
            wakeup(p);
        consumir(x);
    }
}
```

- Usa una variable cont que indica la cantidad de lugares ocupados que tiene el buffer.
- Presenta al menos una condición de concurso. El algoritmo falla.
- La condición de concurso está dada cuando el buffer está vacío y el consumidor acaba de leer cont para verificar si es 0. Ahí, lo interrumpen y pasa al productor; éste ingresa un elemento en el buffer por lo que razonando que cont era 0, despierta al consumidor, pero esa señal se pierde por lo que el consumidor probará el viejo valor de cont y se bloqueará.

B. Con contadores de eventos (Reed y Kanodia 1979)

- Un contador de eventos **e** es una variable especial que tiene 3 operaciones definidas

en las siguientes primitivas:

1. read(e): Devuelve el valor de e.
2. advance(e): Incrementa atómicamente el valor de e en 1.
3. await(e, v): Espera hasta que e ≥ v.

- Solo incrementan el valor, nunca lo disminuye.
- Siempre se inician en cero.

Ejemplo para productor y consumidor:

```
p()
{
    int prod;
    while (1)
    {
        x = producir();
        prod++;
        await(N, prod - sacados);
        ingresar(x);
        advance(ing);
    }
}
```

```
c()
{
    int sec;
    while (1)
    {
        sec++;
        await(ing, sec);
        x = sacar();
        advance(sacados);
        consumir(x);
    }
}
```

ing: Cuenta el número acumulativo de elementos que el productor colocó en el buffer.
sacados: Cuenta el número acumulativo de elementos que el consumidor retiró del buffer.

C. Con Semáforos

```
p()
{
    while (1)
        {
        x = producir();
        P(vacío);
        P(mutex);
        ingresar(x);
        V(mutex);
        V(lleno);
        }
}
```

```
c()
{
    while(1)
        {
        P(lleno);
        P(mutex);
        x = sacar()
        V(mutex);
        V(vacío);
        consumir(x);
        }
}
```

- Usa tres semáforos:
  semáforo mutex = 1, vacío = N, lleno = 0;

Cualesquiera de estos algoritmos propuestos sincronizan el modelo de productor-consumidor que se presentan en múltiples situaciones dentro de un computador. Por ejemplo la CPU produciendo mensajes para un módulo de E/S o viceversa.

D. Hilos (threads) en el Modelo Productor - Consumidor

Existen aplicaciones que se prestan para ser programadas con el modelo de hilos, como es el caso de el problema de los productores-consumidores. Nótese que como comparten un buffer en común no funcionaría el hecho de tenerlos en procesos ajenos. Por último nótese la utilidad de los hilos en sistemas multiprocesadores en dónde pueden realmente ejecutarse en forma paralela.

Consideremos el problema productor-consumidor como el indicado en la Fig. 4.13 donde un proceso produce ítems que son consumidos por un proceso consumidor. Para esto se usa un buffer que puede contener varios ítems; el productor va depositando los ítems a medida que los produce, y el consumidor los va sacando del buffer a medida que los va consumiendo. Por cierto, hay que preocuparse de que el productor no siga poniendo ítems si el buffer está lleno, y que el consumidor no intente sacar un ítem si el buffer está vacío, o sea, los procesos deben sincronizarse para ello vimos algunos algoritmos.

Las **ventajas** de resolver esta clase de problemas usando dos procesos, es que tenemos:

- Modularidad, ya que cada proceso realiza una función bien específica.
- Disminución en el tiempo de ejecución. Mientras un proceso espera por I/O, el otro puede realizar trabajo útil. O bien, si se cuenta con más de un procesador, los procesos pueden trabajar en paralelo.

Pero también hay algunas **desventajas**:

- Los procesos deben sincronizarse entre sí, y deben compartir el buffer. Por las condiciones de Bernstein uno de esos procesos si escribe el otro no puede leer. También podría ser necesario (o conveniente) que compartan otra clase de recursos (como archivos abiertos). ¿Cómo puede hacerse esto si los procesos tienen espacios de direccionamiento disjuntos? (Recordemos que el Sistema Operativo no permite que un proceso escriba en la memoria que le corresponde a otro proceso).
- Los cambios de contexto son caros. En la medida que haya más procesos para resolver un problema, más costos habrá debido a los cambios de contexto. Además, al cambiar de un proceso a otro, como los espacios de direccionamiento son disjuntos, hay otros costos indirectos (que tienen relación con cachés y TLBs, que veremos más adelante en el próximo modulo).

Una interesante alternativa es usar threads (hilos o procesos livianos). Como habíamos visto en el módulo 2, un thread es un hilo de control dentro de un proceso. Un proceso tradicional tiene sólo un thread de control. Si usamos threads, entonces podemos tener varios hilos dentro de un proceso. Cada thread representa una actividad o unidad de computación dentro del proceso, es decir, tiene su propio Program Counter, conjunto de registros y stack, pero comparte con las demás hilos el espacio de direccionamiento y los recursos asignados, como archivos abiertos y otros.

En ese sentido, no hay protección entre threads: nada impide que un thread pueda escribir, por ejemplo, sobre el stack de otro. Bueno, quien debiera impedirlo es quien programa los threads, que es una misma persona o equipo (por eso no se necesita protección).

Comparando hilos con procesos:

- Cambio de contexto entre hilos de un mismo proceso es mucho más barato que cambio de contexto entre procesos; gracias a que comparten el espacio de direccionamiento, un cambio de contexto entre threads no incluye los registros y tablas asociados a la administración de memoria.
- La creación de threads también es mucho más barata, ya que la información que se requiere para mantener un thread es mucho menos que un PCB. La mayor parte de la información del PCB se comparte entre todos los threads del proceso.
- El espacio de direccionamiento compartido facilita la comunicación entre los hilos y el compartimiento de recursos.

Como ejemplo proponemos el siguiente programa escrito en Pascal para resolver el problema de productores y consumidores con buffer limitado:

```
var buffer: array [0..n-1] of item;
var in, out: 0..n-1   (* inicializado a 0 *)

(* productor *)
producir nextp;
while in+1 mod n = out do no-op;
buffer[in] := nextp;
in := (in+1) mod n;

(* consumidor *)
while in = out do no-op;
nextc := buffer[out];
out := (out+1) mod n;
consumir nextc;
```

Usa solamente n-1 posiciones del buffer.

4.6.6. Problema de los lectores escritores:

Este problema existe un área de datos compartida entre una serie de procesos. Hay una serie de procesos que sólo leen los datos, lectores, y otros que sólo escriben los datos, escritores. Se deben satisfacer tres condiciones:

1. Cualquier número de lectores pueden leer los datos simultáneamente.
2. Sólo puede escribir un escritor en cada instante.
3. Si un escritor está accediendo a los datos, ningún lector puede leer.

Cabe aclarar que los escritores no leen y los lectores no escriben.

- **Prioridad a los lectores:** se utiliza un semáforo para respetar la mutua exclusión (mutex). Mientras que un escritor está accediendo a los datos compartidos, ningún otro escritor y ningún otro lector podrá hacerlo (ese es el proceso escritor). Para el lector también se utiliza el semáforo mutex para la mutua exclusión. Para que se permitan varios lectores, hace falta que cuando no haya ninguno, el primer lector que lo intente tenga que esperar en mutex. Cuando ya hay al menos un lector, los siguientes no necesitan esperar antes de entrar. La variable global cantidad_lectores se utiliza para mantener el número de lectores y un semáforo x es para asegurar que la variable global se actualice correctamente.
- **Prioridad a los escritores:** cuando los lectores tenían prioridad estos mantenían el control del área de datos mientras que había al menos uno leyendo. Por lo tanto los escritores estaban sujetos a inanición. Lo que se quiere lograr ahora es no permitir acceder a los datos a ningún lector nuevo si es que al menos un escritor quiere escribir. Para lograr esto en el proceso escritor se agregan un semáforo mutex que inhibe todas las lecturas mientras haya al menos un escritor que desee acceder a los recursos, una variable cantidad_escritores que controla la activación de mutex y un semáforo y que controla la actualización de cantidad_escritores. Para los lectores, se necesita un semáforo más. No hay que permitir que se construya una cola grande sobre mutex, pues los escritores nunca accederían a la sección crítica. Por lo tanto sólo se permite a un lector ponerse en la cola en mutex y todos los demás lectores deben ponerse en la cola en un semáforo z inmediatamente antes de esperar en mutex.

4.6.7. Direccionamiento en el modelo "Cliente – Servidor"

El **"modelo cliente - servidor"** es semejante al modelo "Productor- consumidor" y se basa en un "protocolo solicitud / respuesta":

- Es sencillo y sin conexión.
- No es complejo y orientado a la conexión como OSI o TCP / IP.
- El cliente envía un mensaje de solicitud al servidor pidiendo cierto servicio.
- El servidor:
  - Ejecuta el requerimiento.
  - Regresa los datos solicitados o un código de error si no pudo ejecutarlo correctamente.
- No se tiene que establecer una conexión sino hasta que ésta se utilice.

Veamos como se direccionan:

Para que un cliente pueda enviar un mensaje a un servidor, debe conocer la dirección de éste. Un esquema de direccionamiento se basa en la dirección de la máquina destinataria del mensaje:

- Es limitativo si en la máquina destinataria se ejecutan varios procesos, pues no se sabría para cuál de ellos es el mensaje.

Otro esquema de direccionamiento se basa en identificar los procesos destinatarios en vez de a las máquinas:

- Elimina la ambigüedad acerca de quién es el receptor.
- Presenta el problema de cómo identificar los procesos:
  - Una solución es una nomenclatura que incluya la identificación de la máquina y del proceso:
    - No se necesitan coordenadas globales.
    - Pueden repetirse los nombres de los procesos en distintas máquinas.

Una variante puede ser utilizar **machine.local-id** en vez de **machine.process**:

- local-id generalmente es un entero aleatorio de 16 o 32 bits.
- Un proceso servidor se inicia mediante una llamada al sistema para indicarle al núcleo que desea escuchar a local-id.
- Cuando se envía un mensaje dirigido a machine.local-id el Kernel sabe a cuál proceso debe dar el mensaje.

El direccionamiento machine.process presenta el serio inconveniente de que no es transparente:

- La transparencia es uno de los principales objetivos de la construcción de sistemas distribuidos.
- En este caso el usuario debe conocer la posición del servidor.
- Un cambio de servidor obliga a cambiar los programas recompilarlos.

Otro método de direccionamiento consiste en asignarle a cada proceso una única dirección que no contenga un número de máquina:

- Una forma es mediante un asignador centralizado de direcciones a los procesos que mantenga un contador:
    - Al recibir una solicitud de dirección regresa el valor actual del contador y lo incrementa en uno.
- La desventaja es el elemento centralizado que puede ser un cuello de botella.

También existe el método de dejar que cada proceso elija su propio identificador:

- En un espacio de direcciones grande y disperso, por ej.: enteros binarios de 64 bits.
- La probabilidad de que dos procesos elijan el mismo número es muy pequeña.
- Existe el problema, para el Kernel emisor, de saber a qué máquina enviar el mensaje:
    - En una LAN[3], el emisor puede transmitir un paquete especial de localización con la dirección del proceso destino.
    - Este paquete de transmisión será recibido por todas las máquinas de la red.
    - Todos los núcleos verifican si la dirección es la suya; si lo es, regresa un mensaje aquí estoy con su dirección en la red (número de máquina).
    - El núcleo emisor utiliza esa dirección y la captura para evitar a posteriori una nueva búsqueda del servidor.
- Es un esquema transparente, pero la transmisión provoca una carga adicional en el sistema:
    - Se puede evitar con una máquina adicional para la asociación de:
        - Los nombres de servicios.
        - Las direcciones de las máquinas.

Al utilizar este sistema:

- Se hace referencia a los procesos de los servidores mediante cadenas en ASCII que son las que aparecen en los programas.
- No se referencian números binarios de máquinas o procesos.
- Al ejecutar un cliente que intente utilizar un servidor:
    - En su primer intento envía una solicitud a un servidor especial de asociaciones (servidor de nombres):
        - Le solicita el número de la máquina donde en ese momento se localiza el servidor.
    - Conociendo la dirección del servidor, se le envía la solicitud del servicio requerido.

Otro método utiliza hardware especial:

- Los procesos eligen su dirección en forma aleatoria.
- Los chips de interfaz de la red se diseñan de modo que permitan a los procesos almacenar direcciones de procesos en ellos.
- Los paquetes transmitidos utilizan direcciones de procesos en vez de direcciones de máquinas.
    - Al recibir cada paquete los circuitos de interfase de la red debe examinarlo para determinar si el proceso destino se encuentra en esa máquina:
        - Lo acepta en caso afirmativo.
        - No lo acepta en caso negativo.

4.6.7. Primitivas de transferencia de mensajes

a) Primitivas de Bloqueo Vs. No Bloqueo en Cliente– Servidor (comunicación sincrónica vs. asincrónica)

Las primitivas de transferencia de mensajes consideradas anteriormente se denominan primitivas de bloqueo o primitivas síncronas:

- El proceso emisor se suspende (se bloquea) mientras se envía el mensaje.
- El proceso receptor se suspende mientras se recibe el mensaje.

Una alternativa son las primitivas sin bloqueo o primitivas asíncronas:

- El proceso emisor:
    - No se suspende mientras se envía el mensaje.
    - Sí puede continuar su cómputo paralelamente con la transmisión del mensaje.
    - No puede modificar el buffer de mensajes hasta que se envíe el mensaje.
    - No tiene control sobre la terminación de la transmisión y por lo tanto no sabe cuándo será seguro reutilizar el buffer.

Una solución es:

[3] LAN (**L**ocal **A**rea **N**etwork – red de área local)

- Que el Kernel copie el mensaje a un buffer interno del Kernel.
- Que entonces el Kernel permita al proceso continuar y reutilizar el buffer.

La desventaja de la solución es que cada mensaje de salida debe ser copiado desde el espacio del usuario al espacio del Kernel.
Otra solución es:

- Interrumpir al emisor cuando se envíe el mensaje.
- Informarle que el buffer nuevamente está disponible.

La desventaja radica en la dificultad de la programación basada en interrupciones a nivel usuario.
Generalmente se considera que las desventajas de las primitivas asíncronas no compensan las ventajas del máximo paralelismo que permiten lograr.
El criterio utilizado ha sido el siguiente:

- La diferencia esencial entre una primitiva síncrona y una asíncrona es si el emisor puede volver a utilizar el buffer de mensajes en forma inmediata y segura después de recuperar el control.
- El momento en que el mensaje llega al receptor es irrelevante.

Otro criterio establece lo siguiente:

- Una primitiva síncrona es aquella en que el emisor se bloquea hasta que el receptor ha aceptado el mensaje y la confirmación regresa al emisor.
- Todo lo demás es asíncrono con este criterio.

Desde el punto de vista del S. O. generalmente se considera el primer criterio; el interés está centrado en el manejo de los buffers y en la transmisión de los mensajes.
Desde el punto de vista de los lenguajes de programación se tiende a considerar el segundo criterio; el interés está centrado en el lenguaje de programación y sus facilidades de uso.
Generalmente a las primitivas de envío se las conoce como **send** y a las de recepción como **receive** y ambas pueden ser con bloqueo o sin bloqueo.
Una recepción sin bloqueo le indica al Kernel la localización del buffer y regresa el control:

- El problema es saber quién hizo la llamada cuando se llevó a cabo la operación.
- Las soluciones pueden ser:
    - Proporcionar una primitiva explícita **wait** que permita al receptor bloquearse cuando lo desee.
    - Proporcionar una primitiva **test** que permita verificar el estado del Kernel.

b) Primitivas Almacenadas en Buffer Vs. No Almacenadas en Cliente – Servidor (comunicación directa vs. indirecta)

Las primitivas consideradas hasta ahora son esencialmente primitivas no almacenadas:

- Significa que una dirección se refiere a un proceso específico.
- Una llamada receive (addr, &m) le indica al Kernel de la máquina en donde se ejecuta:
    - Que el proceso que hace la llamada escucha a la dirección addr.
    - Que está preparada para recibir el mensaje enviado a esa dirección.
    - Que se dispone de un único buffer de mensajes al que apunta m para capturar el mensaje que llegará.
    - Que cuando el mensaje llegue será copiado (por el Kernel receptor) al buffer:
        - Se elimina entonces el bloqueo del proceso receptor.

Este esquema funciona bien cuando el servidor llama a receive antes de que el cliente llame a send.
El problema se presenta cuando el send se lleva a cabo antes que el receive:

- El Kernel del servidor:
    - No sabe cuál de sus procesos utiliza la dirección en el mensaje recién llegado.
    - No sabe dónde copiar el mensaje recibido.

Una solución consiste en:

- Descartar el mensaje.
- Dejar que el cliente espere.
- Confiar en que el servidor llame a receive antes de que el cliente vuelva a transmitir; por ello el cliente podría tener que intentar varias veces.

Si dos o más clientes utilizan un servidor con transferencia de mensajes sin almacenamiento en buffers:

- Luego de que el servidor aceptó un mensaje de uno de ellos:
    - Deja de escuchar a su dirección hasta que termina su trabajo.
    - Regresa al principio del ciclo para volver a llamar a receive.
- Si realizar el trabajo insume cierto tiempo, los demás clientes podrían hacer varios intentos de envíos sin éxito.

Otra solución consiste en hacer que el Kernel receptor mantenga pendientes los mensajes por un instante:

- Para prevenir que un receive adecuado se realice en un tiempo corto.
- Cuando llega un mensaje "no deseado", se inicializa el cronómetro:
  - Si el tiempo expira antes de que ocurra un receive apropiado, el mensaje se descarta.
- Se reduce la probabilidad de que un mensaje se pierda.
- Se debe almacenar y manejar los mensajes que llegan en forma prematura.
- Se necesitan los buffers y la administración de los mismos.
- Se puede hacer mediante una estructura de datos llamada **buzón:**
  - Un proceso interesado en recibir mensajes:
    - Le indica al Kernel que debe crear un buzón para él.
    - Especifica una dirección en la cual buscar los paquetes de la red.
  - Todos los mensajes que lleguen en esa dirección se colocan en el buzón.

La llamada a receive elimina un mensaje del buzón o se bloquea (si se utilizan primitivas con bloqueo) si no hay un mensaje presente.
Esta técnica se denomina primitiva con almacenamiento en buffers.
Los buzones tienen el problema de que son finitos y pueden ocuparse en su totalidad:

- Cuando llega un mensaje a un buzón lleno, el Kernel debe elegir entre:
  - Mantener el mensaje pendiente por un momento esperando que algún mensaje sea retirado del buzón a tiempo.
  - Descartar el mensaje.
- Esta es la misma situación que se tiene cuando se trabaja sin almacenamiento en buffers:
  - Con buffers se reduce la probabilidad de problemas, pero los problemas no se eliminan ni cambia su naturaleza.

Otra solución utilizada es no dejar que un proceso envíe un mensaje si no existe espacio para su almacenamiento en el destino:

- El emisor debe bloquearse hasta que obtenga de regreso un reconocimiento:
  - Debe indicar que el mensaje ha sido recibido.
- Si el buzón está lleno, el emisor puede hacer un respaldo y suspenderse de manera retroactiva:
  - La suspensión debe operar como si fuera justo antes de que intentara enviar el mensaje.
  - Cuando haya espacio libre en el buzón, se hará que el emisor intente nuevamente.

4.7. Deadlocks (interbloqueo, bloqueo mutuo o abrazo mortal)

Se define **Deadlock o Abrazo Mortal** como el estado en el que dos o más procesos esperan por condiciones que no se dan y que deben producirse por los procesos de ese conjunto. Se dice que dos o más procesos se encuentran en estado de deadlock cuando están esperando por condiciones que nunca se van a cumplir.

Los términos Dadlock, abrazo mortal, bloqueo mutuo, interbloqueo , estancamiento son sinónimos.

Entonces se podría hablar de Deadlock como el **estado permanente de bloqueo** de un conjunto de procesos que están compitiendo por recursos del sistema o se comunican entre ellos bajo dos situaciones:

- Ante la petición de recursos: si un proceso solicita un recurso que no está disponible éste queda esperando. Si todos los procesos quedan esperando por un recurso que tiene asignado otro proceso del conjunto y que también esta a la espera de otro recurso, se produce Deadlock.
- Ante la comunicación entre procesos: cuando cada proceso de un conjunto espera por un mensaje de otro miembro del grupo, y no existe un mensaje en tránsito, entonces ocurre un Deadlock.

Una gran mayoría de las soluciones propuestas para este tipo de conflicto serían de aplicabilidad para aquellos originados por la comunicación entre procesos.

Un ejemplo puede ser, si consideramos un sistema con una disquetera y una impresora y suponemos que el proceso P1 está usando la disquetera y el proceso P2 tiene asignado la impresora, si P1 solicita la impresora y P2 pide la disquetera, habrá deadlock. ejemplo: solución al problema de los filósofos comensales presentados por Disjktra.

Otro ejemplo: los semáforos Q y S controlan acceso a recursos compartidos, y:

```
Proceso 0:          Proceso 1:
  P(Q);               P(S);
  P(S);               P(Q);
  usar recursos       usar recursos
  V(S);               V(Q);
  V(Q);               V(S);
```

Hay deadlock cuando una máquina se "cuelga" en la jerga informática.

Observemos que el deadlock se debe fundamentalmente por el uso de recursos. Se pueden distinguir dos categorías generales de recursos: reutilizables y consumibles.

Recursos reutilizables

Un recurso reutilizable es aquel que puede ser usado por un proceso y no se agota con el uso. Los procesos obtienen unidades de recursos que liberan posteriormente para que otros procesos las reutilicen. Como ejemplos se tienen los procesadores, canales de E/S, memoria central y secundaria, dispositivos y estructuras de datos tales como archivos, bases de datos y semáforos.

Recursos consumibles

Un recurso consumible es aquel que puede ser creado (producido) y destruido (consumido).

Normalmente, no hay límite en el número de recursos consumibles de un tipo en particular. Un proceso productor que no esta bloqueado puede liberar cualquier numero de recursos consumibles. Cuando un proceso adquiere un recurso este deja de existir. Como ejemplos están las interrupciones, señales, mensajes, o la información en un buffer de E/S.

No hay ninguna estrategia sencilla que pueda solucionar todas las clases de deadlock. Los enfoques más importantes que se han tomado son: detección, prevención y predicción. Previamente consideremos las condiciones necesarias y suficientes para que se dé un deadlock.

4.7.1 Condiciones necesarias y suficientes:

La mayoría de los autores, Silberschatz, Jensen y Tanenbaum, coinciden en que para que se produzca un estado de deadlock las cuatro **condiciones de Coffman** deben producirse simultáneamente. Estas condiciones son las siguientes:

1. **Mutua exclusión:** los recursos no deben ser compartibles, es decir que sólo un proceso a la vez puede usar el recurso.
2. **Retener y esperar** (Hold & Wait)**:** significa que el proceso retiene los recursos que ya tiene asignados mientras espera por nuevos a adquirir del conjunto de recursos del sistema.
3. **No expropiación** (No Preemption)**:** el proceso está reteniendo los recursos concedido y solo puede liberarlos y devolverlos al sistema como resultado de una acción voluntaria de ese proceso. El S.O. no puede obligarle a devolverlos esto se conoce como no expropiación.
4. **Espera circular:** debe existir un conjunto de procesos $\{P_0, P_1, ..., P_n\}$ tal que, P_0 está esperando un recurso que posee P_1, P_1 está esperando por un recurso que posee P_2, ..., y P_n está esperando por un recurso que posee P_0, o sea, los procesos están esperando mutuamente a que el otro libere el recurso requerido formando una cadena circular entre dos o más procesos, en la que cada uno de ellos está esperando un recurso que tiene el próximo miembro de la cadena.

En rigor, espera circular implica retención y espera, pero es útil separar las condiciones, pues basta con que una no se dé para que no haya deadlock.

Si bien la mayoría de los autores toman a estas cuatro condiciones como necesarias y suficientes para que ocurra una situación de Abrazo Mortal, en cambio W. Stallings destaca que las primeras tres condiciones mencionadas, son necesarias pero no suficientes, y la cuarta condición ocurre como consecuencia de las tres primeras. Entonces la diferencia que marca Stallings radica en que la cuarta condición es una consecuencia de las tres primeras. Esto es, si se cumplen las tres primeras condiciones, una secuencia de eventos llevará a que se incurra en una espera circular.

4.7.2. Grafo de asignación de recursos

Otra manera de definir deadlocks es a través de grafos ya que sirve para describir si existe un interbloqueo.

Estos grafos están formados básicamente por dos elementos:

- Un conjunto de vértices formado por los procesos y los recursos del sistema;
- Un conjunto de arcos que representan la asignación o solicitud de recursos.

Para ilustrar la situación (ver figura 4.14) se utiliza un grafo dirigido llamado **Resource Allocation Graph** (Grafo de Asignación de Recursos). En él se representan:

1. Conjunto de vértices V con los procesos (Px) y los recursos (Ry) del sistema

   $P = \{P_1, P_2, ..., P_n\}$ Procesos

   $R = \{R_1, R_2, ..., R_n\}$ Recursos

2. Conjunto de arcos A que unen los procesos con los recursos.

En la Fig. 4.14 podemos ver tres grafos de asignación de recursos. Los recursos fueron representados con cuadrados y los procesos con círculos. Un arco de P_i a R_j significa que el proceso P_i solicito el recurso R_j. Un arco de R_j a P_i significa que el recurso R_j esta asignado al proceso P_i.

Existe **deadlock** cuando en el grafo de asignación se presentan uno o más ciclos y los recursos son críticos como se ve en la Figura 4.14c en que los procesos P_1 y P_2 están bloqueados (**deadlocked**).

- Si cada recurso en el sistema tiene más de una instancia entonces el ciclo es condición necesaria, pero no suficiente.

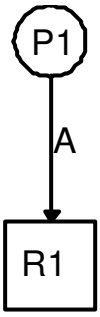


Fig. 4.14a

Fig. 4.14b

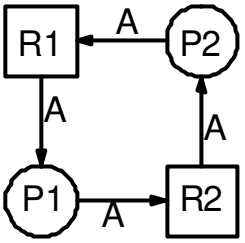


Fig. 4.14c

Un arco dirigido de P1 a R1 indica que P1 pidió el recurso R1 y está esperando. (Fig 4.14a)

Un arco dirigido de R1 a P1 indica que P1 pidió el recurso R1 y R1 está asignado a P1. (Fig 4.14b)

Un conjunto de arcos como se indica en la Fig 4.14c indica que está en deadlock

Fig. 4.14 Representaciones de grafos para solicitud y asignación de recursos

En el primer grafo de la Figura 4.14a se ve que el proceso P1 solicitó el recurso R1, y está esperando a que le sea asignado.
En el segundo grafo se ve que el proceso P1 tiene asignado el recurso R1.
Finalmente en el último grafo vemos una clara situación de deadlock donde cada proceso esta esperando por un recurso que tiene otro proceso que también se encuentra en la misma situación y cumpliéndose las condiciones de Coffman.

4.7.3. Estrategias para tratar deadlocks

En la mayoría de los autores se encontrarán similitudes en el tratamiento de deadlocks, la diferencia se basa en la aplicación de los algoritmos en los distintos métodos, cuyas características se remarcará a lo largo de la explicación de cada uno de ellos.

Según Silbershatz, así como los demás autores, asegura que existen tres métodos para tratar el problema de los deadlocks:

1. Ignorarlos y pensar que en el sistema nunca ocurrirán. La mayoría de los sistemas operativos utilizan esta técnica, incluyendo UNIX.
2. Por medio de la prevención o evasión se asegura que el sistema nunca entrará en estado de deadlock.
3. Permitir que ocurra un estado de deadlock y luego recuperarlo.

1) Ignorarlo

La forma más simple para tratarlo es ignorándolo, sin olvidarnos que estamos permanentemente en riesgo, ya que, una situación de Abrazo Mortal implicaría volver a arrancar el sistema, y las pérdidas en que se incurren podrían ser muy graves. Pero hay veces en las que el costo de ignorarlo es mucho menor al costo que implicaría prevenirlo o detectarlo y recuperarlo. Normalmente esta estrategia se utiliza cuando la frecuencia en que ocurren este tipo de situaciones es relativamente baja. UNIX, es uno de los Sistemas Operativos que adopta esta estrategia. Tanenbaum propone como "**algoritmo del avestruz**" y dice "Si los deadlocks no son frecuentes entonces puede ser más conveniente esconder la cabeza bajo la arena".

Es el enfoque seguido no solo por UNIX sino por muchos otros S.O., ya que los métodos que solucionan el problema del deadlock en las aplicaciones suelen ser bastante costosos e implican volver a arrancar el sistema; generalmente no requieren de los complejos mecanismos para "asegurar" la no ocurrencia. Pero hay aplicaciones que necesariamente requieren asegurar la no ocurrencia, como ser el caso del procesamiento de Sistemas de Control (Aeropuertos, Centrales Nucleares, o en sistemas de bases de datos, etc.) en que es necesario detectarlo y corregirlo a priori. Por otro lado, algunas tendencias hacen que el tema vaya adquiriendo importancia, a medida que progresa la tecnología, tiende a haber más recursos y más procesos en un sistema, entonces se requiere de un mecanismo para detectar o controlar los Deadlocks.

2) Prevención:

Una posibilidad es mantener el grafo de recursos, actualizándolo cada vez que se solicita o libera un recurso, o bien, cada cierto tiempo construir el grafo. (¿Cada cuánto?) Si se detecta un ciclo, se matan todos los procesos del ciclo, o se van matando procesos del ciclo hasta que no queden ciclos (¿cuál matar primero? ¿Qué pasa si un proceso está en medio de una transacción?). Es un método drástico, pero sirve.

Otra estrategia que resuelve el problema es limitando el uso de los recursos e imponiendo restricciones a los procesos. Siendo las cuatro condiciones necesarias para que ocurra un Deadlock basta con asegurarnos que **una de ellas no ocurrirá**. Las restricciones y limitaciones se imponen con el fin de prevenir que una de las condiciones no ocurra. Analizaremos ahora como atacar una de las 4 condiciones (que no ocurra al menos una) y de esa forma prevenir cada una de ellas:

- **Mutua Exclusión**: Los recursos que son compartidos, no causan problemas si hay existencias, ya que los mismos pueden ser utilizados por cualquier proceso en cualquier momento. Las soluciones para aquellos recursos que no pueden ser compartidos son diversas, pero todas se basan en que un proceso no quede esperando en caso de la falta de disponibilidad de dicho recurso. Por ejemplo, la impresora es un recurso no compartido, se podría crear una cola de impresión donde los procesos pondrían sus pedidos y seguirían su ejecución. En cuanto la impresora esté lista, imprimirá el pedido. Esta solución lo provee el SPOOLER simulando la impresora en el disco.
- **Control y espera** (Hold & Wait): Para solucionar este problema, se trata de garantizar que cuando un proceso tenga un recurso asignado no pueda solicitar otro. Hay dos caminos para lograrlo:
  1- Los procesos solicitan todos los recursos en el momento previo a comenzar la ejecución, de no poder ser entregados el proceso queda bloqueado sin los recursos y luego los pedirá nuevamente. Si hay, ejecutará sino se bloqueará.

  Asignación estática completa de todos los recursos que necesita para su ejecución (caso COBOL).

  Uno de los problemas que surgen es la ineficiencia. Los procesos quedan esperando sin poder realizar tarea alguna hasta no poder obtener la totalidad de los recursos solicitados. Más grave aún sería la posibilidad de que la espera se convirtiese en una espera infinita debido a la popularidad de algún recurso solicitado, derivando así en **estado de inanición**. Desde el punto de vista de los recursos, el resultado tampoco sería óptimo, en realidad su utilización es relativamente bajo, ya que estos serían asignados a procesos que quizás los mantendrían sin uso por un largo período. El tiempo en el cual éste podría haber sido asignado y utilizado por algún otro proceso. Otro inconveniente es que no todos los procesos conocen previamente que recursos le serán necesarios durante la ejecución, sino hasta el momento en que la misma se lleve a cabo.

  2- Un proceso primero debe liberar aquellos recursos que posee y luego recién podrá solicitar otros, es decir solo está en condiciones de solicitar un recurso cuando no tiene ninguno asignado.
  El mayor inconveniente aquí es que hay casos en que los procesos necesitan al menos, dos recursos a la vez para su ejecución. Un ejemplo sería aquél en el que un proceso necesita copiar un archivo, de un disco a una cinta; la carencia de uno de ellos le impediría lograr su objetivo.

- **No expropiación** (No Preemption): Para ésta condición también existen dos métodos:
  1. Si un proceso solicita un recurso que no está disponible, éste debe devolver todos aquellos recursos que tenía previamente asignados. De ser necesario tendrá que pedir todos los recursos nuevamente, o sea que, si una tarea que ya posee y solicita nuevos recurso que el S.O. niega, entonces debe devolver todos los que posee y volverlos a pedir más tarde. Es efectiva con el multiplexado rápido de los recursos (Ejemplo CPU, Memoria Central, etc.) e ineficiente para los lentos, como ser cinta, impresoras etc.
  2. Si un proceso pide un recurso que tiene otro proceso, el Sistema Operativo puede obligar a liberar los recursos al otro proceso. De esta forma podríamos decir que la utilización del recurso no sería apropiativa sino expropiativa.

  El primer método es viable sólo en aquellos procesos cuyos estados pueden ser fácilmente grabados y restaurados. El segundo método presenta el inconveniente del estado de inanición (Starvation), en caso de que a un proceso siempre le quiten los recursos y nunca pueda finalizar su tarea.

- **Espera circular**: Consiste en imponer un orden lineal de ejecución que evite las esperas circulares (se ejecuta un algoritmo que analiza si los procesos concurrentes pueden terminar).
  Este problema puede ser resuelto de la siguiente manera:
  1. Estableciendo un orden lineal de los recursos:
     Si tenemos una lista de recursos R_1, R_2,.....R_n, un proceso que solicitó R_h, sólo puede pedir aquellos recursos R_k, con $k > h$. Esto evita que se forme un círculo ya que el que posee el último recurso, no podrá solicitar el primero. Por ejemplo:
     F(CD-ROM)=1 ; F(impresora)=2 ; F(plotter)=3 ; F(Cinta)=4 .

Podemos evitar la espera circular si imponemos un orden total a los recursos (o sea, asignamos a cada recurso R un número único F(R)), y obligamos a los procesos a que soliciten recursos en orden: un proceso no puede solicitar F(k) y después l(ele) si F(k)>F(l). Si un proceso tiene la impresora no puede solicitar el plotter o la cinta.
De esta manera se garantiza que no se generen ciclos en el grafo de recursos. Una mejora inmediata es exigir que ningún proceso solicite un recurso cuyo número es superior a los recursos que ya tiene. Pero tampoco es la panacea. En general, el número potencial de recursos es tan alto que es difícil dar con tal función F para ordenarlos.

2. Otra forma de resolverlo es lo propuesto en Hold & Wait: un proceso sólo está en condiciones de solicitar un recurso cuando no tiene ninguno asignado.

El problema del orden lineal es que también es en cierto grado ineficiente ya que estaría negando recursos a procesos, innecesariamente. Otro punto problemático sería el encontrar un orden lineal que satisfaga a todos los procesos. Si bien el establecer un orden restringe a ciertas aplicaciones, esto no interfiere con la programación interna del kernel del Sistema Operativo.

Con respecto a la segunda propuesta los problemas ya fueron analizados previamente.

Comentarios sobre la Prevención del deadlock

Tanto Silbersschatz como Tanenbaum tratan de prevenir los deadlocks evitando que alguna de las cuatro condiciones de Coffman no se cumpla. Mientras que Stallings clasifica la prevención en dos tipos:

- Método directo: previene la ocurrencia de una de las tres primeras condiciones.
- Método indirecto: este método consiste en tratar de prevenir una espera circular.

- Mutua exclusión

En general todos los autores coinciden en este punto, manifestando que esta condición es muy difícil que no se cumpla ya que hay recursos no compartibles, tales como una impresora, que no puede ser accedida simultáneamente por varios procesos.

- Retención en el tiempo

Para Stallings esta situación puede ser prevenida requiriendo a los procesos que notifiquen todos los recursos que van a demandar y bloquear a los procesos hasta que sus pedidos sean satisfechos. Pero la desventaja que proporciona esta técnica es que un proceso puede llegar a esperar demasiado tiempo para que se le otorgue algún recurso, cuando en realidad pudo haber ganado tiempo con alguno de ellos. Otra desventaja es que un recurso asignado a un proceso puede permanecer sin uso, y sin embargo no lo puede usar otro proceso que también lo necesita.

La idea de Silberschatz es que para que un recurso pueda ser usado requiere que cada proceso pida y le sea asignado todos los recursos que usará al comienzo de la ejecución. Otra alternativa permite a un proceso pedir recursos sólo cuando el proceso no tiene ninguno. Un proceso puede pedir algunos recursos y usarlos. Antes de que pida uno adicional debe liberar todos los recursos asignados.

Para Tanenbaum existen dos técnicas para impedir esta condición: la primera consiste en que todos los procesos pidan sus recursos por adelantado. Si éstos se encuentran disponibles se los asignarán al proceso que los solicitó inmediatamente, de lo contrario tendrá que esperar hasta que se encuentren disponibles. La segunda técnica consiste en que un proceso que solicita un recurso deba liberar, momentáneamente, los recursos que dispone, recuperándolos cuando se cumpla exitosamente la solicitud del pedido.

Para poder aprovechar éstas técnicas es fundamental contar con la información previa de todos los recursos que solicitarán todos los procesos, pero muchos de éstos recursos serán requeridos a lo largo de su ejecución, lo que trae como consecuencia no poder contar siempre con la información requerida, haciendo imposible la prevención de deadlocks.

- Recurso no sustituible

Stallings, Tanenbaum y Silberschatz aseguran que para que no ocurra ésta condición, debemos proceder de la siguiente manera: si un proceso que tiene asignado varios recursos pide otro recurso que no puede ser inmediatamente asignado, todos los recursos que se encuentran en su poder deben ser liberados y luego debe pedir, nuevamente, todos los recursos incluyendo el nuevo. Otra forma puede ser considerando la prioridad, es decir si un proceso pide un recurso que tiene asignado otro proceso, el S.O. puede sacárselo y otorgárselo al primero por tener mayor prioridad.

- Espera circular

Para evitar esta condición se impone un orden de todos los recursos. Se determina un conjunto de recursos $\{R_1,R_2,...,Rn\}$ con su respectiva jerarquía de orden $\{Y_1,Y_2,..,Yn\}$, por ejemplo una disquetera (Ri) tiene un número de orden 1 (Yi), una impresora (Rj) tiene 5 (Yj) y así sucesivamente con los demás. Si

un proceso pide inicialmente el recurso Ri (disquetera) y luego pide el recurso Rj (impresora), sólo lo podrá hacer si y sólo si (Yj) > (Yi), es decir debe obtener primero la disquetera y luego la impresora. Alternativamente cuando un proceso pide un recurso de tipo Rj debe liberar cualquier recurso Ri tal que Yi > Yj. Si se cumple con esta regla no habrá lugar para que ocurra una espera circular, pero el problema se encuentra en que es muy difícil establecer dicho ordenamiento.

La política según Jensen es similar a la de los demás, consiste en que se otorgarán, en forma creciente o decreciente, los recursos de acuerdo a la cantidad de recursos requeridos. Por ejemplo: si se tiene un paquete de cigarrillos y un encendedor, como los cigarrillos son más que el encendedor entonces, se deberán obtener primero los cigarrillos y luego el encendedor. Si P_0 (un proceso) obtuvo primero los cigarrillos, entonces P_1 (otro proceso) tendrá que esperar que P_0 concluya exitosamente, para ello debiendo obtener antes el encendedor, y luego de utilizarlos quedarán disponibles para P_1.

3) Detectar y recuperar

Consiste en abortar un proceso cuando se detecta o se presupone que puede ocurrir el deadlock.
La ventaja de esta táctica frente a las anteriores es que no limita el acceso a los recursos, ni al accionar de los procesos. Pero presenta ciertos inconvenientes como el de decidir la frecuencia con que se llevará a cabo el algoritmo de detección y la aplicación del sistema de recupero.

El algoritmo podría ser ejecutado cada vez que se solicita un recurso, a cada hora, etc. Esto dependerá de la frecuencia en que ocurren los Deadlocks. De todo modo son poco frecuentes.

Las ventajas que surgen de ejecutar el algoritmo mencionado cada vez que se solicita un recurso son: la identificación inmediata de la situación de Abrazo Mortal, la simplicidad del algoritmo de detección ya que está basado en cambios incrementales del sistema, y el poder detectar cual fue el proceso que determinó tal situación.

La desventaja es que se pierde mucho tiempo, es puro overhead. Si se deja pasar un determinado período entre cada detección, puede que hayan surgido tantos ciclos en el grafo, que no se pueda determinar quien fue el causante.

Hay varios métodos para ***recuperar*** a los procesos y a los recursos una vez detectada la situación:

- Abortar todos los procesos involucrados
- Hacer un backup de cada proceso en un punto anterior: **ChekPoint**. En este se graban el estado de los recursos, los recursos asignados y, el estado del proceso entre otros, para poder reiniciarlo. A este proceso de reinicio se lo llama **Rollback**. Consiste en llevar el proceso a un punto anterior al de haberle sido asignado el recurso causante del Deadlock.
- Abortar los procesos uno a uno, hasta que el deadlock desaparezca.
- Ir abortando sucesivamente procesos hasta que no haya más deadlock. Luego de abortar un proceso de debe aplicar nuevamente el algoritmo de detección.
- Quitar un recurso a un proceso y entregárselo a otro que lo haya solicitado. También hay que ejecutar el algoritmo de detección luego de que se quitó algún recurso.

La primera opción es segura con respecto a la solución del Deadlock, pero un tanto riesgosa, ya que puede abortarse un proceso que estaba a punto de finalizar su tarea.

En cambio el inconveniente en la segunda opción es la probabilidad de que el deadlock vuelva a ocurrir, aunque algunos autores alegan que la aleatoriedad del procesamiento concurrente asegurará la no repetición de dicha situación.

Con respecto a la tercera opción hay una gran pérdida de tiempo, ya que cada vez que se aborta un proceso hay que correr el algoritmo de detección hasta que el deadlock desaparezca, y esto es puro overhead.

El problema que se presenta en las últimas dos opciones es el de seleccionar la víctima. Es decir cual será el proceso que será abortado o a cuál proceso se le quitará un recurso. Esta decisión normalmente se toma basándose en el criterio del "mínimo costo". El S.O. decide cuál es el proceso ***víctima*** que debe abortar según alguno de los siguientes criterios:

1. Menor cantidad de uso de CPU hasta el momento.
2. Menor cantidad de líneas de Salida (output) producidas hasta el momento.
3. Mayor tiempo de CPU restante.
4. Menor cantidad y tipo de recursos asignados hasta el momento. Los procesos que estén usando recursos críticos serán los que se eliminan y se abortan aquellos procesos que requieren gran cantidad de recursos.
5. Menor prioridad.
6. Aquel que su reiniciación no incurra en pérdidas significativas.
7. Si el proceso es Interactivo o Batch.

Entre estas variables hay algunas que son mas fáciles de medir que otras, y a veces resulta difícil comparar y tomar una decisión adecuada.
Para la detección y recuperación:

- El sistema monitorea las requisiciones y devoluciones de recursos entonces, actualiza el grafo de recursos y verifica si existe algún ciclo.
- Otra alternativa es no actualizar el grafo y ver periódicamente si un proceso lleva mucho tiempo bloqueado.

Comentarios sobre la Detección de deadlock

En este caso no sólo se debe suministrar un algoritmo que detecte si un estado de deadlock ha ocurrido sino otro algoritmo para la recuperación del mismo.

Tanenbaum, Silberschatz y Stallings coinciden en que el algoritmo de detección de deadlock debe ser invocado dependiendo de que tan frecuente sean los deadlock.

Cuando se detecta un deadlock no se debe continuar otorgando los pedidos de recursos para no incrementar el número de procesos involucrados.

Para Stallings las estrategias de detección no limitan accesos a recurso como en la prevención. Periódicamente el S.O. ejecuta un algoritmo de detección, dependiendo de lo que tarda, frecuentemente, en producirse un deadlock. Esto trae como consecuencia dos ventajas: se detecta el deadlock más rápido y el algoritmo es sencillo. Pero como desventaja se tiene que utiliza mucho tiempo el procesador. Una vez detectado el paso siguiente es recuperarlo cuyas estrategias son las siguientes:

Recuperación de un deadlock

Se puede recuperar de las siguientes formas:

- Manualmente
- Automáticamente: eliminar procesos en deadlock.
- Sustituir recursos

Todos son muy caros, pero son seguros. En el último caso, al sustituir recursos le estamos quitando un recurso a algún proceso y se lo damos a otro.

Se presentan Problemas:

- ¿ a cuál elegir?.
- ¿Qué hacemos con la víctima?. Debe recomenzar, ¿desde dónde?.

La elección depende del costo.

Una Técnica:

- Asignar un costo fijo (***Ci***) a la remoción de un recurso r_i de una tarea que es abortada.

El costo de liberar los recursos de una tarea t_i en deadlock será:

$\sum Ci \bullet g_{(x)}$ donde $g_{(x)} = \{ x \text{ si } x > 0 \text{ y } 0 \text{ si } x \leq 0 \}$ x es un valor empírico y define una función escalón.

Se han desarrollado algoritmos, que mediante eficientes búsquedas en árboles, encuentran soluciones de costo mínimos.

Jensen propone varios algoritmos de detección y recuperación que son los siguientes:

Backup : a primera vista es un algoritmo de evasión más que uno de detección ya que evita que ciertos procesos obtengan ciertos recursos. Sin embargo, es un algoritmo de detección por efecto final sobre los procesos. Estos son forzados a abandonar sus trabajos en forma progresiva. La técnica de este algoritmo se basa en el conocimiento de un número que tiene asignado cada recurso. Entonces la destrucción aparece cuando un proceso reclama a un proceso con número j mayor que k (el recurso con menor número que posee el proceso) y este no está disponible.

El sistema realiza un seguimiento de todos los procesos que piden recursos y mantiene un esquema interno de esta gráfica que le permite determinar cuándo se produce un ciclo, el sistema saca algún proceso del ciclo liberando el o los recursos que poseía. El proceso que se debería abortar es aquel que haya realizado menos trabajo, para hacer una buena elección.

Time out : se coloca un límite superior de tiempo para que un proceso termine su tarea o para que un recurso sea poseído por un proceso. En los dos casos si se excede el tiempo el sistema toma como si el proceso está en deadlock, entonces aborta el mismo devolviendo todos los recursos.

Este algoritmo trae como principal desventaja que se puede llegar a producir inanición (starvation), por ejemplo si un proceso demanda mucho tiempo para completar su ejecución siempre va a haber time out impidiendo que nunca finalice su trabajo.

Time stamping: usar time stamping en vez de time out minimiza el riesgo de abortar recursos innecesariamente. Cuando se expira el time out el proceso no se aborta, sino hasta que se vea un potencial bloqueo. Cada vez que un proceso empieza una tarea, se lo marca con un time stamp,

garantizando que nunca habrá dos time stamp iguales. En caso de bloqueo este número se utiliza para decidir si existe un potencial deadlock y si es necesario abortar el proceso.

Algoritmo de detección
A continuación proponemos un algoritmo para examinar el estado del sistema y determinar si hay o no deadlock.
Se deben analizar para dos tipos de recursos:
- Recursos de n puntos de entrada.
- Recursos de 1 punto de entrada.

a) Detección con n puntos de entrada.
La idea general es verificar los n procesos corriendo en el sistema y observar si la suma de recursos usados más recursos pedidos debe ser menor o igual a los recursos disponibles en cada momento Ti. Si no se cumple la condición ≥ Pi entonces se produce deadlock.
Se necesitan tres estructuras de datos: n procesos y m recursos
- D: vector de m posiciones indicando cuantos recursos hay en ese instante disponibles de cada tipo => D[j] = 3 => hay 3 j's disponibles (siento j un solo tipo de recursos. Por ejemplo impresoras).
- A: asignados. Matriz n x m
- P: pedidos. Matriz n x m

Notaciones:
* A <= B <==> **símbolo i: 1 <= i <= n: A[i] <= B[i]
* Si M es n x m => Mi es el vector Mi,1 Mi,2 ... Mi,n

Política:
Si un proceso puede finalizar => asumimos que no va a pedir más r recursos y actuamos como si ya hubiera terminado, reclamamos sus recursos ficticiamente y seguimos el análisis.
* Si esto no se cumple => el algoritmo lo detectará la siguiente vez que sea ejecutado.

Algoritmo
* Vector (D) temp[m], donde se simularán las devoluciones.
* Vector termino[n], si el proceso terminó o no.

```
temp = D
    Para todos los procesos que tengan recursos asignados termino[i] = 0.
While (existe Pi tal que termino[i] = 0 && Pi <= Dtemp)
    {
    Dtemp = Dtemp + Ai      /*simulamos que el proceso terminó y devuelve sus recursos */
    termino[i] = 1
    }
     {
     if (existe i tal que termino[i] = 0)
        return TRUE;    // el sistema está en deadlock (Pi)
     else
        return False;   // el sistema no está en deadlock
     }
```

Ejemplo: 3 procesos y 2 recursos R_1, R_2 => R_1 tiene 4 puntos de entrada
R_2 tiene 2 puntos de entrada
Esta configuración en T_0 no produce deadlock
Problema, el algoritmo es O(m x n2)

b) Detección con 1 punto de entrada

Modifica el grafo de asignación de recursos para crear el de espera => si Pi tiene un recurso que quiere Pj se dibuja un arco dirigido de Pi $\longrightarrow$ Pj.
* Si hay ciclos => deadlock.
* Aunque es O(n2), todavía es costoso y produce mucho overhead.

4) Evitar dinámicamente (Avoidance).

Antes de explicar esta estrategia es necesario saber:

El Estado del Sistema: es la asignación actual de recursos a procesos, definido por el número de recursos disponibles, número de recursos asignados y el máximo pedido de cada proceso. Este estado es registrado mediante una función de contabilidad en el S.O.

Un **estado es seguro** es cuando existe un orden tal en el que los procesos pueden llevarse a cabo por completo, sin resultar en Deadlock. Se dice que un estado es seguro si existe una secuencia de otros estados que lleva a que todos los procesos obtengan sus recursos y terminen su ejecución en un cierto tiempo. Un **estado inseguro** es aquél en que su ejecución conduce a un abrazo mortal.

Para evitar una situación de Abrazo Mortal se necesita contar con mucha información por adelantado que debe ser provista por el S.O..

Esta estrategia, se basa en asegurar que los procesos y los recursos permanezcan en un estado seguro: cuando un proceso solicita un recurso, se asume que le fue otorgado, entonces se chequea el sistema y se determina en que estado se encuentra el mismo. De ser un estado seguro, el recurso es efectivamente entregado al proceso, ya que de lo contrario el proceso queda bloqueado hasta que sea seguro entregárselo.

Para ello se cuenta con dos protocolos:

1. No comenzar un proceso si las demandas pueden incurrir en Deadlock
2. No asignarle a un proceso en ejecución otro recurso si eso lo puede conducir a un Deadlock.

Las ventajas de esta estrategia con respecto a la de DETECCIÓN son las de no necesitar la aplicación del rollback ni la sustracción de un recurso a un proceso y la de menor existencia de restricciones.

Las desventajas son que no siempre un proceso conoce de antemano los recursos que va a necesitar durante su ejecución y debe existir un número fijo de recursos a asignar y un número fijo de procesos.

Para el desarrollo de esta estrategia varios autores definen matrices para guardar la información. Por ejemplo Crocus presenta tres matrices. La primera describe el estado inicial del sistema, proporcionando la cantidad total de recursos que existen en cada clase, este vector permanece constante. La segunda matriz define los recursos asignados a los procesos y en la última los recursos solicitados por los procesos. Así cada vez que se solicite un recurso se modificaran los datos de las matrices y por ecuaciones se determinará si el estado del sistema es seguro o no.

La ecuación a la que se hace referencia es $A - B \leq C + D$, siendo:

A = Cantidad de recursos solicitados
B = Recursos asignados
C = Cantidad de recursos disponibles
D = Cantidad de recursos liberados por procesos terminados.

Stallings propone el siguiente ejemplo:

Hay cuatro procesos P_1 a P_4 y tres Recursos R_1, R_2 y R_3. La cantidad total de recursos existentes en el sistema es $R_1 = 9$, $R_2 = 3$ y $R_3 = 6$. En el estado actual se dispone de $R_1 = 0$, $R_2 = 1$ y $R_3 = 1$. La pregunta es si el sistema está en un estado seguro con los pedidos de recurso expresado en las siguientes matrices:

	MÁXIMO REQUERIMIENTO				ASIGNACIÓN				
	P1	P2	P3	P4	P1	P2	P3	P4	DISPONIBLE
R1	3	6	3	4	1	6	2	0	0
R2	2	1	1	2	0	1	1	0	1
R3	2	3	4	2	0	2	1	2	1

Paso 1) Se asigna todo lo solicitado al Proceso 2, quien se completa y devuelve al Vector DISPONIBLE los recursos que tenía asignado.

	MÁXIMO REQUERIMIENTO				ASIGNACIÓN				
	P1	P2	P3	P4	P1	P2	P3	P4	DISPONIBLE
R1	3	0	3	4	1	0	2	0	6
R2	2	0	1	2	0	0	1	0	2
R3	2	0	4	2	0	0	1	2	3

Paso 2) Se asigna todo lo solicitado al Proceso 1, quien se completa y también devuelve los recursos que tenía asignado.

	MÁXIMO REQUERIMIENTO				ASIGNACIÓN				
	P1	P2	P3	P4	P1	P2	P3	P4	DISPONIBLE
R1	0	0	3	4	0	0	2	0	7
R2	0	0	1	2	0	0	1	0	2
R3	0	0	4	2	0	0	1	2	3

Paso 3) Se completa el Proceso 3

	MAXIMO REQUERIMIENTO				ASIGNACIÓ				
	P1	P2	P3	P4	P1	P2	P3	P4	DISPONIBLE
R1	0	0	0	4	0	0	0	0	9
R2	0	0	0	2	0	0	0	0	3
R3	0	0	0	2	0	0	0	2	4

y cuarto paso se completa el último P_4 demostrando que el sistema estaba en un estado seguro.

Veamos otro ejemplo: Supongamos 5 procesos y tres recursos y se encuentra en el siguiente estado de ejecución:

	ASIGNADOS		
	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2

SOLICITUDES		
A	B	C
0	0	0
2	0	2
0	0	0
1	0	0
0	0	2

DISPONIBLES		
A	B	C
0	0	0

Se puede ejecutar la secuencia P2; P0; P3; P4 y P1 y todos terminan, o sea, esta en estado seguro. Pero, veamos el mismo ejemplo en que P2 pide un recurso mas de C:

	ASIGNADOS		
	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2

SOLICITUDES		
A	B	C
0	0	0
2	0	2
0	0	**1**
1	0	0
0	0	2

DISPONIBLES		
A	B	C
0	0	0

Ejecuta solamente P0 y todos los demás quedan en deadlock, o sea el estado de la última asignación fue inseguro.

Comentarios sobre como Evitar

Se pueden evitar los deadlocks mediante un algoritmo siempre y cuando dispongamos de cierta información con anticipación.

Para Silberschatz, Jensen, Tanenbaum y Stallings, dicha información consiste en el conocimiento sobre los futuros pedidos de recursos.

Un algoritmo muy conocido es el del banquero propuesto por Dijkstra. Para entender este algoritmo previamente debemos conocer si es que está en un estado seguro**.**

Entonces ahora se podrá explicar como funciona el algoritmo del banquero, consiste en advertir si luego de una operación, pedidos de recursos, se estará en un estado seguro. Si lo hace, se le otorgará el recurso solicitado, de lo contrario se pospone hasta más adelante.

Tanto para este algoritmo como para los demás difieren en la cantidad de información requerida, la mayoría de los mismos requieren el máximo número de recursos que un proceso puede necesitar.

4.7.4. Conflicto en la comunicación entre procesos

Dado un conjunto T de procesos, podemos definir Deadlock del conjunto T cuando:

- Todos los procesos del conjunto T están bloqueados a la espera de mensajes.
- Los procesos del conjunto T están a la espera de mensajes de procesos pertenecientes al mismo conjunto.
- No existen mensajes en tránsito.

En estos casos son de aplicabilidad las estrategias ya mencionadas de Prevención, Detección y Recuperación.

Grafo de comunicación entre procesos

Con respecto a la aplicación de las soluciones propuestas en el punto anterior para el caso de conflicto en la comunicación entre procesos, debemos mencionar que existe una gran diferencia en la

forma en que se pueden determinar los Deadlocks, ya que el grafo presentaría forma de nudo, lo que implica que un ciclo en el grafo no determina un Deadlock como lo explicado anteriormente.

Los grafos de comunicación entre procesos están formados por un conjunto de vértices constituido por los procesos involucrados y un conjunto de aristas. En este caso una arista de un proceso P_h a un proceso P_k implica que el proceso P_h está esperando un mensaje de P_k .

En el grafo siguiente podemos ver dos situaciones. En el primer grafo, si bien hay un ciclo, esto no implica que haya Deadlock, ya que el proceso P_5 no está aguardando mensajes de ningún otro proceso y puede recibirse el mensaje de P_1.

En el segundo caso, podemos ver claramente un Deadlock, ya que se cumplen las tres situaciones mencionadas anteriormente.

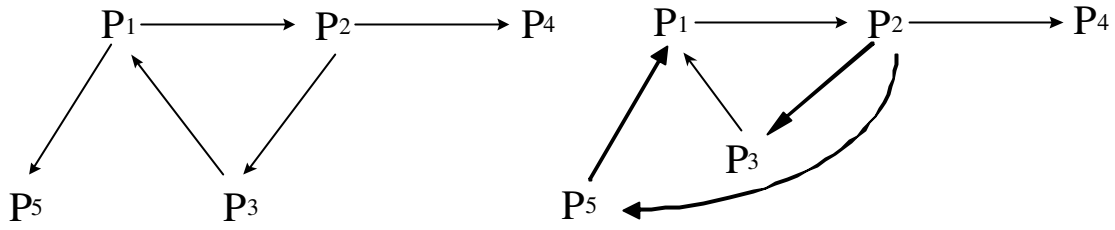


Fig. 4.15 Grafo de Comunicación entre Procesos.

Conclusión sobre tratamiento de deadlock

La elección de la estrategia para el tratamiento de deadlocks es una tarea difícil. En primer lugar hay que hacer un análisis y detectar cuales son las variables más relevantes para la toma de decisión. Si la probabilidad de ocurrencia es relativamente baja, la mejor alternativa es ignorarlo; ya que su costo no será relevante. Adoptar otras soluciones implicará analizar e implementar aquella que mejor responda al criterio del decididor tomando como base el sistema del que se trate.

Así y todo hay autores que consideran que ninguna de las cuatro estrategias por separado es buena. Una opción es elegir una estrategia adecuada para cada tipo de recurso. Esta idea está basada en el supuesto que considera que los recursos pueden ser particionados en clases y ordenados jerárquicamente. Supuestamente un deadlock no podría involucrar a más de una clase.

Ejercicio de abrazo mortal como ejemplo:

Cenicienta y el Príncipe se divorcian. Para dividir sus propiedades, tienen el siguiente algoritmo: cada mañana, cada uno puede enviar una carta al abogado del otro pidiendo un ítem de propiedad conjunta. Han acordado que si ambos descubren que han pedido el mismo ítem el mismo día, el día siguiente se enviará una carta cancelando el pedido. Entre sus propiedades figuran su perro Woofer con su cucha, su canario Tweeter con su jaula. Debido a que los animales aman a sus casas, se acordó que no se los separaría de ellas. Tanto La Cenicienta como el Príncipe desean desesperadamente a Woofer. Se van de vacaciones (por separado). Cada uno programó una computadora para manejar el problema. Cuando vuelven, las computadoras siguen negociando ¿por qué? ¿Es posible el abrazo mortal? Discutir los argumentos.

Respuesta: Las computadoras siguen negociando porque no existe una solución posible, produciéndose inanición.

El abrazo mortal no es posible porque para que lo sea se tienen que dar las cuatro condiciones en forma simultánea, y en este caso no se cumple la condición que sostiene que un recurso no puede ser sustituido por otro (en el ejemplo, "... han acordado que si ambos descubren que han pedido el mismo ítem el mismo día...").

Inanición

Este estado se produce por la espera prolongada a que son sometidos los procesos debido a la falta de asignación de recursos. Esta espera se produce porque hay otros procesos (por ejemplo que tienen mayor prioridad) que se adelantan adquiriendo los recursos que necesitan los que están esperando. Así se ven postergados indefinidamente en sus pretensiones de adquirir los recursos necesarios para completarse. Este envejecimiento (aging) puede llegar a ser infinito si no se descubre esta situación o el S.O. no modifica la estrategia para darle por ejemplo mas prioridad a medida que pasa el tiempo de espera.

Problema de la cena de los filósofos

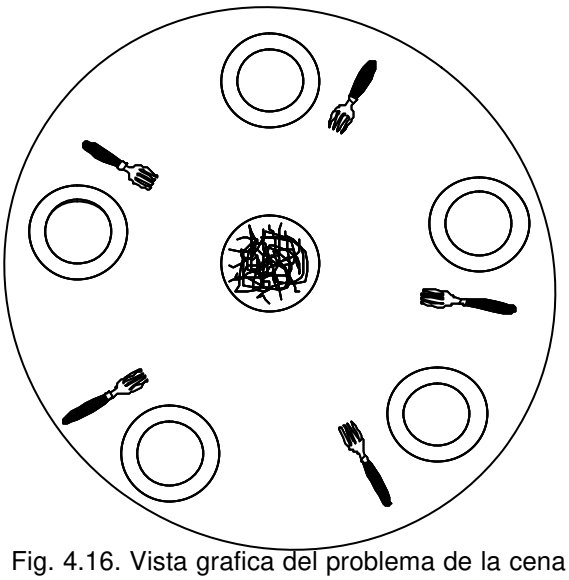


Fig. 4.16. Vista grafica del problema de la cena de los filósofos

Para ejemplificar los deadlock existe un problema conocido como la cena de los filósofos propuesto por Disjktra, en donde cinco filósofos viven juntos en una casa, la vida de estos filósofos consiste principalmente en comer y pensar. Con el paso del tiempo los filósofos han acordado que la comida que más contribuye para pensar era el espagueti.

Estos cinco filósofos tienen una manera de comer muy especial y reglada en donde todos se sientan en una mesa circular con cinco platos y cinco tenedores estando ubicado en el centro de la mesa un recipiente con los espaguetis, entonces cada filósofo posee un plato y un tenedor, pero aquí se presenta el inconveniente, para poder comer cada filósofo necesita obligatoriamente dos tenedores por lo que debe pedir un tenedor a cualquier otro filósofo siempre y cuando se encuentre a lado suyo. Otro punto es que ningún filósofo puede retener su tenedor si no desea comer y su compañero se lo pide.

Aquí se ve claramente el problema de deadlock e inanición ya que si todos al mismo tiempo toman su tenedor y quieren comer todos juntos se produce un deadlock y a su vez todos los procesos mueren por inanición.

Existen varias soluciones y las mejores implementaciones se encuentran realizadas con semáforos.

Una solución posible es que se le permita el acceso al “comedor” solo cuatro filósofos por vez, de esta manera con solo cuatro filósofos sentados en la mesa se puede asegurar que por lo menos uno comerá. Esta solución esta libre de deadlock y de inanición.

4.8. Mecanismos de concurrencia en distintos S.O.:

A) UNIX

UNIX provee una variedad de mecanismos para la comunicación entre procesos y la sincronización.

Pipes: Es un buffer circular(tipo FIFO) de tamaño fijo, que permite a dos procesos comunicarse siguiendo el modelo productor-consumidor. El sistema operativo se encarga de que solo un proceso acceda al buffer a la vez. Hay 2 tipos: las pipes con nombre y las pipes sin nombre (que pueden ser accedidas solo por procesos relacionados).

Mensajes: UNIX provee las system calls *sndmsg* y *msgrcv* para el paso de mensajes. Asociada a c/proceso hay una cola que funciona como mailbox. Los procesos son suspendidos cuando tratan de leer una cola vacía o escribir una llena.

Memoria compartida: Es un bloque de memoria virtual compartida por múltiples procesos. A un proceso se le dará permiso de escritura - lectura o solo - lectura y trabajara con la memoria compartida con las mismas instrucciones con las que trabaja su espacio de memoria. La exclusión mutua corre por cuenta del proceso.

Semáforos: En UNIX system V se usan generalizaciones de las primitivas *wait* y *signal* que pueden incrementar y decrementar el valor del semáforo en mas de 1 unidad. Un semáforo contiene los siguientes elementos:

- Valor actual del semáforo.
- PID del ultimo proceso que opero el semáforo.
- El número de procesos esperando que el valor del semáforo sea superior al actual.
- El número de procesos esperando que el valor del semáforo sea cero.

Los semáforos son creados en conjuntos de uno o más y se puede operar sobre el conjunto de semáforos.

Señales: Son un mecanismo de software que informa a un proceso de la ocurrencia de evento asincrónico; a diferencia de las interrupciones no hay prioridades entre señales. La señal se entrega actualizando un campo en la entrada de la tabla de procesos del proceso que la recibe. Como las señales son mantenidas como un bit, las señales de un mismo tipo no pueden ser guardadas en cola.

B) PRIMITIVAS PARA LA SINCRONIZACIÓN DE THREADS EN SOLARIS

Además de los mecanismos anteriormente nombrados, Solaris soporta cuatro primitivas para la sincronización de threads (nivel kernel y usuario).
La ejecución de una primitiva crea una estructura de datos para la sincronización. Una vez que es creado este objeto se puede entrar (adquisición y encierro) y se puede liberar (abrir). Si se intenta el acceso sin utilizar la primitiva apropiada a algo supuestamente bloqueado, se obtendrá el acceso; las primitivas son herramientas para la sincronización, no un mecanismo para la mutua exclusión o prevención de deadlock.

Cerradura Mutex: Previene que más de un thread proceda cuando la cerradura es adquirida.

mutex_enter()	Adquiere la cerradura, bloqueante si otro thread ya la tiene.
mutex_exit()	Libera la cerradura, desbloquea a un thread en espera (sí lo hay).
mutex_tryenter()	Adquiere la cerradura si otro thread no la tiene, sino de todas formas no se bloquea. Este permite usar busy waiting en los threads nivel usuario.

Semáforos

sema_p()	Decrementa el semáforo, potencialmente bloqueante.
sema _v()	Incrementa el semáforo, potencialmente desbloqueante.
sema _tryp()	Decrementa el semáforo si el bloqueo no es requerido. Este permite usar busy waiting en los threads nivel usuario.

Cerradura lectores/escritor: Solo permite el acceso de un thread(el escritor) mientras se escribe, múltiples threads(lector/es) mientras solo se lee.

rw_enter)	Intenta adquirir la cerradura como lector o escritor.
rw_exit()	Libera la cerradura como lector o escritor.
rw_tryenter()	Adquiere la cerradura si el bloqueo no es requerido.
rw_downgrade	Un thread que a adquirido una cerradura de escritura la convierte en una de lectura. Todo thread escritor permanece esperando hasta que este thread libere su cerradura.
rw_tryupgrade	Intenta convertir una cerradura de lectura en una de escritura.

C) MECANISMOS DE CONCURRENCIA DE WINDOWS NT

NT implementa la sincronización con una familia de objetos para la sincronización:

- Procesos
- Threads
- Archivos
- Entradas por consola
- Notificación de cambio de archivo
- Mutex
- Semáforo
- Evento
- Reloj de tiempo esperable

Los cuatro primeros no son objetos dedicados a la sincronización pero también se usan.
La instancia de un objeto puede esta en estado señalado o no–señalado. Un thread puede ser suspendido en un objeto en estado no–señalado, cuando ese objeto entra en estado señalado el thread es liberado.

4.9. BIBLIOGRAFÍA RECOMENDADA PARA ESTE MÓDULO

1. Operating Systems. (Second Edition), Stallings Willams; Prentice Hall, Englewood Cliff, NJ. 1995, 702 pages.

2. Applied Operating Systems Concepts (First Edition), Silberschatz, A., Galvin P. B; and Gagne G.. Wiley. 2003.

3. Operating Systems Concepts (Fifth Edition), Silberschatz, A. and Galvin P. B; Addison Wesley 1996, 850 pages.

4. Operating Systems Concepts and Design (Second Edition), Milenkovic, Millan; Mc Graw Hill 1992

5. Modern Operating Systems, Tanenbaum, Andrew S.; Prentice - Hall International 1992, 720 pág.

6. Operating Systems, Design and Implementation, Tanenbaum, Andrew S.; Prentice - Hall International, 1987, 800 pág.

7. Fundamentals of Operating Systems, Lister, A.M. ; Macmillan, London 1979

8. Operating System Design - The XINU Approach, Comer, Douglas; Prentice Hall 1984

9. Operating System, Lorin, H., Deitel, H.M.; Addison Wesley; 1981;

10. A discipline of Programming, Dijkstra, Edward W.; Englewood Cliffs, Prentice Hall; 1976

11.The Art of computer Programming (Volumen 1, 2, y 3); Knuth, Donald; Addison - Wesley Publishing Co.; 1974

12. Operating Systems: Structures and Mechanisms; Jensen Philippe; Academic Press 1985

13. The UNIX Operating System, Christian, K.; John Wiley; 1983

14. UNIX System Architecture, Andleigh, Prabhat K.; Prentice - Hall International; 1990

15. The UNIX Programming Environment, Kernighan, B.W. and Pike, R.; Prentice - Hall International 1984

16. The UNIX System V Environment, Bourne, Stephen R.; Addison Wesley; 1987

17. UNIX Utilities a Programmerer's Guide.; Tare, R. S.; Mc Graw-Hill.; 640 pág.

18. Tricks of the UNIX Masters. ; Sage, R. S.;

19. UNIX System Readings and Applications. (Vol I y II); AT&T Bell Laboratories.; Prentice Hall Englewood Cliffs; 1987- 416 pag.

20. Introducing UNIX System V; Morgan, R. and McGilton, H.; Mc Graw Hill; 480 pág.

21. Sistemas Operativos Conceptos y diseños.(Segunda Edición); Milenkovic Milan; Mc Graw Hill; 1994.

22. Sistemas de Explotación de Computadores; CROCUS; Paraninfo; 1987 424 pág.

23. Sistemas Operativos MS-DOS, Unix, OS/2, MVS, VMS, OS/400; E. Alcalde,J. Morera, J.A. Pérez - Campanero.; Mc Graw Hill; 1992.

GLOSARIO DE TÉRMINOS EN IDIOMA INGLÉS

Reader	Writer	Threads	fork
join	Label	begin	end

Race condition	for	print	Busy waiting
Spinlock	Master-Slave	Switcher	Program
While	true	False	do
swap	compare	System Call	down
up	block	wakeup	sleep
enqueue	dequeue	communication link	send
receive	mesage	mailbox	link
server	accept	daemons	Bounded Buffer
Buffer	Advance	Deadlock	Resource allocation graph
Hold and Wait	No preemption	Kernel	Rollback
Check point	Overhead	Backup	Time out
Time Stamping			

GLOSARIO DE TÉRMINOS EN CASTELLANO

Sincronización entre procesos	Comunicación entre procesos
Programa secuencial	Programa Concurrente
Velocidad de ejecución	Mecanismos de sincronización
Mutua Conclusión	Grafos de precedencia
Condiciones de concurrencia	Especificaciones concurrentes
Instrucciones Fork - Join	Proceso padre
Proceso hijo	Proceso independiente
Proceso interactuante	Condición de concurso
Puntos de entrada de un recurso	Región Crítica
Recurso crítico	Protocolo de sincronización
Propiedades para usar una Región Crítica	Algoritmos de sincronización
Espera Activa	Espera ocupada por turnos
Alternancia estricta	Mecanismos de hardware para sincronización
Instrucciones TAS	Instrucciones CAS
Cola de espera	Primitivas
Semáforo	Valor del Semáforo
Ejecución atómica	Algoritmo sin espera activa
Semáforo MUTEX	Semáforo Binario
Regiones Críticas Condicionales	Monitor
Comunicación directa	Comunicación indirecta
Mensaje	Vínculo
Buzón	Send asincrónico
Send condicional	Receive incondicional
Receive Condicional	Sincronización mediante mensajes
Rendez- Vouz	Buffer de Mensajes
Llamada a Procedimiento Remoto	Comunicación semisincrónica
Paso de Mensajes	Memoria Compartida
Modelo Productor - Consumidor	Abrazo Mortal - Deadlock
Condiciones de Coffman	Grafo de asignación de recursos
Estrategias para tratar Deadlocks	Estado del Sistema
Estado seguro	Estado Inseguro
Inanición	Algoritmo del Banquero
Costo de Recuperación	Conflicto entre las Comunicaciones entre procesos

ACRÓNIMOS USADOS EN ESTE MÓDULO

R	Reader	W	Writer
S	Sentencias	etiq	etiqueta - Label
PCB	Process Contol Block	Cobegin	Concurrent begin
Coend	Concurrent end	R.C.	Región Crítica
var	variable	TAS	Test And Set
CAS	Compare and Swap	TSL	Test set lock
CPU	Central Processing Unit	Q(v)	Cola de espera
Typedef	Type definition	struct	structure
MSG	Mensage	t	transmisor
r	receptor	NºMSG	Número de mensaje
IPC	inter process Communication	RPC	Remote Procedure Call

Max	máximo	p	productor
c	consumidor	FIFO	First In First Out

ANEXO 4.a.- VP: Nueva operación para semáforos

Introducción

Uno de los principales mecanismos para poder implementar la concurrencia de procesos de manera óptima son los semáforos, creados por Dijkstra en 1965.
Un semáforo es una variable entera no negativa en la cual solo operaciones P y V son permitidas. La construcción de un semáforo es soportada por muchos sistemas operativos y es frecuentemente usada para implementar otras estructuras de sincronización.. Debido a que la estructura del semáforo es de bajo nivel, un programa basado en semáforos puede contener errores de sincronización que son muy difíciles de detectar.
La nueva operatoria **VP**, cuya sintaxis es VP(s1,s2) donde s1 y s2 pueden ser los mismos o bien diferentes semáforos. Así una ejecución VP(s1,s2) para un proceso T equivale a la secuencia "V(s1); P(s2)" con la salvedad de que cuando T ejecuta la primitiva V(s1), se asegura que será el próximo proceso al acceder al recurso s2. Asumimos que la cola del semáforo es FIFO y que el orden de bloqueo de procesos en una cola se semáforos es el mismo orden en el cual se ejecutan P operaciones en el semáforo.

Ventaja y desventajas del operador VP

Un programa basado en semáforos frecuentemente contiene la secuencia "V(s1); P(s2)". Esta secuencia es usada con el fin de permitir realizar el proceso V(s1) y luego inmediatamente autobloquearse en P(s2). Asumimos que el acierto de este programa se basa en procesos siendo bloqueados en P(s2) en el mismo orden en el que ejecutan V(s1). Si un cambio de contexto ocurre inmediatamente después de la ejecución de V(s1) por el proceso P1, la siguiente secuencia de eventos podría ocurrir: el proceso P2 ejecuta P(v2) y se autobloquea en s2, el proceso P3 ejecuta V(s2) con la intención de despertar P1, pero en lugar de eso despierta P2. Esta situación puede resultar en ejecuciones erróneas que son difíciles de detectar. La prevención de esta situación requiere un uso muy cuidadoso de operadores P y V adicionales y una posible adición de semáforos. Usando los operadores VP se eliminan este tipo de error y hace que los programas basados en semáforos mas fáciles de entender y verificar.

ANEXO 4.b.- Implementación de la construcción de la región critica condicional (CCR)

Para usar CCRs, las variables compartidas en un programa concurrente son divididas en grupos llamados recursos. Un CCR es definido como "región r: cuando B hace, S termina", cuando r es el nombre del recurso, S es una lista de estados, y B es una expresión booleana referida como guardián de su CCR o como guardián de r. Ambos r y S pueden referenciar variables compartidas en r como así también variables locales para ejecutar procesos. La mutua exclusión es utilizada en CCRs por el mismo recurso. Los procesos que están esperando entrar al CCR para el recurso r pueden ser divididos en dos clases: nuevos y viejos procesos para r. Un proceso viejo para r es un proceso que esta intentando entrar a CCR para el recurso r y ya ha evaluado el guardián de este CCR por lo menos una vez. Un nuevo proceso para r es un proceso que esta intentando entrar al CCR para el recurso r, pero todavía no a evaluado al guardián del CCR.
Una correcta implementación de un CCR debe satisfacer los siguientes requerimientos:
Mutua exclusión: como ya dijimos en CCR se fuerza la mutua exclusión
Progreso: esto es que cuando un proceso intenta entrar a la región crítica y se dan las condiciones, este pase a ejecutar dentro de la misma por un tiempo finito.
Imparcialidad: si un proceso quiere entrar a la CCR, entonces ésta reevaluará si el proceso cumple las condiciones para ingresar a ella en un tiempo finito.
Lo que se pretende remarcar es que en la implementación de esta construcción la utilización de la operación VP nos garantizará que la sincronización sea segura, que los programas sean más fáciles de entender. Servirá para implementar monitores asegurándonos que los procesos accederán a los recursos en el mismo orden en que llegaron al monitor.

ANEXO 4.c.- Sincronización del problema productor/consumidor utilizando semáforos, monitores y Ada rendezvous

Introducción

Para comprender el modelo Productor/Consumidor debemos reconocer tres elementos fundamenta les: el productor, el consumidor y un buffer.Los productores son aquellos procesos que almacenan datos en un buffer.El buffer es donde se almacena temporalmente y que sirve para adaptar velocidades, en este modelo se trata de un buffer limitado.El consumidor es quien toma los datos que los productores dejaron en el buffer para realizar su procesamiento.

Variaciones del problema cliente-servidor (Productor-Consumidor) se presenta en procesos como colas de espera, control de memoria y transferencias de disco. El proceso productor debe detener su producción cuando el buffer esta lleno, y el proceso cliente debe dejar de consumir cuando el buffer esta vació. También, algún mecanismo debe prevenir que los procesos accedan concurrentemente al buffer.

Implementación de semáforos

Los semáforos son mecanismos de sincronización que fueron introducidos por Dijkstra. Normalmente, operadores como WAIT y SIGNAL operan sobre el semáforo de la siguiente manera: Cuando un proceso ejecuta un operador WAIT que tiene un valor en el semáforo de cero, dicho proceso es bloqueado. Cuando un proceso ejecuta un operador
SIGNAL y hay procesos esperando (bloqueados), uno de los procesos en espera es activado (puesto en la cola de espera de procesos listos para entrar); de otra manera, si no hay procesos en espera, el valor del semáforo es incrementado en uno. Se asume que el proceso bloqueado por el semáforo pierde el procesador y entra a una lista de espera en lugar que mantener al procesador en un estado de ocupado-esperando. La organización de la lista de espera es FIFO.
Los semáforos son generalmente considerados como mecanismos de bajos nivel. Son de difícil implementación por el hecho de la sincronización de las llamadas a las primitivas WAIT y SIGNAL.
Ejemplo de implementación de la sincronización con semáforos de dos procesos, un productor y un consumidor, que ejecutan concurrentemente:

```
Program prod_cons1;
Var
    in, out                  : integer;
    buff                     : array[0..n-1] of integer;
    not_empty, not_full, mutex : semaphore;

Procedure insert(x: insert);
Begin
  WAIT(not_full);
  WAIT(mutex);
  buff[in]:= x;
  SIGNAL(mutex);
  in:= (in+1) mod n;
  SIGNAL(not_empty);
End;

Procedure producer;
Var item: integer;
Repeat
  Generate item;
  Insert(item);
Until false;

Procedure remove(var y: integer);
Begin
  WAIT(not_empty);
  WAIT(mutex);
  Y:= buff[out];
  SIGNAL(mutex);
  Out:= (out+1) mod n;
  SIGNAL(not_full);
End;

Procedure consumer;
Var item: integer;
Repeat
  Remove (item);
  Use item;
Until false;

Begin
{main program}
In := 0; out:= 0; mutext:= 1; not_full:= n; not_empty:= 0;
 Cobegin
    Producer;
    Consumer;
 Coend;
End.
```

Implementación del monitor

Los monitores fueron propuestos por Brinch Hansen y refinados pro Hoare con el fin de implementar sincronización automática de procesos. Un modulo monitor encapsula datos mutuamente excluyentes y procesos que puedan acceder los datos protegidos. Los usuarios hacen llamadas a los procedimientos monitor, usando al monitor como una barra de estados para determinar cuando preceder y cuando suspender una operación.

Solo una llamada al procedimiento monitor puede ser activada al mismo tiempo. Esto protege a los datos dentro del monitor de accesos simultáneos por múltiples usuarios. Los usuarios que intentan acceder al monitor mientras este esta ocupado son bloqueados en una cola de espera.

Los operadores de sincronización del monitor se denominan WAIT y SIGNAL, como en los semáforos. Sin embargo, se utilizan variables condicionales son usadas en lugar de los semáforos y se comportan en forma diferente. No tiene valores numéricos como los semáforos.

La implementación de monitores no permiten al productor o al consumidor acceder a ningún de los mecanismos de sincronización de condicionales variables, y tampoco acceder a ninguna variable del monitor (incluyendo su buffer).

Ejemplo de la implementación del modelo productor consumidor con monitores.

```
Program prod_cons2;
 Monitor buffer;
        Var
          count, in, out: integer;
          buff: array[1..n-1]of integer;
          not_full, not_empty: condicion;

Procedure insert (x: integer);
  Begin
       If count > n then
WAIT(not_full);
       buff[in]:= x;
       in:= (in+1) mod n;
       count:= count + 1;
       SIGNAL(not_empty);
End;

Begin {monitor}
        count:= 0;
        in:= 0;
        out:= 0;
End;

Begin {unit program}
  Cobegin
        User_1;
        User_2;
   End;
End.
```

```
Procedure remove (var y: integer);
  Begin
       If count = 0 then WAIT
(not_empty);
       y:= buff[out];
       out:= (out+1) mod n;
       count:= count -1;
       SIGNAL(not _full);
  End;

 Procedure consumer;
 Var item: integer;
 Repeat
   Buffer.remove (item);
    Use item;
 Until false;

Procedure producer;
Var item: integer;
Repeat
  Generate item;
  Buffer.insert(item);
Until false;
```

Implementación de Ada Rendez-vous

Ada fue para el Departamento de Defensa de los Estados Unidos para ser usado como lenguaje primario de programación para los sistemas del departamento. Ada es rico en sintaxis y tiene un inmenso poder de programación. Desafortunadamente, es también un lenguaje extenso y algo complicado. Para la sincronización Ada usa ambas técnicas de monitor y una técnica llamada Rendez-vous, introducida

por C. A. R. Hoare. El Ada para la sincronización utiliza dos técnicas, monitores y *rendez-vous*. Ya que monitores fue descrita en el punto anterior ahora trataremos la sincronización con Ada Rendez-vous. En Ada, un programa es llamado proceso, y los procesos concurrentes: tareas. Rendez-vous ocurre cuando alcanza un punto o posición determinado en el código. SI uno alcanza su punto de Rendez-vous antes que el otro, se bloqueara y perderá al procesador. Una vez que el Rendez-vous esta completo ambas tareas pueden continuar. Luego de esta breve introducción sobre el tema veamos la implementación del modelo productor consumidor con Ada Rendez-vous.

```
Procedure prod_cons3;
  Task buffer is
        Entry insert(x : in integer);
        Entry remove(y: out integer);
  End buffer;
  Task producer;
  Task consumer;
Task body buffer is
        Count : integer:= 0; in: integer:= 0; out: integer:= 0;
        Buff: array(1..n-1)of integer;
        Begin
                Loop
                        Select
                                When count < n =>
                                  Accept insert(x: in integer) do
                                        Buff(in):= x;
                                  End insert;
                                In:= (in + 1) mod n; count := count + 1;
                        Or
                                When count > 0 =>
                                  Accept remove(y: out integer) do
                                        Y:= buff(out);
                                  End remove;
                                Out:= (out + 1) mod n; count := count – 1;
                        End Select;
        End loop;
End buffer;

Begin –- main procedure
        Null;
End prod_cons3;
```

```
Task body producer is
 item: integer;
Loop
  Generate item;
  Buffer.insert(item);
End loop;
End producer;
```

```
Task body consumer;
Var item: integer;
Loop
  Buffer.remove
(item);
  Use item;
End loop;
End consumer;
```

El Rendez-Vous extendido consiste en una extensión del mecanismo anterior, con la diferencia de que el proceso receptor solamente envía una respuesta al transmisor después de la ejecución de un cuerpo de comandos que operan sobre el mensaje recibido. Mientras tanto el transmisor queda bloqueado a la espera hasta que el cuerpo de comandos haya terminado sus tareas. La respuesta puede poseer parámetros que contengan resultados de los cálculos efectuados por el receptor.

Pipes

Los pipes o tuberías son un medio que permiten la comunicación entre procesos. Estos funcionan como una cola de tipo FIFO en el cual un proceso escribe y el otro lee.

Tipos

Existen dos tipos de pipes, los pipes con nombre y los pipes sin nombre o simplemente pipes. Ambos son iguales a excepción por la forma en que un proceso los accede inicialmente.

Sin nombre

Son creados mediante el System Call pipe (s). Este revuelve un descriptor para lectura y otro para escritura.

Las propiedades mas útiles del pipe se presentan cuando un proceso crea un proceso hijo. También es muy útil para la comunicación entre ambos, pudiendo tomar cualquiera de los dos el rol de escritor o lector sobre el mismo.

Como los pipes requieren almacenamiento temporal, presentan un problema en cuanto a la cantidad de espacio necesario. Una solución tradicional es provista por el filesystem que permite escribir parte de los datos en disco. Otra es llevar los datos a una memoria superior pero con un mecanismo de comunicación más rápido.

Con nombre

Otro de los problemas de los pipes sin nombre es que el paso de datos debe ser entre procesos que se encuentren relacionados, esto se soluciona mediante la implementación de pipes con nombre, los cuales permiten la comunicación entre dos procesos cualesquiera. La tubería se va a encontrar en el filesystem.

Señales

Función de las señales

A través de las señales se le informa a los procesos la ocurrencia de eventos asincrónicos. Las señales pueden ser enviadas por un proceso hacia otro con el uso del System Call kill, o puede enviarlas el kernel internamente.

Clasificación

- Para la terminación de procesos: son utilizados cuando termina un proceso, así también cuando termina un proceso hijo.
- Para procesos inducidos a excepciones: se da en casos como cuando un proceso accede a una dirección que se encuentra fuera de su espacio de direcciones virtuales, en varios errores de hardware o al ejecutar instrucciones privilegiadas.
- Para condiciones irrecuperables durante un System Call.
- Causadas por condiciones de error inesperadas durante un System Call: como la creación de un System Call inexistente por el paso de un parámetro ilegal, la creación de un pipe que no tendrá lector, etc.
- Generadas por un proceso en modo usuario: como cuando un proceso desea recibir una señal de alarma después de un tiempo determinado.

Manipulación de señales

La manipulación de señales se refiere al tipo de tratamiento o postura que se toma frente a la recepción de una señal y los problemas que se pueden presentar.

Tratamiento

Desde el punto de vista del kernel, este trata a las señales en el contexto del proceso que recibe la señal, Hay tres posturas que pueden tomarse frente a la recepción del mismo, por parte de los procesos: el proceso puede terminarse, ignorar la señal, o ejecutar una función particular a partir de ella.

Problemas

Pueden presentarse los siguientes problemas cuando se recibe una señal.

- **Querer atender las señales**

Si bien un proceso puede ignorar o terminarse cuando recibe una señal, se presentan algunos problemas cuando trata de manipular las señales. El problema principal es que puede darse una race condition, una solución para esto podría ser acumular las señales que se reciben para luego tratarlas, lo que trae problemas en cuanto al espacio para ello, o ignorarlas luego del llenado de la pila, pero se perdería información. Finalmente una buena solución es que el kernel bloquee la recepción de señales hasta que el proceso termine de tratar cada una.

- **Tomar señales que ocurren mientras el proceso está en un system call.**

En este caso el proceso se encuentra en estado durmiendo, con una prioridad de interrupción. El proceso deja de estar dormido volviendo al modo usuario, trata la señal y regresa del system call con un mensaje de error que indica esta interrupción. El usuario luego puede volver a ejecutar nuevamente el system call, pero sería conveniente que el kernel lo haga automáticamente.

- **Ignorar señales**

El problema que se presenta en este caso es que el kernel se da cuanta de que el proceso ignora la señal luego de que despierta y corre al mismo, que se encontraba en estado durmiendo con prioridad de interrupción. Una solución podría ser guardar las direcciones de las señales en la tabla de entradas del proceso, donde el kernel podrá fijarse si debe despertar al proceso para recibir la señal, o no.

- **Tratamiento especial de una "finalización de hijo"**

Cuando el proceso recibe esta señal, el kernel no la carga en la tabla de señales de entrada, o sea que la ignora. Luego si el proceso ejecuta un system call para verificar si hay finalización de hijos, el kernel le enviará una señal de finalización de hijos si tiene hijos en estado zombie.

Envío de señales

En UNIX se utiliza el system call kill para el envio de señales por parte de un proceso. El formato es el siguiente:

Kill (pid, signum)

Donde *signum* es el número de señal a enviar y *pid* es el identificador de procesos que recibirán la señal.

Si el pid es un entero positivo el kernel envía la señal al proceso con el número de pid.

Si es igual a 0, envía la señal a los procesos que se encuentren el mismo grupo del proceso que lo envía.

Si es igual a –1, envía la señal a todos los procesos que tengan el mismo ID que el emisor. Los únicos procesos que no recibirán la señal son el proceso 0 y 1.

Si es negativo pero distinto de –1, envía la señal a todos los procesos en el grupo de procesos igual al valor absoluto del pid.

AUTOEVALUACIÓN DEL MODULO 4:

Preguntas:

1.- ¿Las primitivas WAIT y SIGNAL pueden interrumpirse?.

2.- ¿El Paso de Mensajes sólo puede ser implementado en sistemas monoprocesador?.

3.- ¿En un direccionamiento Indirecto, todos los buzones pertenecen al Sistema Operativo?.

4.- ¿La programación concurrente requiere de mecanismos de sincronización y comunicación entre los procesos?.

5.- ¿Cuando un proceso que se encuentra dentro de un monitor realiza una operación WAIT sobre una variable de condición, el proceso espera fuera del monitor en una cola de procesos bloqueados asociada a dicha variable de condición?.

6. ¿La gran desventaja que tienen las instrucciones TSL (test and set instructions) es que no pueden trabajar en sistemas con múltiples procesadores?

7.- ¿Cuál es la diferencia entre la cooperación y la competencia entre procesos?

8.- ¿Qué operaciones se pueden realizar sobre un semáforo?

9.- ¿Cuál es la diferencia entre bloqueado y no bloqueado con respecto a los mensajes?

10.- ¿Qué condiciones están generalmente asociadas con el problema de lectores/escritores?

11.- ¿Qué son los recursos reutilizables y consumibles?

12.- ¿Cuáles son las condiciones de Coffman que deben estar presentes para que sea posible un deadlock?

13.- ¿Es siempre la espera ocupada menos eficiente(en términos de usar el tiempo del procesador)que una espera bloqueante? Explicar.

14.- ¿Cuáles son lasdiferencias del monitor con respecto al semáforo?

Múltiple Choice:

1.-La Exclusión Mutua: a) Deriva en dos problemas: el Deadlock e inanición. b) Trabaja sobre los recursos críticos. c) También se implementa en operaciones de lectura. d) Todas las anteriores son correctas. e) Todas las anteriores son incorrectas	**2.- Requisitos para la Exclusión Mutua:** a) Un proceso permanece en su sección crítica un tiempo finito. b) Se realizan suposiciones del tiempo que un proceso puede tardar en finalizar para mejorar las prestaciones del sistema. c) Un proceso no puede interrumpirse en su sección crítica. d) Todas las anteriores son incorrectas.
3.- Los semáforos: a) Pueden definirse como variables enteras. b) Se pueden inicializar con un valor negativo. c) Utilizan las señales WAIT y SIGNAL para cooperar entre procesos. d) La señal WAIT incremente el valor del semáforo. e) Todas las anteriores son incorrectas	**4.-¿Qué sucede cuando un proceso que está dentro del monitor ejecuta un WAIT?** a) Sale del monitor y se posiciona en la cola de entrada. b) Permanece en el monitor hasta que algún otro proceso desee entrar. c) Se ubica en la cola de procesos que esperan entrar de vuelta cuando cambie la condición. d) Todas las anteriores son incorrectas.
5.- Los puntos de diseño por los cuales el concepto de concurrencia son importantes son: a) Multiprogramación dado que es la gestión de varios procesos dentro de un sistema monoprocesador. b) Multiproceso dado que es la gestión de varios procesos dentro de un sistema multiprocesador. c) Proceso distribuido dado que es la gestión de varios procesos que ejecutan en sistemas de computadores múltiples y remotos. d) Todas las anteriores son correctas.	**6. El análisis de un grafo de asignación de recursos sirve para:** a) La prevención de interbloqueos. b) La evitación de interbloqueos. c) La detección de interbloqueos. d) La recuperación de interbloqueos. e) Describir si existe un interbloqueo f) Ninguna de las anteriores son ciertas
7. -El algoritmo de Peterson corresponde a: a) Una estrategia de sincronización de procesos. b) Un método de ordenación de sucesos en un sistema distribuido. c) Una política de sustitución de páginas al producirse un fallo	**8.-Cuando un proceso que se encuentra dentro de un monitor realiza una operación wait sobre una variable de condición:** a) El proceso espera dentro del monitor a que otro proceso haga un signal sobre la misma variable de condición.

de página. d) Una solución al problema de la exclusión mutua. e) Ninguna de las anteriores son ciertas	b) El proceso espera fuera del monitor en una cola de procesos bloqueados asociada a dicha variable de condición. c) El proceso espera, fuera del monitor, sobre la cola de entrada principal al monitor. d) Ninguna de las anteriores es cierta.
9.- La estrategia denominada "Algoritmo del Banquero" propuesta por Dijkstra es: a) Un algoritmo de detección de deadlock. b) Un algoritmo de predicción de deadlock. c) Un algoritmo de prevención de deadlock. d) Ninguna de las anteriores es correcta.	**10.- El grado de conocimiento en una relación de Cooperación entre procesos queda definida porque:** a) Tienen conocimiento directo de los otros procesos. b) Tienen conocimiento indirecto de los otros procesos. c) No tienen conocimiento de los otros procesos. d) Todas las anteriores son ciertas. e) Ninguna
11.- Las soluciones con instrucciones de máquina TSL tienen las siguientes ventajas: a) Es aplicable a cualquier número de procesos con memoria compartida tanto de monoprocesador como multiprocesador. b) No puede producir inanición. c) Es simple y fácil de verificar. d) Sirve para varias secciones críticas, cada una con su propia variable. e) Todas las anteriores f) Ninguna de las anteriores	**12.- La comunicación entre los procesos puede ser realizada mediante….** a) Monitores con notificación y difusión b) Un área común de memoria. c) El intercambio de mensajes llamado paso de mensajes. d) Semáforos Mutex e) Instrucciones de máquina TSL f) Todas las anteriores son ciertas. g) Ninguna de las anteriores es correcta
13.- La comunicación mediante el paso de mensajes se caracteriza por…. a) La comunicación entre procesos es ofrecida por el S.O. b) Para intercambiar grandes cantidades de datos. c) Evita conflictos. d) No usarse para comunicación entre computadoras. e) La forma de datos y ubicación están determinadas por los procesos y no por el S.O f) Máxima velocidad y conveniencia en la comunicación. g) Presenta problemas de protección y sincronización. h) Ninguna de las anteriores	**14.- La comunicación mediante un área común de memoria se caracteriza por….** a) La comunicación entre procesos es ofrecida por el S.O. b) Intercambiar pequeñas cantidades de datos. c) Evita conflictos. d) No usase para comunicación entre computadoras. e) La forma de datos y ubicación están determinadas por los procesos y no por el S.O f) Máxima velocidad y conveniencia en la comunicación. g) Presenta problemas de protección y sincronización. h) Ninguna de las anteriores
15.- El "modelo cliente - servidor" se caracteriza por…. a) Es sencillo y sin conexión. b) No es complejo y orientado a la conexión como OSI o TCP/IP. c) El cliente envía un mensaje de solicitud al servidor pidiendo cierto servicio. d) El servidor ejecuta el requerimiento y regresa los datos solicitados o un código de error si no pudo ejecutarlo correctamente. e) No se tiene que establecer una conexión sino hasta que ésta se utilice. f) Todas las anteriores son ciertas. g) Ninguna de las anteriores es correcta	**16.- En la comunicación Sincrónica. El Rendez-vouz se caracteriza porque…** a) Las primitivas de este tipo de comunicación se caracterizan por no bloquear a los procesos que las ejecutan. b) Es una primitiva síncrona en que el emisor se bloquea hasta que el receptor ha aceptado el mensaje y la confirmación regresa al emisor. c) Es importante en el caso del receptor ya que sigue ejecutando aunque no le llegue ningún mensaje. d) Permite una fuerte sincronización entre procesos ya que ambos, el emisor y el receptor, se bloquean hasta que se entrega el mensaje. e) Todas las anteriores son ciertas . f) Ninguna de las anteriores es correcta
17.- Los Semáforos fueron creados por… a) Silberschatz b) Tanenbaum c) Dijkstra d) Brinch Hansen e) Hoare f) Lamport. g) Ninguna de los anteriores	**18.- ¿Cuántos procesos activos puede haber dentro de un monitor?** a) No hay ningún límite. b) Sólo uno como máximo. c) Como máximo sólo uno en cada operación del monitor. d) Tantos como procedimientos tenga el monitor e) Tantos como se lo ha programado al monitor. f) Ninguna de las anteriores es cierta.
19.- ¿Qué es la región crítica de un proceso? a) Un trozo de código que cada usuario ejecuta para iniciar una sesión en un sistema multiusuario. b) Un trozo de código en el que se utiliza un recurso compartido y que se ejecuta de forma exclusiva c) La fase o etapa en la vida de un proceso concurrente en la cual accede a un recurso crítico para modificarlo o alterarlo. d) Un trozo de código que se ejecuta de forma exclusiva para competir por la utilización de un recurso compartido e) Ninguna es correcta.	20.- El protocolo que tiene que cumplir toda solución al problema de la exclusión mutua se aplica a las instrucciones …. a) A las instrucciones de la sección de entrada y a las de la sección de salida. b) A las instrucciones máquina. c) A las instrucciones de la sección crítica. d) A las instrucciones de CAS (Compare And Swap) e) Ninguna es correcta

Respuestas a las preguntas

1.- ¿Las primitivas WAIT y SIGNAL pueden interrumpirse?.
F. Ambas instrucciones son atómicas.

2.-¿El Paso de Mensajes sólo puede ser implementado en sistemas monoprocesador?.
F. Pueden ser implementados en sistemas distribuidos o en sistemas multiprocesador y monoprocesador de memoria compartida.

3.-¿En un direccionamiento Indirecto, todos los buzones pertenecen al Sistema Operativo?.
F. El buzón puede ser propiedad del proceso creador o puede ser considerado como propiedad del SO.

4.-¿La programación concurrente requiere de mecanismos de sincronización y comunicación entre los procesos?.
V La programación concurrente requiere de mecanismos de sincronización y comunicación entre los procesos.

5. ¿Cuando un proceso que se encuentra dentro de un monitor realiza una operación WAIT sobre una variable de condición, el proceso espera fuera del monitor en una cola de procesos bloqueados asociada a dicha variable de condición?.
Falso: El proceso espera dentro del Monitor

6. ¿La gran desventaja que tienen las instrucciones TSL (test and set instructions) es que no pueden trabajar en sistemas con múltiples procesadores?
Falso: No es una desventaja. Funciona en multiprocesador**.**

7.- ¿Cuál es la diferencia entre la cooperación y la competencia entre procesos?
Competencia entre procesos por los recursos: los procesos concurrentes entran en conflicto cuando compiten por el uso del mismo recurso. Si dos o más procesos necesitan acceder a un recurso durante el curso de su ejecución, y cada proceso no es consciente de la existencia de los otros y no se ve afectado por su ejecución, de aquí se obtiene que cada proceso debe dejar tal y como está el estado de cualquier recurso que utilice. Aunque no haya intercambio de información entre los procesos en competencia, la ejecución de un proceso puede influir en el comportamiento de los procesos que compiten.
Cooperación entre procesos por compartición: varios procesos pueden tener acceso a variables compartidas, archivos, o bases de datos compartidas. Los procesos pueden emplear y actualizar los datos compartidos sin hacer referencia a los otros procesos, pero son conscientes de que estos otros pueden tener acceso a los mismos datos. Los procesos deben cooperar para asegurar que los datos que se compartan se gestionen correctamente. Como los datos se guardan en recursos, también se presentan los problemas de control de exclusión mutua, deadlock e inanición. La única diferencia es que se puede acceder a los datos para lectura y para escritura, pero sólo las operaciones de escritura deben ser mutuamente excluyentes.
Cooperación entre procesos por comunicación: los distintos procesos participan en una labor común que une a todos los procesos. La comunicación es una manera de sincronizar o coordinar las distintas actividades. La comunicación puede caracterizarse por estar formada por mensajes de algún tipo. En el paso de mensajes no se comparte nada entre los procesos, por lo tanto no es necesario el control de la exclusión mutua. Sin embargo los problemas de deadlock e inanición siguen presentes.

8.- ¿Qué operaciones se pueden realizar sobre un semáforo?
Se pueden contemplar los semáforos como variables que tienen un valor entero sobre el que se define las siguientes operaciones:

1. Puede inicializarse con un valor no negativo.
2. La operación wait decrementa el valor del semáforo, si éste se hace negativo el proceso que ejecuta el wait se bloquea.
3. La operación signal incrementa el valor del semáforo, si el valor no es positivo se desbloquea un proceso bloqueado por una operación wait.

9.- ¿Cuál es la diferencia entre bloqueado y no bloqueado con respecto a los mensajes?

Envío bloqueante, recepción bloqueante(Rendez Vouz): esta combinación permite una fuerte sincronización entre procesos ya que ambos, el emisor y el receptor, se bloquean hasta que se entrega el mensaje.
Envío no bloqueante, recepción bloqueante: permite que un proceso envíe uno o más mensajes a varios destinos tan rápido como sea posible. El receptor se bloquea hasta que llega el mensaje solicitado. Esta es la combinación más útil.
Envío no bloqueante, recepción no bloqueante: nadie debe esperar. Como no hay bloqueo para hacer entrar en disciplina al proceso, esos mensajes pueden consumir recursos del sistema, incluido tiempo del procesador y espacio en buffer, en detrimento de otros procesos y del SO.

10.- ¿Qué condiciones están generalmente asociadas con el problema de lectores/escritores?
En este problema existe un área de datos compartida entre una serie de procesos. Hay una serie de procesos que sólo leen los datos, lectores, y otros que sólo escriben los datos, escritores. Se deben satisfacer tres condiciones:

a) Cualquier número de lectores pueden leer los datos simultáneamente.
b) Sólo puede escribir un escritor en cada instante.

Si un escritor está accediendo a los datos, ningún lector puede leer

11.- ¿Qué son los recursos reutilizables y consumibles?

- **Recursos reutilizables:** son aquellos que pueden ser usados con seguridad por un proceso y no se agotan con el uso. Los procesos obtienen unidades de recursos que liberan posteriormente para que otros procesos las reutilicen.
- **Recursos consumibles:** son aquellos que pueden ser creados y destruidos, es decir producidos y consumidos. Cuando un proceso adquiere un recurso éste deja de exis tir.

12.- ¿Cuáles son las condiciones de Coffman que deben estar presentes para que sea posible un deadlock?

1. **Mutua Exclusión:** sólo un proceso puede usar un recurso simultáneamente.
2. **Retención y esperar:** un proceso puede retener unos recursos asignados mientras espera que se le asignen otros.
3. **No expropiación:** ningún proceso puede ser forzado a abandonar un recurso que retenga.
4. **Espera circular:** existe una cadena cerrada de procesos, cada uno de los cuales retiene, al menos, un recurso que necesita el siguiente proceso de la cadena.

13.-¿Es siempre la espera ocupada menos eficiente(en términos de usar el tiempo del procesador)que una espera bloqueante? Explicar.

La espera ocupada o busy waiting es siempre menos eficiente porque el proceso en espera utiliza el procesador para verificar el valor de la variable que le dé el permiso para acceder a su sección crítica. En cambio, en la espera bloqueante se puede utilizar el procesador para algo más productivo.

14.- ¿Cuáles son las diferencias del monitor con respecto al semáforo?

El semáforo tanto la mutua exclusión como la sincronización los tiene que programar el programador.

En el monitor todas las funciones de sincronización están confinadas dentro del monitor y no tiene que ser programado por el programador.

En un monitor el acceso al recurso protegido es correcto para todos los procesos.

En un semáforo el acceso sólo será correcto si todos los procesos están correctamente programados.

Respuestas del múltiple choice.

1.- a, b.	2.- a.	3.- a, c.	4.- c.	5.- d.
6.- c, e.	7.- a, d.	8.- a.	9.- c.	10.- d.
11.- a, c, d.	12.- b, c.	13.- a, c.	14.- e, f, g.	15.- f.
16.- b, d.	17.- c.	18.- b.	19.- b,c.	20.- a.