



EQUIPO 2. Capitulo 5 y 6

Materia: Sistemas Operativos

Docente: Norma Edith Marin Martínez

Hora: M4

JOSUE CARLOS MORENO MAGALLANES	1846526	ITS
ELIUD JONATHAN LUCIO GARCÍA	2000116	IAS
EMILIO DE JESUS IBARRA GUTIERREZ	2000396	IAS
VICTOR ALFONSO DELGADO BAUTISTA	2006517	IAS
DAMARIS HERNANDEZ HERNANDEZ	2005278	IAS
GREGORIO MARTINEZ MARTINEZ	2014975	IAS



CAPITULO 5.

Concurrencia. Exclusión mutua y
sincronización.



5.1. Principios de la concurrencia

Concurrencia. Exclusión mutua y sincronización

- **Compartición de recursos globales:** Cuando varios procesos comparten recursos globales, como variables, la forma en que acceden y modifican esos recursos se vuelve crítica. Si no se gestiona adecuadamente, esto puede llevar a resultados inesperados o incorrectos.
- **Asignación óptima de recursos:** El sistema operativo enfrenta dificultades para asignar recursos de manera óptima entre los procesos. Por ejemplo, un proceso puede obtener acceso a un recurso y luego ser suspendido antes de usarlo, lo que complica la gestión de los recursos y puede conducir a problemas como el interbloqueo.
- **Localización de errores:** Debido a la naturaleza concurrente de la ejecución, los errores de programación pueden ser difíciles de localizar, ya que los resultados pueden ser no deterministas y no reproducibles.



5.1. Principios de la concurrencia

Concurrencia. Exclusión mutua y sincronización

Estas dificultades se aplican tanto a sistemas monoprocesador como a sistemas multiprocesador, ya que en ambos casos no se puede prever la velocidad relativa de ejecución de los procesos. Aunque en los sistemas multiprocesador también se enfrentan problemas derivados de la ejecución simultánea de múltiples procesos, en esencia, los desafíos son similares a los de los sistemas monoprocesador.

Es importante destacar que el procesamiento concurrente es un tema complejo en la informática, y estas dificultades subrayan la necesidad de implementar técnicas adecuadas de exclusión mutua y sincronización para garantizar la correcta interacción entre los procesos y evitar problemas como interbloqueo y resultados inesperados.



5.1. Principios de la concurrencia

CONDICIÓN DE CARRERA

Una condición de carrera sucede cuando múltiples procesos o hilos leen y escriben datos de manera que el resultado final depende del orden de ejecución de las instrucciones en los múltiples procesos.

INTERACCIÓN DE PROCESOS

Podemos clasificar las formas en que los procesos interaccionan en base al grado en que perciben la existencia de cada uno de los otros.



5.1. Principios de la concurrencia

REQUISITOS PARA LA EXCLUSIÓN MUTUA

Cualquier mecanismo o técnica que vaya a proporcionar exclusión mutua deber a cumplimentar los siguientes requisitos:

- La exclusión mutua debe hacerse cumplir: sólo se permite un proceso al tiempo dentro de su sección crítica, de entre todos los procesos que tienen secciones críticas para el mismo recurso u objeto compartido.
- Un proceso que se pare en su sección no crítica debe hacerlo sin interferir con otros procesos.



5.1. Principios de la concurrencia

REQUISITOS PARA LA EXCLUSIÓN MUTUA

- No debe ser posible que un proceso que solicite acceso a una sección crítica sea postergado indefinidamente: ni interbloqueo ni inanición.
- Cuando ningún proceso esté en una sección crítica, a cualquier proceso que solicite entrar en su sección crítica debe permitirse entrar sin demora.
- No se hacen suposiciones sobre las velocidades relativas de los procesos ni sobre el número de procesadores.
- Un proceso permanece dentro de su sección crítica sólo por un tiempo finito.
- Hay varias maneras de satisfacer los requisitos para la exclusión mutua. Una manera es delegar la responsabilidad en los procesos que desean ejecutar concurrentemente.



REQUISITOS PARA LA EXCLUSIÓN MUTUA

- No debe ser posible que un proceso que solicite acceso a una sección crítica sea postergado indefinidamente: ni interbloqueo ni inanición.
- Cuando ningún proceso esté en una sección crítica, a cualquier proceso que solicite entrar en su sección crítica debe permitirse entrar sin demora.
- No se hacen suposiciones sobre las velocidades relativas de los procesos ni sobre el número de procesadores.
- Un proceso permanece dentro de su sección crítica sólo por un tiempo finito.
- Hay varias maneras de satisfacer los requisitos para la exclusión mutua. Una manera es delegar la responsabilidad en los procesos que desean ejecutar concurrentemente.

5.2. EXCLUSIÓN MUTUA: SOPORTE HARDWARE

Se han ideado diversos algoritmos de software para lograr la exclusión mutua, siendo el más reconocido el algoritmo de Dekker. Aunque estas soluciones de software pueden resultar en una sobrecarga de procesamiento considerable y aumentan el riesgo de errores lógicos, su estudio ofrece una comprensión fundamental de los conceptos y problemas en la programación concurrente.

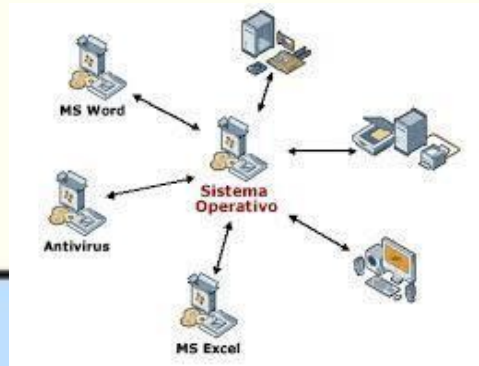
Para aquellos interesados, el Apéndice A del documento proporciona un análisis detallado de estas soluciones de software. Además de las soluciones software, en la sección también se exploran diversas soluciones hardware interesantes para abordar el problema de la exclusión mutua.



DESHABILITAR INTERRUPCIONES

En un sistema monoprocesador, los procesos concurrentes pueden entrelazarse, pero no solaparse. Un proceso se ejecuta hasta que realiza una llamada al sistema o es interrumpido.

Para lograr la exclusión mutua en este contexto, es suficiente evitar que un proceso sea interrumpido. Esto se puede lograr mediante primitivas proporcionadas por el núcleo del sistema, que permiten deshabilitar y habilitar las interrupciones.



5.2.



INSTRUCCIONES MÁQUINA ESPECIALES

En sistemas multiprocesador, donde múltiples procesadores comparten una memoria principal, operan de manera independiente y equitativa, careciendo de relación maestro/esclavo o mecanismos de interrupción para garantizar la exclusión mutua.

Para abordar esto a nivel de hardware, los diseñadores de procesadores han ideado instrucciones especiales de máquina que ejecutan acciones atómicas, como leer/ escribir o leer/ comprobar, en una sola posición de memoria durante un ciclo de instrucción.



5.3 Semáforos

Los semáforos son fundamentales para la cooperación entre procesos. Estos permiten que múltiples procesos trabajen juntos a través de señales simples, asegurando que un proceso espere en un punto específico hasta recibir una señal particular. Los semáforos, variables especiales utilizadas para la señalización, son clave en este proceso. Transmitir una señal a través de un semáforo implica ejecutar la primitiva `semSignal(s)`, mientras que recibir una señal se logra con `semWait(s)`; si la señal correspondiente aún no ha sido enviada, el proceso se detendrá temporalmente hasta que la transmisión ocurra.



5.3

Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que contiene un valor entero sobre el cual sólo están definidas tres operaciones:

1. Un semáforo puede ser inicializado a un valor no negativo.
2. La operación `semWait` decrementa el valor del semáforo. Si el valor pasa a ser negativo, entonces el proceso que está ejecutando `semWait` se bloquea. En otro caso, el proceso continúa su ejecución.
3. La operación `semSignal` incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación `semWait`.

```
struct semaphore {  
    int cuenta;  
    queueType cola;  
}  
  
void semWait(semaphore s)  
{  
    s.cuenta --;  
    if (s.cuenta < 0)  
    {  
        poner este proceso en s.cola;  
        bloquear este proceso;  
    }  
}  
  
void semSignal(semaphore s)  
{  
    s.cuenta++;  
    if (s.cuenta <= 0)  
    {  
        extraer un proceso P de s.cola;  
        poner el proceso P en la lista de listos;  
    }  
}
```



Orden de extracción

Los semáforos utilizan una cola para mantener a los procesos en espera. Con esto surge la pregunta sobre cómo decidir qué proceso debe salir primero de esta cola. La política preferida es FIFO (primero-en-entrar-primero-en-salir), lo que significa que el proceso que ha estado bloqueado por más tiempo será el primero en ser extraído de la cola. Cuando un semáforo está definido bajo esta política, se le llama "semáforo fuerte". Sin embargo, si no se especifica un orden en el que los procesos son liberados de la cola, el semáforo se considera "débil".

5.3 Semáforos



Implementación

Es imperativo que las funciones `semWait` y `semSignal` sean implementadas como primitivas atómicas. Una manera obvia es implementarlas en hardware o firmware. A falta de esto, se han propuesto variedad de esquemas. La esencia del problema es la de la exclusión mutua: sólo un proceso al tiempo puede manipular un semáforo bien con la operación `semWait` o bien con `semSignal`. Así, cualquiera de los esquemas software, tales como el algoritmo de Dekker o el de Peterson, pueden usarse, si bien esto supondría una substancial sobrecarga de procesamiento.

5.3 Semáforos



Ejemplos de Implementación

La primera figura muestra la utilización de la instrucción "test and set". En esta implementación se incluye un nuevo componente entero, s.ocupado.

En un sistema de procesador único, es posible inhibir interrupciones durante las operaciones semWait y semSignal, tal y como se sugiere en la segunda figura.

```
semWait(s)
{
    while (!testset(s.ocupado))
        /* no hacer nada */;
    s.cuenta--;
    if (s.cuenta < 0)
    {
        poner este proceso en s.cola;
        bloquear este proceso
        (adem s.poner s.ocupado a 0);
    }
    else
        s.ocupado = 0;
}
```

```
semSignal(s)
{
    while (!testset(s.ocupado))
        /* no hacer nada */;
    s.cuenta++;
    if (s.cuenta <= 0)
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
    s.ocupado = 0;
}
```

```
semWait(s)
{
    inhibir interrupciones;
    s.cuenta--;
    if (s.cuenta < 0)
    {
        poner este proceso en s.cola;
        bloquear este proceso
        y habilitar interrupciones;
    }
    else
        habilitar interrupciones;
}

semSignal(s)
{
    inhibir interrupciones;
    s.cuenta++;
    if (s.cuenta <= 0)
    {
        extraer un proceso P de s.cola;
        poner el proceso P en la lista de listos;
    }
    habilitar interrupciones;
}
```


5.4 Monitores



Definición

El monitor es una construcción del lenguaje de programación que proporciona una funcionalidad equivalente a la de los semáforos, pero es más fácil de controlar. La construcción monitor ha sido implementada en cierto número de lenguajes de programación, incluyendo Pascal Concurrente, Pascal-Plus, Modula-2, Modula-3 y Java.

También ha sido implementada como una biblioteca de programa. Esto permite a los programadores poner cerrojos monitor sobre cualquier objeto. En concreto, para algo como una lista encadenada, puede quererse tener un único cerrojo para todas las listas, por cada lista o por cada elemento de cada lista.

5.4 Monitores

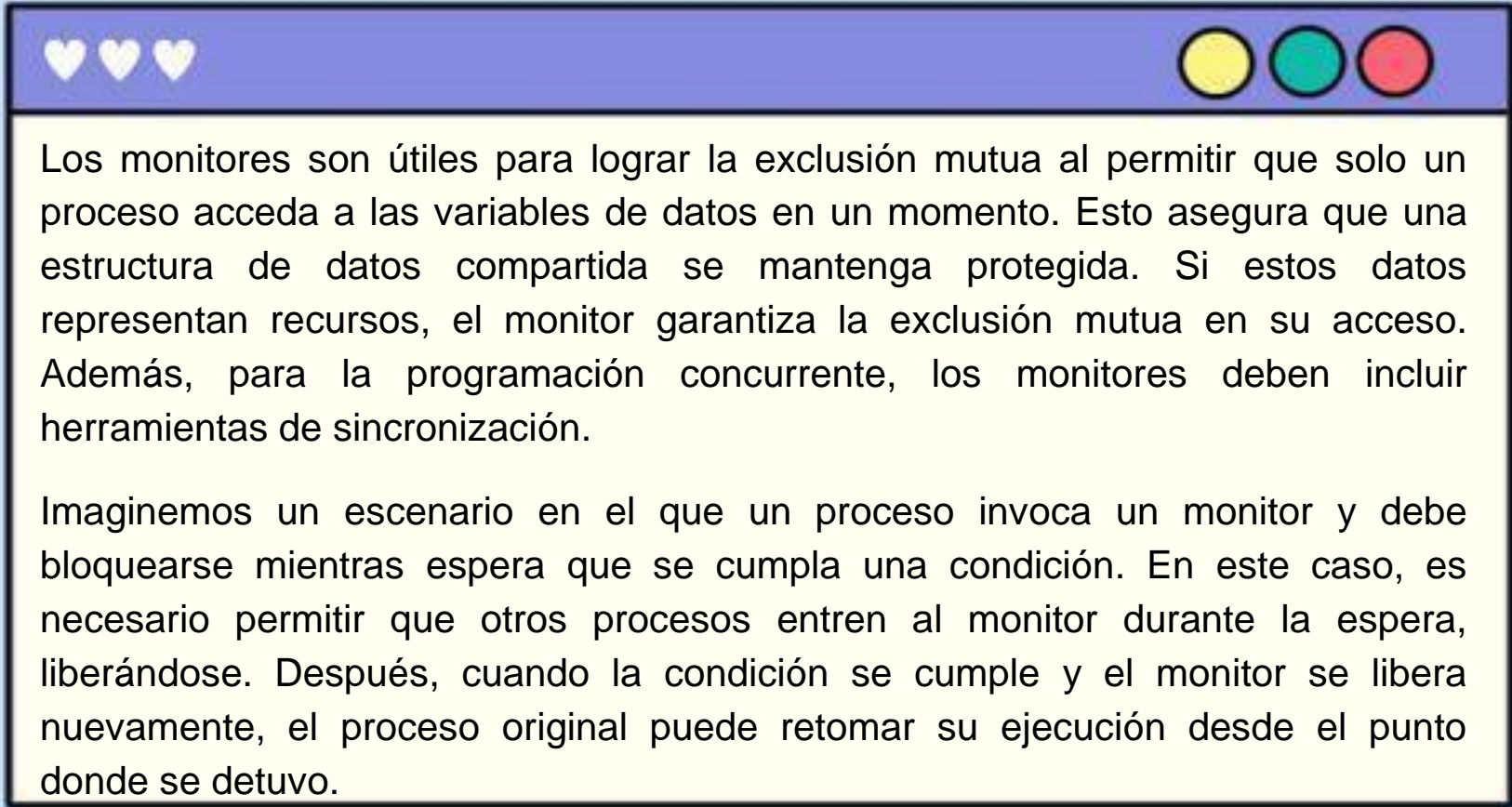


Monitor con señal (Versión de Hoare)

Un monitor es un módulo software consistente en uno o más procedimientos, una secuencia de inicialización y datos locales. Las principales características de un monitor son las siguientes:

1. Las variables locales de datos son sólo accesibles por los procedimientos del monitor y no por ningún procedimiento externo.
2. Un proceso entra en el monitor invocando uno de sus procedimientos.
3. Sólo un proceso puede estar ejecutando dentro del monitor al tiempo; cualquier otro proceso que haya invocado al monitor se bloquea, en espera de que el monitor quede disponible.

5.4 Monitores



Los monitores son útiles para lograr la exclusión mutua al permitir que solo un proceso acceda a las variables de datos en un momento. Esto asegura que una estructura de datos compartida se mantenga protegida. Si estos datos representan recursos, el monitor garantiza la exclusión mutua en su acceso. Además, para la programación concurrente, los monitores deben incluir herramientas de sincronización.

Imaginemos un escenario en el que un proceso invoca un monitor y debe bloquearse mientras espera que se cumpla una condición. En este caso, es necesario permitir que otros procesos entren al monitor durante la espera, liberándose. Después, cuando la condición se cumple y el monitor se libera nuevamente, el proceso original puede retomar su ejecución desde el punto donde se detuvo.

5.4 Monitores

Los monitores implementan la sincronización utilizando variables condición internas. Estas variables, específicas del monitor, se gestionan mediante dos funciones:

-`cwait(c)`: Pausa la ejecución del proceso en la condición `c`, permitiendo que otro proceso use el monitor.

-`csignal(c)`: Reactiva un proceso previamente bloqueado por `cwait` en la misma condición. Si hay varios procesos, se elige uno; si no, no hace nada.

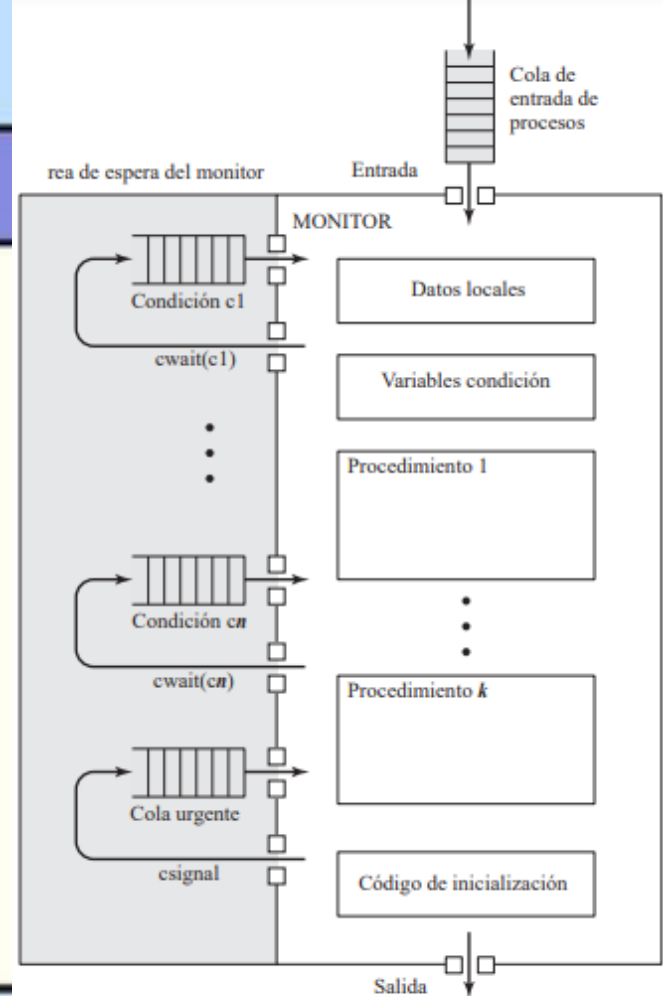


Figura 5.15. Estructura de un monitor.

5.4 Monitores



MODELO ALTERNATIVO DE MONITORES CON NOTIFICACIÓN Y DIFUSIÓN

La definición de monitor de Hoare requiere que sí hay al menos un proceso en una cola de una condición, un proceso de dicha cola ejecuta inmediatamente cuando otro proceso realice un csignal sobre dicha condición. Así, el proceso que realiza el “csignal” debe bien salir inmediatamente del monitor o bien bloquearse dentro del monitor.


Hay dos desventajas en esta solución:

1. Si el proceso que realiza el csignal no ha terminado con el monitor, entonces se necesitarán

dos cambios de proceso adicionales: uno para bloquear este proceso y otro para retomarlo

2. La planificación de procesos asociada con una señal debe ser perfectamente fiable.

5.4 Monitores



Lampson y Redell desarrollaron una definición diferente de monitor para el lenguaje Mesa. Su solución resuelve los problemas que se acaban de enunciar y aporta varias extensiones útiles. La estructura del monitor de Mesa se utiliza también en el lenguaje de programación de sistemas Modula-3. La primitiva “csignal” se sustituye por la “cnotify” y las sentencias “if” se remplazan por bucles “while”.

Una ventaja de los monitores de Lampson/Redell sobre los monitores de Hoare es que la solución de Lampson/Redell es menos propensa a error. En la solución de Lampson/Redell, dado que, al usarse la construcción while, cada procedimiento comprueba la variable condición después de ser señalado, un proceso puede señalar o difundir incorrectamente sin causar un error en el programa señalado. El programa señalado comprobará la variable relevante y si la condición no se cumple, volverá a esperar.

Otra ventaja del monitor Lampson/Redell es que se presta a un enfoque más modular de la construcción de programas.

5.5 PASO DE MENSAJES



El paso de mensajes es la herramienta básica de comunicación entre procesos. Un proceso puede mandar cualquier información a otro mediante este procedimiento. Deben satisfacerse dos requisitos fundamentales:

- Sincronización.
- Comunicación.



Ventajas del paso de mensajes:

- Fácilmente transportable de sistemas monoprocesadores a sistemas de memoria compartida o a sistemas distribuidos.
- Fomenta la modularidad y la arquitectura cliente-servidor.

La función real del paso de los mensajes se proporciona normalmente de un par de primitivas:

- send (destino, mensaje)
- receive (origen, mensaje)

5.5 PASO DE MENSAJES

 		
<p>Este es el conjunto mínimo de operaciones necesarias para que los procesos puedan entablar un paso de mensajes. El proceso envía información en forma de mensaje a otro proceso designado por destino.</p> <p>El proceso recibe información ejecutando la primitiva receive, indicando la fuente y el mensaje.</p>	Sincronización	Formato
	<i>Send</i> Bloqueante No bloqueante <i>Receive</i> Bloqueante No bloqueante Comprobación de llegada	Contenido Longitud Fija Variable
	Direccionamiento	Disciplina de cola
	Directo <i>Send</i> <i>Receive</i> Explícito Implícito Indirecto Estático Dinámico Propiedad	FIFO Prioridad

5.5 PASO DE MENSAJES



Sincronización: Es la comunicación de un mensaje entre dos procesos implica cierto nivel de sincronización entre los dos.

Direccionamiento: Especificar en la primitiva del envío que procesos deben recibir el mensaje, la mayor parte de las implementaciones permiten al proceso del receptor indicar el origen del mensaje.

Formato de un mensaje:

Cabecera

Cuerpo

Tipo de mensaje

ID de destino

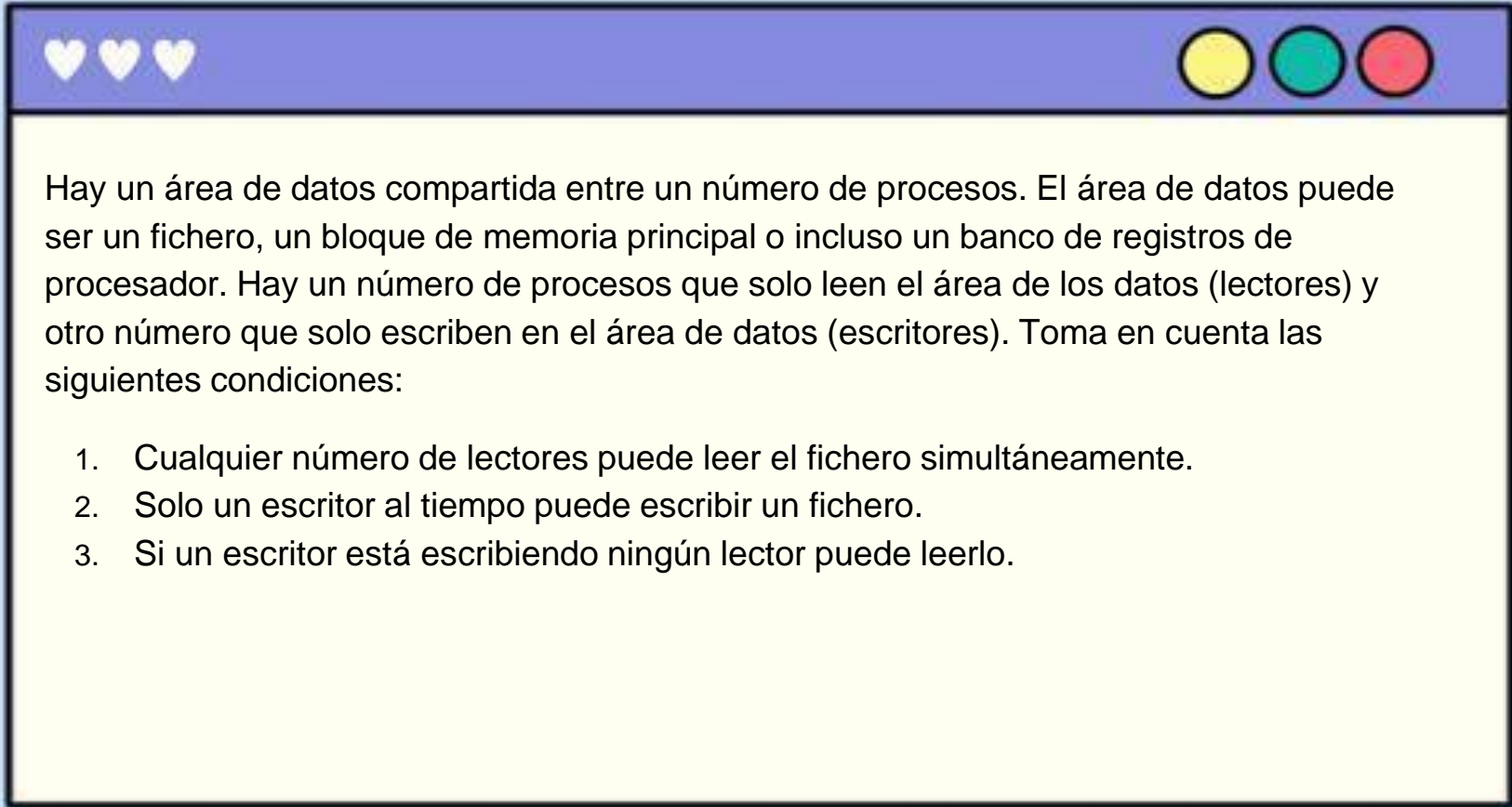
ID de origen

Longitud del mensaje

Información de control

Contenido de mensajes

5.6 EL PROBLEMA DE LOS LECTORES/ESCRITORES

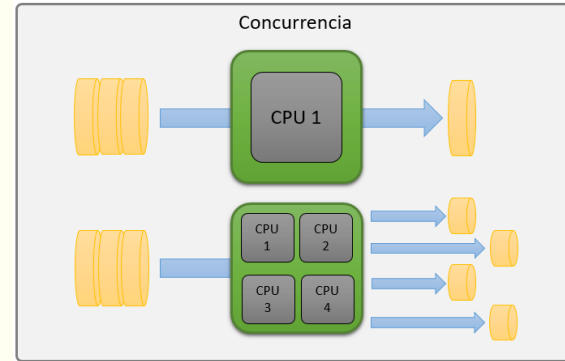


Hay un área de datos compartida entre un número de procesos. El área de datos puede ser un fichero, un bloque de memoria principal o incluso un banco de registros de procesador. Hay un número de procesos que solo leen el área de los datos (lectores) y otro número que solo escriben en el área de datos (escritores). Toma en cuenta las siguientes condiciones:

1. Cualquier número de lectores puede leer el fichero simultáneamente.
2. Solo un escritor al tiempo puede escribir un fichero.
3. Si un escritor está escribiendo ningún lector puede leerlo.

CAPITULO 6.

CONCURRENCIA. INTERBLOQUEO E INANICIÓN



6.1 Fundamentos del Interbloqueo



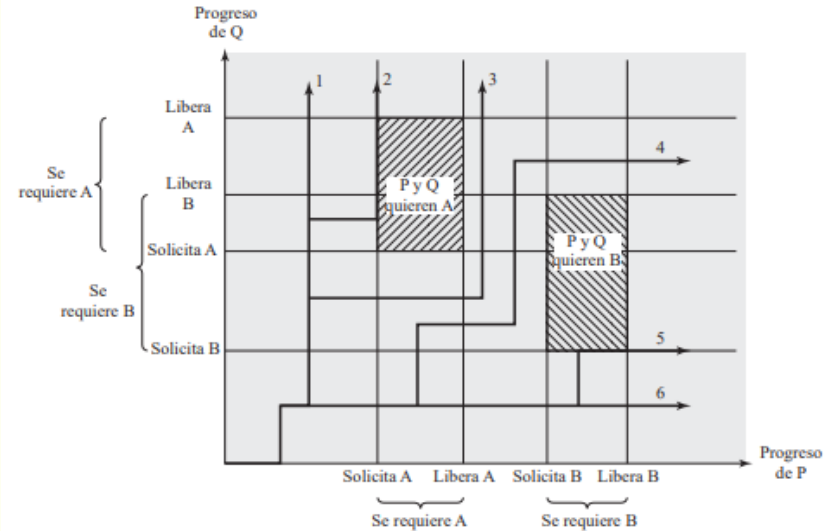
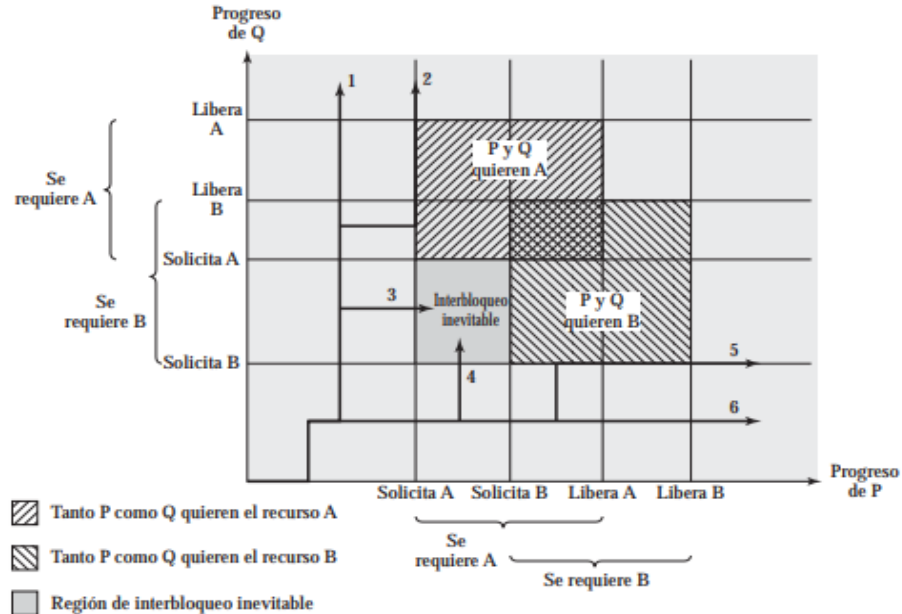
El interbloqueo es el bloqueo permanente de procesos que compiten por recursos del sistema.

Un conjunto de procesos se puede considerar interbloqueado cuando cada proceso espera un evento para ejecutarse, generalmente es liberación de recursos.

El interbloqueo se considera permanente cuando no pueden producirse ningún evento que se requiera.



EJEMPLOS DE INTERBLOQUEOS:



RECURSOS:

Los recursos se distinguen entre reutilizables y consumibles, el **reutilizables** es aquel que no se destruye luego de su uso, los interbloqueos de estos recursos suceden cuando procesos conservan el recurso y piden otro a la vez, como la memoria.

La forma de prevenir estos bloqueos es imponer orden en la solicitud de los recursos, un ejemplo de esto sería tener cierta memoria disponible y pedir de más sin liberar.

Por otro lado, los **consumibles** se usan y se destruyen, generalmente se pueden producir ilimitados, como la información de entrada y salida. Estos generalmente se interbloquean cuando un proceso pide algo que no existe.

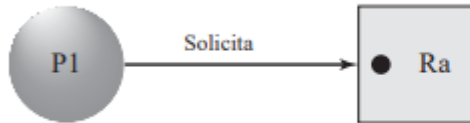




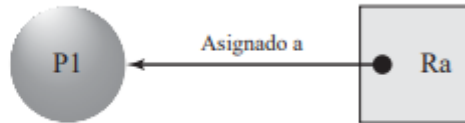
ESTRATEGIA	POLÍTICA DE RESERVA DE RECURSOS	ESQUEMA ALTERNATIVO	VENTAJAS	DESVENTAJAS
PREVENCIÓN	INFRAUTILIZAR RECURSOS	SOLICITUD SIMULTÁNEA	BUENA CON ACTIVIDADES ÚNICAS	RETRASA PROCESOS
		EXPROPIACIÓN	RESTAURA RECURSOS FÁCILMENTE	BLOQUEA MÁS DE LO NECESARIO
		ORDENAMIENTO	ES DE FÁCIL DISEÑO	IMPIDE SOLICITUDES GRADUALES
PREDICCIÓN	ENTRE DETECCION Y PREVENCIÓN	PREDICE UN CAMINO SEGURO	NO NECESITA EXPROPIAR	SE REQUIERE CONOCER LOS REQUISITOS FUTUROS
DETECCIÓN	TODO SE CONCEDE SI ES POSIBLE	SE UTILIZA PARA COMPROBAR INTERBLOQUEOS	NUNCA SE RETRASA	PÉRDIDAS INHERENTES POR EXPROPIACIÓN

GRAFOS DE ASIGNACIÓN DE RECURSOS

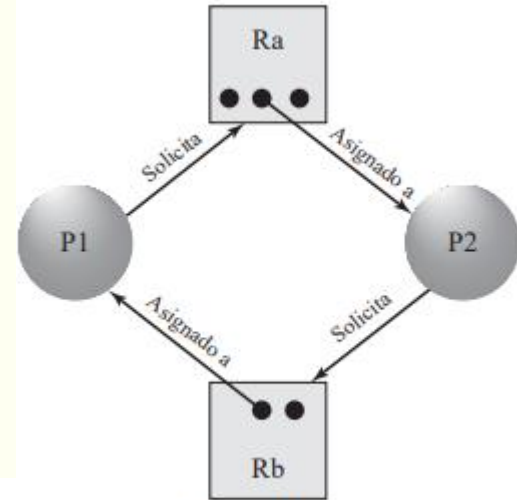
Es una herramienta útil para gestión de recursos, esto se define con nodos, que representan procesos y recursos, como también aristas que representan solicitudes o asignaciones, dentro de un nodo de recursos puede haber puntos, estos simbolizan instancias múltiples, cuando la arista sale de un punto es que esta instancia se asignó.



(a) Recurso solicitado



(b) Recurso asignado



(d) Sin interbloqueo

CONDICIONES DE INTERBLOQUEO



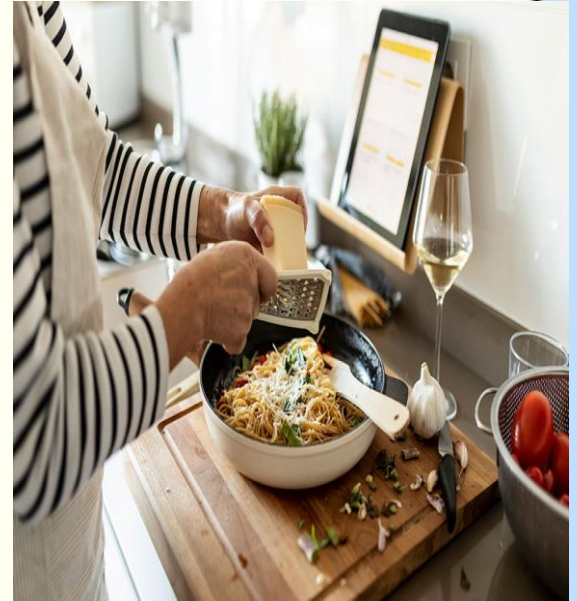
1. Exclusión mutua: Sólo un proceso puede usar un recurso a la vez.
2. Retención y espera: Un proceso mantiene sus recursos mientras espera la asignación del nuevo recurso.
3. Sin expropiación: No se puede forzar una prioridad de recursos.
4. Espera circular: Un proceso retiene un recurso del siguiente paso del ciclo de procesos.

Generalmente las 3 condiciones iniciales, pueden causar interbloqueos, la cuarta representa un resultado de las primeras, creando una esfera que no se puede romper.


6.2 PREVENCIÓN DEL INTERBLOQUEO

En resumen, es diseñar un sistema que excluya la posibilidad de interbloqueo, si no hay posibilidades, el interbloqueo simplemente no sucede (imposible), se listará como evitarlas.

1. Exclusión mutua: No es posible eliminarla, por ende, siempre existe únicamente está como factor inamovible.
2. Retención y espera: Si se programa el proceso para que en un inicio pida todo lo que necesita, evitas que posteriormente el proceso deba esperar.
3. Sin expropiación: Si a un proceso le deniegas una petición posterior, debe liberar sus recursos previos, sólo se puede aplicar a recursos reutilizables.
4. Esfera circular: Planificando bien un orden lineal, no suele ser eficiente.



6.3 Predicción del Interbloqueo



A diferencia de la prevención, en la predicción, las 3 condiciones de interbloqueo existen, pero se tienen planes razonables para no caer en el interbloqueo y aquí se requiere tener conocimiento de todo el proceso y lo que sucederá después, ir un paso adelante.

Las 2 técnicas para esto serían:

- Denegación de proceso: No hacer algo que pueda llevar a un interbloqueo.
- Denegación de recursos: No conceder una petición a un proceso que pueda bloquearse más adelante.

6.4 Detección del interbloqueo

Esta estrategia no limita el acceso a los recursos ni restringe las acciones de los procesos. Con la detección del interbloqueo, los recursos pedidos se conceden a los procesos siempre que sea posible.

```
struct estado
{
    int recursos[m];
    int disponibles[m];
    int necesidad[n][m];
    int asignacion[n][m];
}
```

(a) estructuras de datos globales

6.4 Detección del interbloqueo

```
if (asignacion [i,*] + peticion [*] > necesidad [i,*])  
    < error >;                                /* petición total > necesidad */  
else if (peticion [*] > disponibles [*])  
    < suspender al proceso >;  
else                                          /* simular asignación */  
{  
    < definir nuevo_estado como:  
    asignacion [i,*] = asignacion [i,*] + peticion [*];  
    disponibles [*] = disponibles [*] - peticion [*] >;  
}  
if (seguro(nuevo_estado))  
    < llevar a cabo la asignación >;  
else  
{  
    < restaurar el estado original >;  
    < suspender al proceso >;  
}
```

(b) algoritmo de asignación de recursos

Lógica para la predicción del interbloqueo

6.4 Detección del interbloqueo

```
boolean seguro (estado E)
{
    int disponibles_actual[m];
    proceso resto[<n mero de procesos>];
    disponibles_actual = disponibles;
    resto = {todos los procesos};
    posible = verdadero;
    while (posible)
    {
        <encontrar un proceso  $P_k$  en resto tal que
            necesidad  $[k,*]$  - asignacion  $[k,*]$  <= disponibles_actual;>
        if (encontrado)                                /* simular ejecución de  $P_k$  */
        {
            disponibles_actual = disponibles_actual + asignacion  $[k,*]$ ;
            resto = resto -  $\{P_k\}$ ;
        }
        else
            posible = falso;
    }
    return (resto == null);
}
```

(c) algoritmo para comprobar si el estado es seguro (algoritmo del banquero)

Lógica para la predicción del interbloqueo

6.4 Detección del interbloqueo



Algoritmo de detección del interbloqueo

Es una parte fundamental en el control de sistemas concurrentes para prevenir situaciones de interbloqueo, donde los procesos o hilos quedan atrapados en una espera mutua y no pueden avanzar. Tiene dos ventajas:

- Detección temprana de posibles errores
- El algoritmo es sencillo

Cabe mencionar que la gran desventaja de este algoritmo es que frecuentemente consume un considerable tiempo del procesador.

6.4 Detección del interbloqueo



Algoritmo de detección del interbloqueo

Se utilizan la matriz de Asignación y el vector de Disponible, además, se define una matriz de solicitud S tal que S_{ij} representa la cantidad de recursos de tipo j solicitados por el proceso i . El algoritmo actúa marcando los procesos que no están en un interbloqueo. Inicialmente, todos los procesos están sin marcar.

6.4 Detección del interbloqueo

Algoritmo de detección del interbloqueo

A continuación, se llevan a cabo los siguientes pasos:

1. Se marca cada proceso que tenga una fila de la matriz de Asignación completamente a cero.
2. Se inicia un vector temporal T asignándole el vector Disponibles.
3. Se busca un índice i tal que el proceso i no esté marcado actualmente y la fila i - de S sea menor o igual que T . Es decir, $S_{ik} \leq T_k$, para $1 \leq k \leq m$. Si no se encuentra ninguna fila, el algoritmo termina.
4. Si se encuentra una fila que lo cumpla, se marca el proceso i y se suma la fila correspondiente de la matriz de asignación a T . Es decir, se ejecuta $T_k = T_k + A_{ik}$, para $1 \leq k \leq m$. A continuación, se vuelve al tercer paso.

6.4 Detección del interbloqueo



Algoritmo de detección del interbloqueo

La estrategia de este algoritmo es encontrar un proceso cuyas peticiones de recursos puedan satisfacerse con los recursos disponibles, y, a continuación, asumir que se conceden estos recursos y el proceso se ejecuta hasta terminar y libera todos sus recursos.

6.4 Detección del interbloqueo

Algoritmo de detección del interbloqueo

Se puede utilizar la Figura 6.10 para mostrar el algoritmo de detección del interbloqueo. El algoritmo actúa de la siguiente manera:

1. Marca P4, porque no tiene recursos asignados.
2. Fija $T = (0\ 0\ 0\ 0\ 1)$.
3. La petición del proceso P3 es menor o igual que T, así que se marca P3 y ejecuta $T = T + (0\ 0\ 0\ 1\ 0) = (0\ 0\ 0\ 1\ 1)$.
4. Ningún otro proceso sin marcar tiene una fila de S que sea menor o igual a T. Por tanto, el algoritmo termina. El algoritmo concluye sin marcar P1 ni P2, indicando que estos procesos están en un interbloqueo.

6.4 Detección del interbloqueo

Algoritmo de detección del interbloqueo

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Matriz de solicitud S

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Matriz de asignación A

R1	R2	R3	R4	R5
2	1	1	2	1

Vector de recursos

R1	R2	R3	R4	R5
0	0	0	0	1

Vector de disponibles

Figura 6.10. Ejemplo de detección del interbloqueo.

6.4 Detección del interbloqueo



Recuperación

Ya que se detectó el interbloqueo, se requiere de un método para poder recuperarlo.

- Abortar todos los procesos involucrados en el interbloqueo. Esta es, se crea o no, una de las más usuales, si no la más, solución adoptada en los sistemas operativos.
- Retroceder cada proceso en interbloqueo a algún punto de control (checkpoint) previamente definido, y rearrancar todos los procesos. Esto requiere que se implementen en el sistema mecanismos de retroceso y re arranque.

6.4 Detección del interbloqueo



Recuperación

- Abortar sucesivamente los procesos en el interbloqueo hasta que este deje de existir. El orden en que seleccionan los procesos para abortarlos deber a estar basado en algunos criterios que impliquen un coste mínimo. Después de cada aborto, se debe invocar de nuevo el algoritmo de detección para comprobar si todavía existe el interbloqueo.
- Expropiar sucesivamente los recursos hasta que el interbloqueo deje de existir. Como en el tercer punto, se deber a utilizar una selección basada en el coste, y se requiere una nueva invocación del algoritmo de detección después de cada expropiación. Un proceso al que se le ha expropiado un recurso debe retroceder a un punto anterior a la adquisición de ese recurso.

6.5 Una estrategia integrada de tratamiento del interbloqueo



En las estrategias para el tratamiento de interbloqueo existen ventajas y desventajas y para evitar el uso de ambas, es mejor opción usar una estrategia según requiera la situación, por ejemplo:

- Agrupar los recursos en diversas clases de recursos diferentes.
- Utilizar la estrategia de la orden lineal definida previamente para prevenir la espera circular impidiendo los interbloqueos entre clases de recursos.
- Dentro de una clase de recursos, usar el algoritmo que sea más apropiado para esa clase.

6.5 Una estrategia integrada de tratamiento del interbloqueo



Como un ejemplo de esta técnica, considere las siguientes clases de recursos:

- Espacio de intercambio.
- Bloques de memoria en almacenamiento secundario utilizados al expulsar los procesos.
- Recursos del proceso.
- Dispositivos asignables, como dispositivos de cinta y ficheros.
- Memoria principal.
- Asignable a los procesos en páginas o segmentos.
- Recursos internos.
- Como canales de E/S.


6.5 Una estrategia integrada de tratamiento del interbloqueo



El orden de la lista anterior representa el orden en el que se asignan los recursos. Es un orden razonable, considerando la secuencia de pasos que puede seguir un proceso durante su tiempo de vida. Dentro de cada clase, se podrían utilizar las siguientes estrategias:

- Espacio de intercambio. La prevención de los interbloqueos, obligando a que se asignen al mismo tiempo todos los recursos necesarios que vayan a usarse, como en la estrategia de prevención de la retención y espera. Esta estrategia es razonable si se conocen los requisitos máximos de almacenamiento, lo cual frecuentemente es cierto. La predicción también es una posibilidad factible.

6.5 Una estrategia integrada de tratamiento del interbloqueo

- 
- Recursos del proceso. La predicción ser usualmente efectiva para esta categoría, porque es razonable esperar que los procesos declaren anticipadamente los recursos de esta clase que requerirán. La prevención mediante ordenamiento de recursos es también posible para esta clase.
 - Memoria principal. La prevención por expropiación parece la estrategia más apropiada para la memoria principal. Cuando se expropia a un proceso, simplemente es expulsado a memoria secundaria, liberando el espacio para resolver el interbloqueo.
 - Recursos internos. Puede utilizarse la prevención mediante el ordenamiento de recursos.

6.6 El problema de los filósofos comensales



Solución utilizando semáforos:

Presentando por Dijkstra, cinco filósofos viven en una casa, donde hay una mesa preparada para ellos. Básicamente, la vida de cada uno consiste en pensar y comer, después de años, cada filósofo estaba de acuerdo que la única comida ayudaba a su fuerza mental son los espaguetis. Debido a su falta de habilidad manual, cada filósofo necesita dos tenedores para comer.

El problema: Diseñar un algoritmo que permita a los filósofos comer. El algoritmo debe satisfacer la exclusión mutua (no puede haber dos filósofos que puedan utilizar el mismo tenedor a la vez).

Se podrían comprar cinco tenedores adicionales (una solución más higiénica) o enseñar a los filósofos a comer espaguetis con un solo tenedor. Como alternativa, se podría incorporar un asistente que sólo permitiera que haya cuatro filósofos al mismo tiempo en el comedor. Con un máximo de cuatro filósofos sentados, al menos un filósofo tendrá acceso a los dos tenedores.

6.6 El problema de los filósofos comensales

Solución utilizando un monitor:

Muestra una solución al problema de los filósofos comensales utilizando un monitor.

Se define un vector de cinco variables de condición, una variable de condición por cada tenedor. Estas variables de condición se utilizan para permitir que un filósofo espere hasta que esté disponible un tenedor. Además, hay un vector de tipo booleano que registra la disponibilidad de cada tenedor (verdadero significa que el tenedor está disponible). El monitor consta de dos procedimientos. El procedimiento obtiene tenedores lo utiliza un filósofo para obtener sus tenedores, el situado a su izquierda y a su derecha.

```
* programa filosofos_comensales */
semaforo tenedor (5) = (1);
int i;
void filosofo (int i)
{
    while (verdadero)
    {
        piensa 0);
        wait (tenedor[i]);
        wait (tenedor[(i+1) mod 5]);
        como (;
        signal (tenedor[(+1) mod 5]);
        signal (tenedor(i));
    }
}

void main ()
{
    paralelos (filosofo (0), filosofo (1), filosofo (2).
}
filosofo (3), filosofo (4));
```

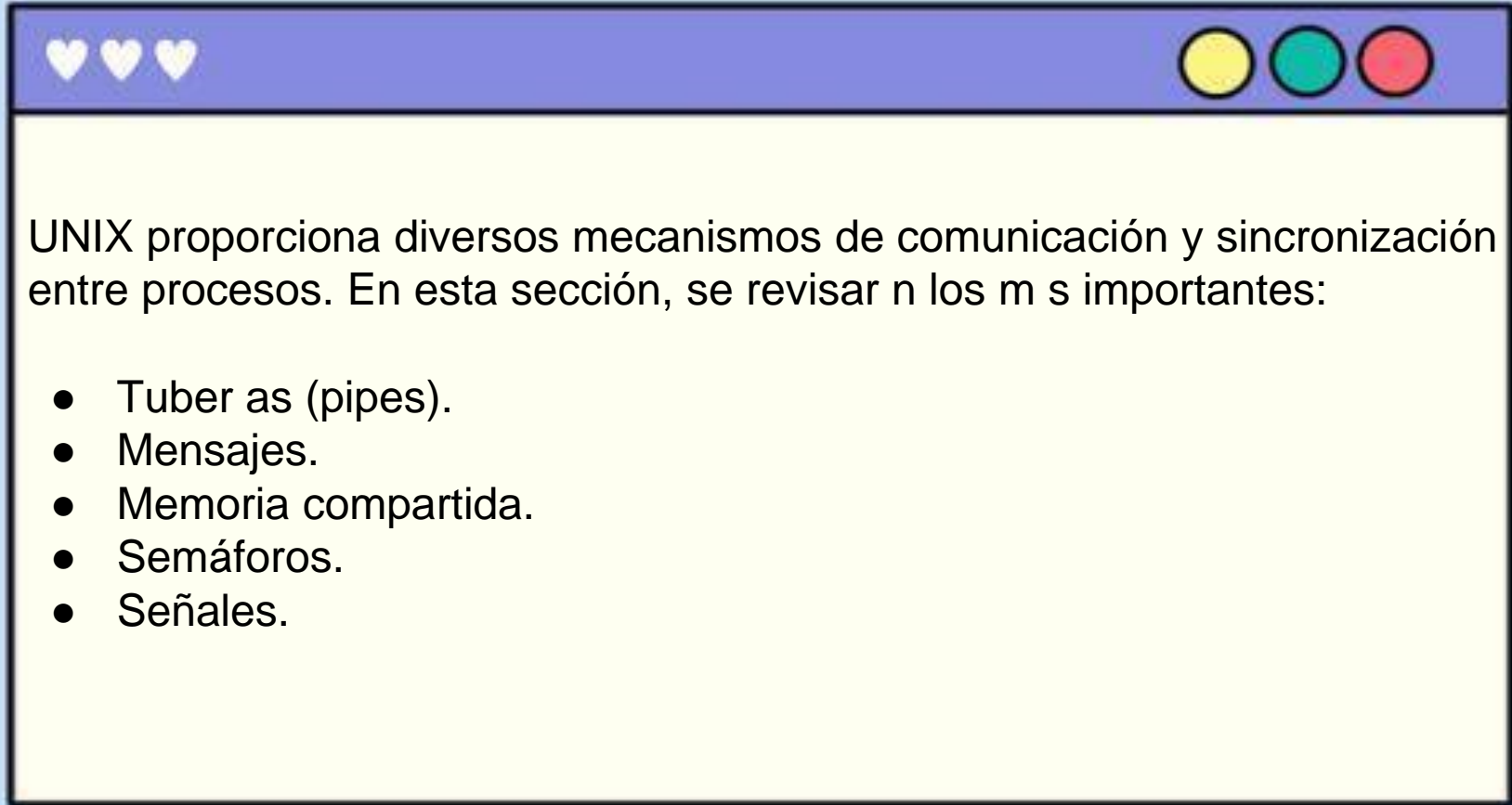
6.6 El problema de los filósofos comensales

CONTINUACION:

Si ambos tenedores están disponibles, el proceso que corresponde con el filósofo se encola en la variable de condición correspondiente. Esto permite que otro proceso filósofo entre en el monitor. Se utiliza el procedimiento libera tenedores para hacer que queden disponibles los dos tenedores. En ambos casos, un filósofo toma primero el tenedor de la izquierda y después el de la derecha. A diferencia de la solución del semáforo, esta solución del monitor no sufre interbloqueos, porque sólo puede haber un proceso en cada momento en el monitor.

```
* programa filosofos_comensales */
semaforo tenedor (5) = (1);
int i;
void filosofo (int i)
{
    while (verdadero)
    {
        piensa 0);
        wait (tenedor[i]);
        wait (tenedor[(i+1) mod 5]);
        como (;
        signal (tenedor[(+1) mod 5]);
        signal (tenedor(i));
    }
}
void main ()
{
    paralelos (filosofo (0), filosofo (1), filosofo (2).
}
filosofo (3), filosofo (4));
```

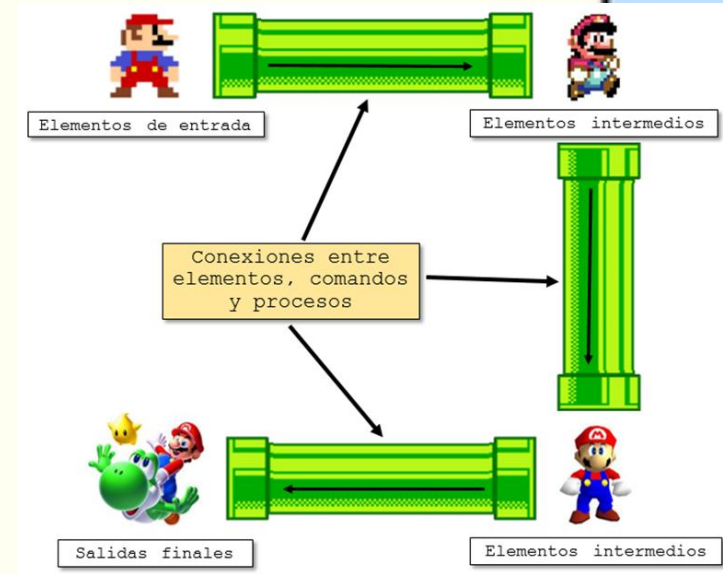
6.7 Mecanismos de concurrencia de UNIX



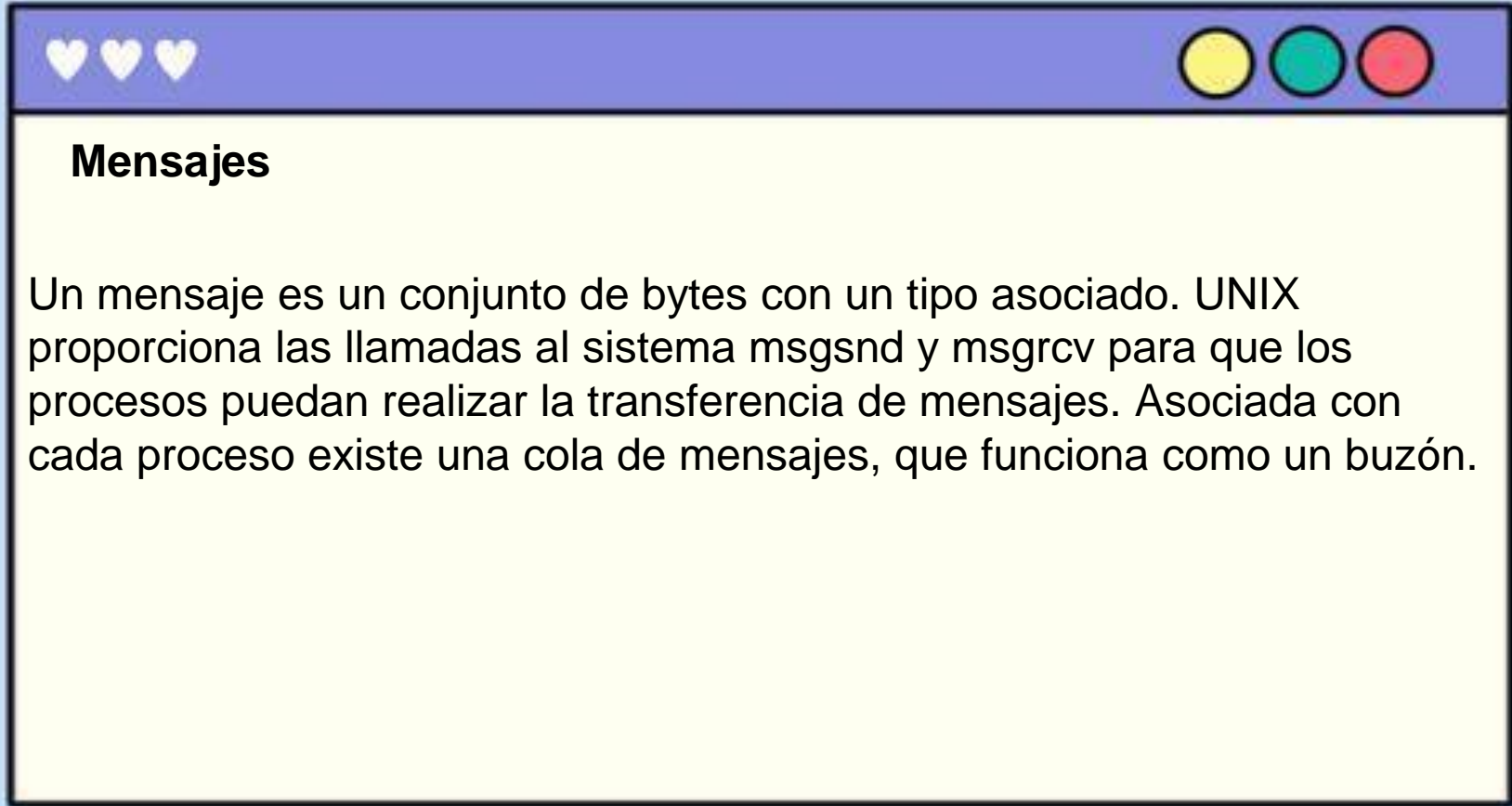
6.7 Mecanismos de concurrencia de UNIX

Tuberías

Una de las contribuciones más significativas de UNIX en el desarrollo de los sistemas operativos es la tubería. Inspirado por el concepto de corrutina [RITC84], una tubería es un buffer circular que permite que dos procesos se comuniquen siguiendo el modelo productor-consumidor. Por tanto, se trata de una cola de tipo el primero en entrar es el primero en salir, en la que escribe un proceso y lee otro.



Mecanismos de concurrencia de UNIX



Mecanismos de concurrencia de UNIX

Memoria compartida

Se trata de un bloque de memoria virtual compartido por múltiples procesos. Los procesos leen y escriben en la memoria compartida utilizando las mismas instrucciones de máquina que se utilizan para leer y escribir otras partes de su espacio de memoria virtual.



6.7 Mecanismos de concurrencia de UNIX



Semáforos

Son un mecanismo de concurrencia utilizado para coordinar y sincronizar el acceso a recursos compartidos entre procesos. Los semáforos pueden ser utilizados para evitar condiciones de carrera y garantizar que los procesos accedan a los recursos de manera ordenada y segura.

Mecanismos de concurrencia de UNIX



Semáforos

Un semáforo consta de los siguientes elementos:

- El valor actual del semáforo.
- El identificador del último proceso que operó con el semáforo.
- El número de procesos en espera de que el valor del semáforo sea mayor que su valor actual.
- El número de procesos en espera de que el valor del semáforo sea cero.
- Asociado con el semáforo están las colas de los procesos bloqueados en ese semáforo.

Mecanismos de concurrencia de UNIX



Señales

Es un mecanismo software que informa a un proceso de la existencia de eventos asíncronos. Una señal es similar a las interrupciones hardware, pero no emplea prioridades.

Es decir, todas las señales se tratan por igual; las señales que ocurren al mismo tiempo se le presentan al proceso una detrás de otra, sin ningún orden en particular.

Mecanismos de concurrencia de UNIX

Señales

Valor	Nombre	Descripción
01	SIGHUP	Desconexión; enviada al proceso cuando el núcleo asume que el usuario de ese proceso no está haciendo trabajo útil
02	SIGINT	Interrupción
03	SIGQUIT	Abandonar; enviada por el usuario para provocar la parada del proceso y la generación de un volcado de su memoria (<i>core dump</i>)
04	SIGILL	Instrucción ilegal
05	SIGTRAP	<i>Trap</i> de traza; activa la ejecución de código para realizar un seguimiento de la ejecución del proceso
06	SIGIOT	Instrucción IOT
07	SIGEMT	Instrucción EMT
08	SIGFPE	Excepción de coma flotante
09	SIGKILL	Matar; terminar el proceso
10	SIGBUS	Error de bus
11	SIGSEGV	Violación de segmento; el proceso intenta acceder a una posición fuera de su espacio de direcciones
12	SIGSYS	Argumento erróneo en una llamada al sistema
13	SIGPIPE	Escritura sobre una tubería que no tiene lectores asociados
14	SIGALRM	Alarma; emitida cuando un proceso desea recibir una señal cuando transcurra un determinado periodo de tiempo
15	SIGTERM	Terminación por software
16	SIGUSR1	Señal 1 definida por el usuario
17	SIGUSR2	Señal 2 definida por el usuario
18	SIGCHLD	Muerte de un proceso hijo
19	SIGPWR	Interrupción en el suministro de energía

Mecanismos de concurrencia del núcleo de Linux.

Operaciones atómicas

Las operaciones atómicas son secuencias que garantizan accesos atómicos de una sola palabra compartida.

Esto significa que las operaciones atómicas no pueden proteger los accesos a estructuras de datos, así que los bloqueos pueden proporcionar una forma muy eficiente de dar acceso a una sola palabra.



Mecanismos de concurrencia del núcleo de Linux.

Tipos de operaciones atómicas

Con enteros:

Las operaciones atómicas con enteros pueden utilizar solamente este tipo de datos, no permitiéndose ninguna otra operación sobre el mismo.

Con mapa de bits:

Las operaciones atómicas con mapas de bits operan sobre una secuencia de bits almacenada en una posición de memoria indicada por una variable de tipo puntero.



Mecanismos de concurrencia de Windows



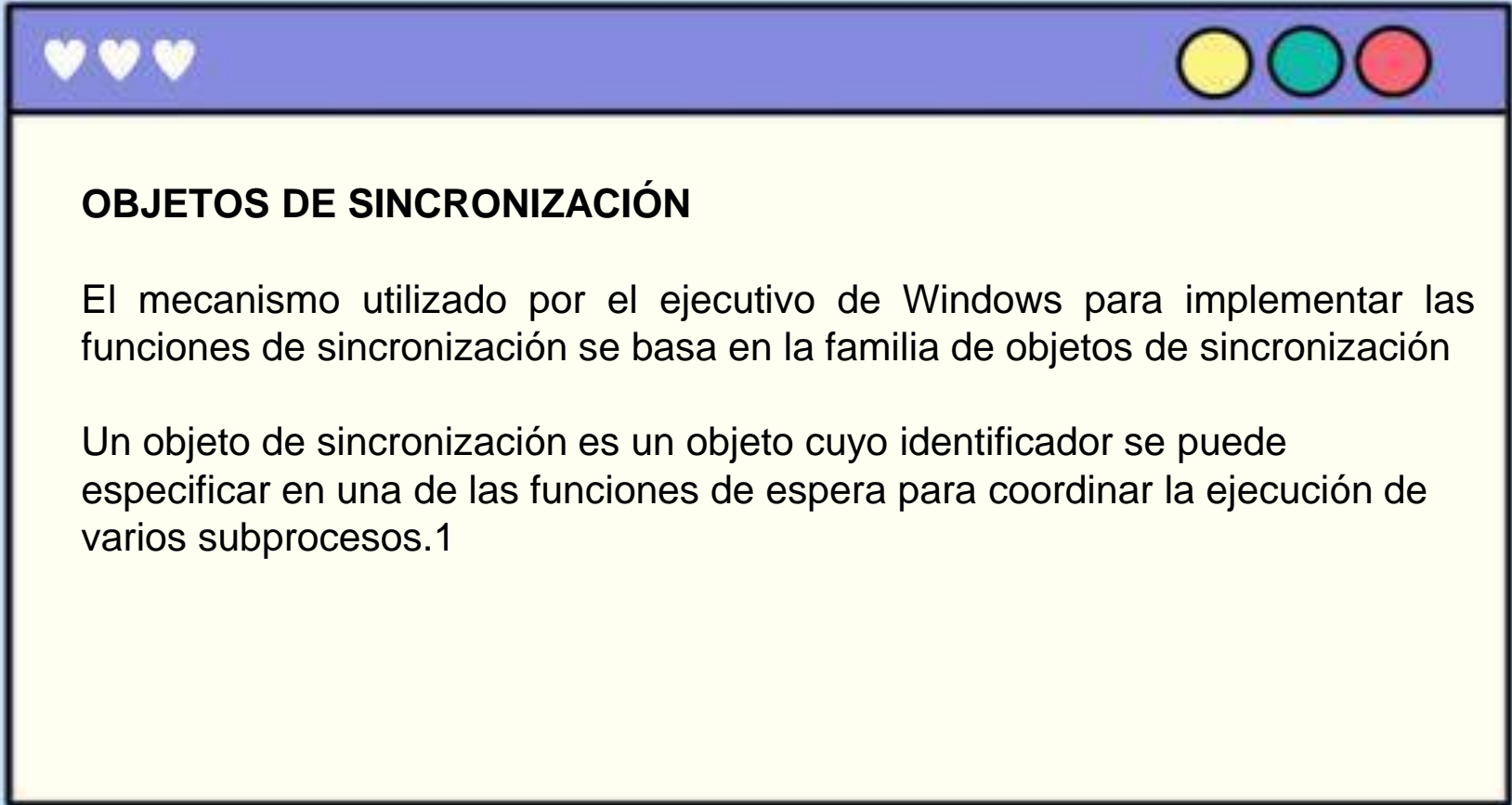
Funciones de espera

Las funciones de espera permiten que un hilo bloquee su propia ejecución. Las funciones de espera no retornan hasta que se cumplen los criterios especificados.

El tipo de función de espera determina el conjunto de criterios utilizado.



Mecanismos de concurrencia de Windows



Mecanismos de concurrencia de Windows

