Sistemas Operativos **II**

Módulo I

CONCURRENCIA

1

Temas del Módulo 1

- **Principios de la Concurrencia de Procesos.**
  - Comunicación entre procesos.
  - Competencia por compartimiento de recursos (sección crítica).
  - Sincronización entre procesos.
- **Problema de la sección crítica.**
  - Soluciones por Hardware.
  - Soluciones por Software.
- **Semáforos.**
  - Una solución al problema de la sección crítica.
  - Sincronización de procesos con semáforos.
- **Sincronización usando mensajes.**
- **Llamadas a procedimientos Remotos (RPC).**

2

Introducción a la Concurrencia

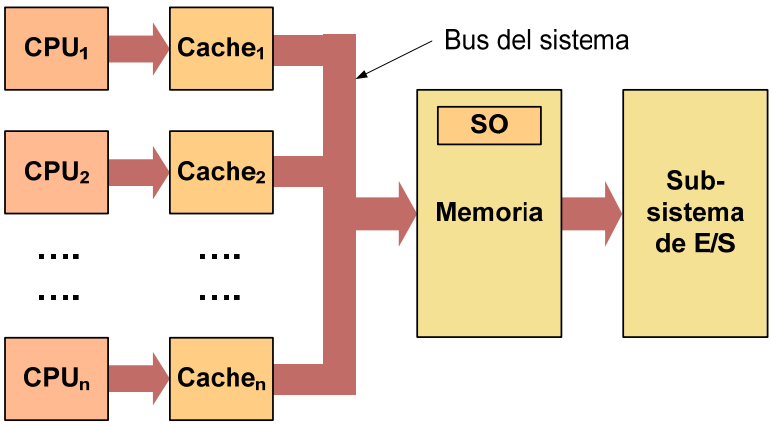
- **¿Qué significa el término *concurrencia* en SO's?**
  Se refiere a la ejecución "al mismo tiempo" de dos o más procesos o hilos.
  La concurrencia es un concepto clave a partir del cual se construyen los SO modernos.
- **¿Dónde ocurre la concurrencia de procesos e hilos? ¿En qué ámbitos o entornos?**
  - **Multiprogramación:** múltiples procesos e hilos ejecutan en un sistema monoprocesador.
  - **Multiprocesamiento:** múltiples procesos e hilos ejecutan en un sistema multiprocesador.
  - **Procesamiento Distribuido:** múltiples procesos e hilos ejecutan en varios nodos (computadores) interconectados por una red de comunicación.

3

Introducción a la Concurrencia

Ejemplo 1: Sistema Multiprocesador (SMP)

Memoria y SO únicos, compartidos por varios procesadores que están sobre un mismo bus del sistema

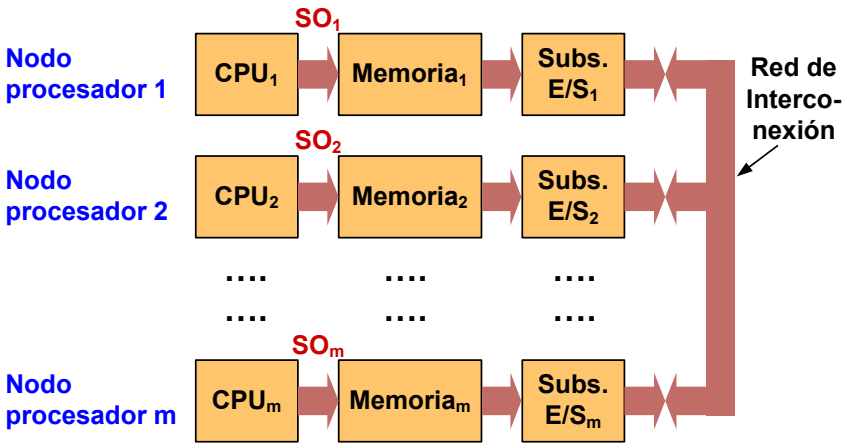


4

Introducción a la Concurrencia

Ejemplo 2: Sistema Multiprocesador (Multicomputador)

Varios nodos procesadores con interconexión de alta velocidad.
Cada nodo (CPU + Mem + Subs. E/S) ejecuta su propio SO

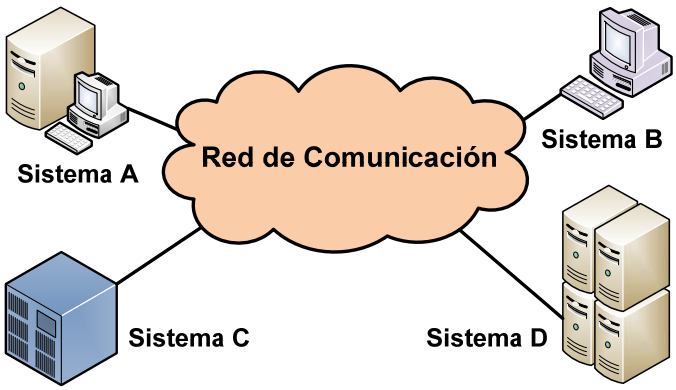


5

Introducción a la Concurrencia

Ejemplo 3: Procesamiento Distribuido

Varios sistemas diferentes en Hw y SO, comunicados por una red de área extensa o interred (por ejemplo, Internet)



6

Introducción a la Concurrencia

Síntesis de los Sistemas Multiprocesadores

Sistemas Multiprocesadores

1. **Varios procesadores con única memoria y bus del sistema.**
   (Plataforma multiprocesador)

2. **Varios nodos con fuerte conexión**
   nodo = procesador + memoria + sist. de E/S
   (Plataforma multinodo o multicomputador)

3. **Varios computadores completos interconectados por una red o interred con conexión débil: red o interred**
   (Sistema Distribuido)

7

Introducción a la Concurrencia

- **¿Qué implica la Concurrencia?**
  - Existe concurrencia en un sistema cuando múltiples procesos o hilos ejecutan en cualquiera de los entornos mencionados anteriormente.
  - Además, esta ejecución concurrente implica interacción entre los procesos, que debe ser controlada por el SO.
  - Esta interacción supone que los recursos del sistema son compartidos entre los procesos, lo cual puede generar complicaciones.
- **Tipos de interacción entre procesos.**
  - **Compartimiento de (o competencia por) recursos** (El SO debe proveer a los procesos un acceso ordenado a los recursos de uso exclusivo).
  - **Comunicación entre procesos**.
  - **Sincronización de actividades de múltiples procesos**.
  - **Planificación de procesos** (El SO debe administrar el tiempo del procesador que asignará a los procesos).

8

Introducción a la Concurrencia

- **¿Cuándo se manifiesta la concurrencia?**

Puede aparecer en 3 contextos diferentes:

- **Ejecución de Múltiples aplicaciones:**
  - Multiprogramación: varias aplicaciones activas comparten dinámicamente el tiempo de procesamiento.
- **Programación de Aplicaciones estructuradas:**
  - Diseño modular y programación estructurada: se logra mayor eficiencia cuando las aplicaciones se programan como un conjunto de procesos concurrentes (paralelismo).
- **Diseño de la Estructura del SO:**
  - El SO es un conjunto de procesos o hilos que se diseña aplicando las mismas ventajas usadas en las aplicaciones estructuradas.

9

Introducción a la Concurrencia

Conclusión sobre Entornos de la Concurrencia

La concurrencia de procesos o hilos está presente tanto:

1. En sistemas multiprocesador y distribuidos. Aquí hay concurrencia **física** o **simultánea** de varios procesos: cada proceso es independiente y ejecuta en un procesador diferente.
2. En sistemas monoprocesador con multiprogramación. Aquí la concurrencia de procesos es **lógica** o **virtual**: varios procesos comparten dinámicamente el tiempo de un único procesador.

A continuación se analiza la interacción entre procesos e hilos cuando ejecutan concurrentemente.

10

Interacción entre Procesos

Competencia y Cooperación entre Procesos.

Durante su vida en el sistema, los procesos (del SO y de los usuarios) interactúan entre sí compartiendo recursos y, también, compitiendo por el uso de los mismos.

1. Competencia

- Cuando se diseña un SO, un problema importante a resolver es cuando dos o más procesos compiten por el uso de un recurso que no puede ser compartido en forma simultánea (este tipo de recurso se denomina recurso de uso exclusivo).

  Ejemplo: dos procesos necesitan modificar el mismo registro de una base de datos.En este caso, el registro es un recurso de uso exclusivo.

- Procesos que intervienen en este caso: procesos competitivos.

11

Interacción entre Procesos

Competencia y Cooperación entre Procesos. *(cont.)*

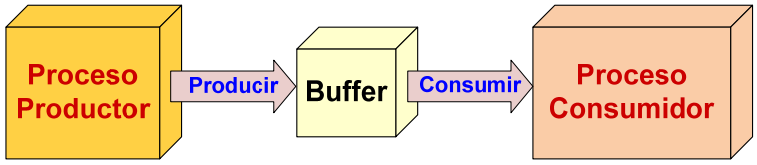
2. Cooperación

- Situación diferente a la competencia: aparece cuando dos o más procesos cooperan para resolver un mismo problema.
- En este caso cada proceso *conoce* la existencia de los otros y por lo tanto se debe sincronizar su ejecución con la del resto de los procesos con los cuales coopera.
- Procesos que intervienen en este tipo de tareas: procesos cooperativos.
- Ejemplo de cooperación: procesos del tipo *productor/consumidor.* (ver figura siguiente).

12

Interacción entre Procesos

Ejemplo de procesos cooperativos:



Proceso productor: genera datos que son depositados en el buffer.

Proceso consumidor: extrae (consume) los datos del buffer.

Este tipo de tareas aparecen en diversas situaciones en los SO´s:

- Ej 1: *Driver* de la impresora deposita datos en el buffer (productor) Proceso del módulo de E/S los retira (consumidor) p/imprimirlos.
- Ej 2: Un proceso *compilador* produce un código en assembler que es consumido por el proceso *ensamblador*, el que, a su vez, genera el código de máquina que es consumido por el proceso *loader,* quien carga dicho programa en la memoria principal del computador.

13

Interacción entre Procesos

Competencia y Cooperación entre Procesos. *(cont.)*

2. Cooperación *(cont.)*

Importante: Cuando ejecutan procesos cooperativos, el SO debe proveer **sincronización** entre los mismos.

- Necesidad de sincronización: se debe a que los procesos productor y consumidor se ejecutan en forma concurrente y, posiblemente, a diferentes velocidades (Ej. de la impresora: driver ejecuta en CPU central, y proceso de escritura ejecuta en el procesador del módulo de E/S).
- Modo en que el SO provee sincronización: usa primitivas de sincronización que, por ejemplo, evitan que el proceso productor sobreescriba datos en el buffer antes que el proceso consumidor los haya retirado. (ejemplo del productor/consumidor).

14

Interacción entre Procesos

Clasificación de Procesos según el tipo de Interacción.

Según haya competencia o cooperación los procesos pueden ser:

1. Procesos independientes entre sí.
2. Procesos indirectamente dependientes entre sí.
3. Procesos directamente dependientes entre sí.

15

Interacción entre Procesos

Clasificación de Procesos según el tipo de Interacción.

1. **Procesos independientes entre sí:** no conocen la existencia de otros procesos.

   Situación típica que se presenta en sistemas multiprogramados: procesos independientes que intentan acceder a recursos de uso exclusivo. Ej.: una rutina de E/S intenta acceder a un disco, a una impresora (alocamiento), o a escribir en un registro de archivo. (es independiente del número de procesadores que contenga el sistema).

- Tipo de interacción: **competencia** (por el uso de un recurso).
- El SO debe proveer los medios para que el acceso de los procesos a tales recursos sea de manera secuencial (no simultánea).

Conclusión: se debe proveer **exclusión mutua** entre procesos.

16

Interacción entre Procesos

Clasificación de Procesos según el tipo de Interacción. *(cont)*

2. Procesos indirectamente dependientes entre sí: no conocen la existencia de otros procesos en forma explícita, pero comparten algún objeto en común.

Ejemplo: el *driver* de una impresora comparte el buffer con el proceso de escritura del módulo de E/S.

- Tipo de interacción: **cooperación** (por compartir un recurso, p/ej, un buffer).
- El sistema debe proveer: **exclusión mutua** y **sincronización** entre los procesos cooperativos.

17

Interacción entre Procesos

Clasificación de Procesos según el tipo de Interacción. *(cont)*

3. Procesos directamente dependientes entre sí: son capaces de comunicarse directamente entre sí por su identificación (ID), trabajando en forma conjunta y cooperándose mutuamente en alguna actividad.

- La comunicación se realiza mediante el envío de mensajes.
- Las primitivas de envío y recepción de mensajes deben ser proporcionadas por el lenguaje de programación o por el kernel del SO.
- Tipo de interacción: **cooperación** (mensaje entre procesos).
- El SO debe proveer: **sincronización** entre los procesos.

18

Interacción entre Procesos

Clasificación de Procesos según el tipo de Interacción. *(cont)*

3. Procesos directamente dependientes entre sí. *(cont)*

Importante: aquí los procesos no comparten ni compiten por ningún recurso (sólo se envían mensajes), por lo tanto no es necesario proveer exclusión mutua entre ellos.
Sí puede haber interbloqueo e inanición.

A continuación se muestran dos tablas que resumen lo expuesto sobre Clasificación de Procesos según el tipo de Interacción.

19

Interacción entre Procesos

Grado de dependencia	Relación	Influencia de un proceso sobre otro
1. Procesos **independientes**	**Competencia** por **uso exclusivo** de un recurso	Los resultados de un proceso son **independientes** de la acción de otros. La velocidad de ejecución puede verse afectada.
2. Procesos **indirectamente dependientes**	**Cooperación** por **compartimiento** de un recurso	Los resultados de un proceso **pueden depender** de la información provista por otros procesos. La velocidad de ejecución puede verse afectada.
3. Procesos **directamente dependientes**	**Cooperación** por **comunicación** entre procesos	Los resultados de un proceso **pueden depender** de la información provista por otros procesos. La velocidad de ejecución puede verse afectada.

20

Interacción entre Procesos

Grado de dependencia	Relación	Problemas y soluciones
1. Procesos **independientes**	**Competencia** por **uso exclusivo** de un recurso	Debe proveerse a los procesos **exclusión mutua** evitando **interbloqueo, inanición** y **corrupción de los datos**
2. Procesos **indirectamente dependientes**	**Cooperación** por **compartimiento** de un recurso	Debe proveerse a los procesos **exclusión mutua** y **sincronización** evitando **interbloqueo, inanición** y **corrupción de los datos**
3. Procesos **directamente dependientes**	**Cooperación** por **comunicación** entre procesos	Debe proveerse a los procesos **sincronización** evitando **interbloqueo** e **inanición**

21

Interacción entre Procesos

Clasificación de Procesos según el tipo de Interacción. *(cont)*

Potenciales problemas derivados de la concurrencia de procesos:

- **Inanición de procesos:** una parte fue tratado en SO-I. (planificación de la CPU). Otra parte será tratado en Módulo1.
- **Exclusión Mutua:** será tratado en el presente Módulo1.
- **Interbloqueo:** se estudiará en el Módulo2.

22

Problema de la Sección Crítica

Exclusión Mutua.

- Se denomina así a la solución que se debe proveer para evitar que dos o más procesos accedan simultáneamente a un recurso compartido de uso exclusivo.
- Exclusión mutua: implica que los procesos deben tener la capacidad de autoexcluirse entre sí en el uso de un recurso.
- Para proveer exclusión mutua se debe crear una **sección critica** en los procesos intervinientes.

23

Problema de la Sección Crítica

Introducción.

- Necesidad de crear la sección crítica: nace de la competencia de los procesos por el acceso a **recursos compartidos** por ellos.
  - Cuál es el problema?: posible corrupción de datos cuando varios procesos pueden aleatoria y libremente modificar el contenido de un recurso que es compartido entre ellos.
  - Ante esto, el SO debe proporcionar mecanismos para proteger al recurso del acceso simultáneo de dos o más procesos.
- **Recursos compartidos de uso exclusivo**: se denominan así los recursos a los que puede usar un solo proceso por vez.
- Pueden ser: lógicos (software) o físicos (hardware):
  - Recursos exclusivos lógicos: archivo, estructura de datos, variable compartida.
  - Recursos exclusivos físicos: disco, impresora, placa de red.

24

Problema de la Sección Crítica

Temas de la Sección Crítica:

- Un ejemplo sobre el problema de la sección crítica.
- Análisis del problema en un escenario general.
- Condiciones que debe cumplir la solución al problema.
- Solución por software.
- Solución por hardware.
- Solución con semáforos. Operaciones P y V.

25

Problema de la Sección Crítica

Ejemplo sobre el problema de la sección crítica:

Supóngase que 2 procesos P0 y P1 están ejecutando concurrentemente y cada uno debe incrementar una variable ***x*** compartida[*]. ***x*** puede ser, p/ej., el nº de archivos abiertos, o el nº de procesos terminados del sistema, etc.

P0: . . ***x = x + 1***

P1: . . ***x = x +1***

Estas instrucciones en lenguaje de alto nivel son traducidas en varias instrucciones en código de máquina. (ver diap. siguiente)

---

[*]No confundir variable compartida con variable global.
Var. compartida: puede ser accedida por todos los proc´s. que la referencian.
Var. global: es accedida sólo por los procedimientos o rutinas de un proceso).
Una variable compartida es definida en un espacio de direcciones "universal".
Una variable global es definida solo en el espacio de direcciones de un proceso

26

Problema de la Sección Crítica

Ejemplo sobre el problema de la sección crítica. *(cont.)*

- Para simplificar el ejemplo dado considérese un computador con 2 procesadores centrales C0 y C1, cada uno con sus respectivos registros internos R0 y R1.
- Si C0 procesa a P0 y C1 procesa a P1 se podrían tener los siguientes casos de secuencia de ejecución de estos procesos en lenguaje de máquina: (en diapositiva siguiente)

27

Problema de la Sección Crítica

Ejemplo sobre el problema de la sección crítica. *(cont.)*

Caso 1:

P0:	P1:
R0 = x	…
R0 = R0 + 1	…
x = R0	…
	R1 = x
	R1 = R1 + 1
	x = R1

R0 = 3 R1 = 4
x = 4

28

Problema de la Sección Crítica

Ejemplo sobre el problema de la sección crítica. *(cont.)*

Caso 2:

P0:	P1:
R0 = x	…
R0 = R0 + 1	R1 = x
x = R0	R1 = R1 + 1
	x = R1

R0 = 3 R1 = 3
x = 3

29

Problema de la Sección Crítica

Ejemplo sobre el problema de la sección crítica *(cont.)*

Análisis del ejemplo:

- En Caso1 y Caso2 se obtienen resultados diferentes en el valor de ***x*** que **dependen del instante** en que comenzaron a ejecutarse P0 y P1, lo que es totalmente inaceptable.
- Para asegurar la obtención de la solución correcta (***x*** = 4) debe permitirse que **sólo un proceso por vez** incremente la variable ***x.***
- Esto implica que la sentencia ***x = x +1*** es una **sección crítica** del programa, debiéndose evitar que dos o más procesos ingresen simultáneamente a esa sección crítica.

  Dicho de otra forma: si la sentencia ***x = x + 1*** es una sección crítica, sólo se debería permitir el ingreso de los procesos en forma mutuamente excluyente o secuencial a la misma para evitar la corrupción de datos.

30

Problema de la Sección Crítica

Escenario General del problema de la Sección Crítica

Para introducir el escenario general, se considerará el caso de los procesos secuenciales cíclicos: aquellos que se comunican entre sí a través de una o más variables compartidas, en un ciclo repetitivo "infinito".

Para este tipo de procesos es necesario que: si cada proceso tiene en su código una sección crítica en la cual accede a un dado recurso compartido de uso exclusivo es necesario:

- Lograr que en cualquier momento **sólo uno** de los procesos se encuentre procesando en su sección crítica.
- Es lo mismo que decir: una vez que un dado proceso se encuentra ejecutando en su sección crítica, ningún otro proceso puede ejecutar en su correspondiente sección crítica (para que esto ocurra debe estar implementada la exclusión mutua entre procesos).

31

Problema de la Sección Crítica

Modelo para analizar el problema de la Sección Crítica

Planteo general: existen ***n*** procesos (**P0, P1, … Pn**) los cuales pueden acceder a un mismo recurso compartido. El esquema es el siguiente:

P0:	**P1:**	…	**Pn:**
`while(true)`	`while(true)`		`while(true)`
`{`	`{`		`{`
`< SC0 >`	`< SC1 >`	…	`< SCn >`
$programa_1$	$programa_2$		$programa_n$
`}`	`}`		`}`

Por ejemplo: supóngase que **P0, P1, … Pn** son ***n*** procesos que pueden modificar registros de una base de datos. Si **P2** se encuentra en su **SC2**, ningún otro proceso puede estar en su correspondiente **SCi** hasta tanto **P2** no haya salido de su **SC2**.

32

Problema de la Sección Crítica

Requisitos de la solución para garantizar la exclusión mutua de varios procesos respecto de un recurso compartido:

no solamente debería cumplir con la condición de acceso exclusivo de uno de los procesos a su SC, sino con otras condiciones que impidan, por ejemplo:

- **Inanición** de uno o más procesos.
- **Bloqueo indefinido** de uno o más procesos.

En resumen: la solución que asegure la exclusión mutua entre procesos, **no debe generar** inanición o bloqueo indefinido en otros procesos. Para lograr esto:

Son 4 las condiciones que debe cumplir la solución propuesta para el acceso de los procesos a un recurso compartido de uso exclusivo. (a continuación)

33

Problema de la Sección Crítica

Condiciones a cumplir por la Solución de la Sección Crítica

1. Los procesos P_0, P_1, …P_n no deben estar **simultáneamente** en sus secciones críticas (debe proveerse **exclusión mutua**).
2. Un proceso que está fuera de su SC - ni siquiera intentando ingresar a su SC[*] - **no debe evitar** que otro proceso entre a su SC (puede generar **bloqueo indefinido** en el 2º proceso).
3. No debe ocurrir que un proceso ingrese a su SC en repetidas oportunidades y que otro proceso **nunca** tenga la posibilidad de ingresar a su SC (puede generar **inanición**).
4. Dos o más procesos que están por ingresar a sus respectivas SC´s no deben entrar en un lazo de espera infinito (si así sucediera, se puede generar **bloqueo indefinido**).

[*] Se dice que un proceso no está, *ni siquiera intentando ingresar a su SC*, cuando no está ejecutando el código que precede a la SC y que ha sido incluido en el programa para resolver el problema de la SC.

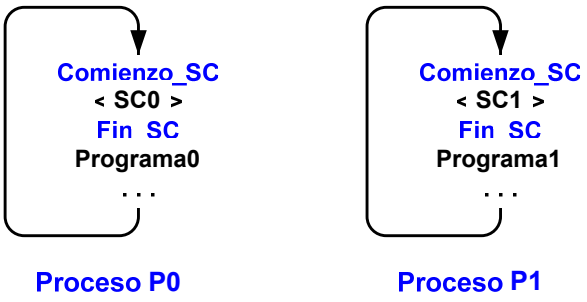
34

Soluciones al Problema de la Sección Crítica

Solución por Software:

Solución para el caso de 2 procesos respecto de un recurso.

Objetivo: En base al diagrama esquemático de los procesos mostrado en la figura, la solución implica diseñar el código **Comienzo_SC** y **Fin_SC** tal que cumpla con las 4 condiciones enunciadas anteriormente.



35

Soluciones al Problema de la Sección Crítica

Solución por Software - Propuesta por Peterson

- La primera solución al problema de la sección crítica utilizando herramientas de software fue propuesta por Dekker. Si bien su solución cumple las 4 condiciones, el algoritmo es poco claro.
- Luego Peterson en 1981 propuso una solución que es mucho más simple y elegante. Se muestra en la diap. siguiente.

36

Soluciones al Problema de la Sección Crítica

Solución por Software - Propuesta por Peterson

```
boolean estado[2] = {false, false}; // Variables compartidas.
int turno;          // Variable compartida.

void P0()
{
  while(true)
  {
    estado[0] = true;
    turno = 1;
    while(estado[1] && turno==1)
        /*no hacer nada*/;
    /*SECC. CRÍTICA 0*/;
    estado[0] = false;
    /*programa 0*/;
  }
}
```

```
void P1()
{
  while(true)
  {
    estado[1] = true;
    turno = 0;
    while(estado[0] && turno==0)
        /*no hacer nada*/;
    /*SECC. CRÍTICA 1*/;
    estado[1] = false;
    /*programa 1*/;
  }
}
```

37

Soluciones al Problema de la Sección Crítica

Solución por Software - Propuesta por Peterson *(cont.)*

Prueba de que la solución de Peterson cumple con las 4 condiciones establecidas para la solución de la SC.

A. Prueba de las condiciones 2, 3 y 4: (contemplan los casos de bloqueos mutuos entre procesos).
Si existiese bloqueo mutuo, por lo menos uno de los procesos debería permanecer indefinidamente en su lazo de espera.

Sin embargo, esto no es posible debido a que si P0 está en lazo de espera indefinidamente, P1 puede estar en cualquiera de los siguientes lugares:

1. P1 no intentando ingresar a su SC1.
2. P1 esperando en su propio lazo de espera.
3. P1 ejecutando repetidamente su lazo completo (`while(true)`).

Los 3 casos se analizan a continuación.

38

Soluciones al Problema de la Sección Crítica

Solución por Software - Propuesta por Peterson *(cont.)*

A. Prueba de las condiciones 2, 3 y 4 *(cont.)*

1. P1 no intentando ingresar a su SC1: P0 detecta que `estado[1]` es `false` y por lo tanto la condición de su lazo de espera no se cumple; entonces P0 puede entrar a su SC0.
2. P1 esperando en su propio lazo de espera: es imposible que se presente ya que `turno` es 0 o 1, por lo tanto en uno de los procesos la condición de lazo de espera se evaluará a `false` y, por lo tanto, P1 podrá ingresar a su SC1.
3. P1 repetidamente ejecutando su lazo completo (`while(true)`): también es imposible que se presente, ya que P1 pone a `turno = 0` y, por ende, no puede volver a entrar en su SC1 hasta que P0 ingrese a su SC0.

39

Soluciones al Problema de la Sección Crítica

Solución por Software - Propuesta por Peterson *(cont.)*

B. Prueba de garantía de la exclusión mutua

Se supone P0 está en su SC0. En tal caso `estado[0]` debe ser `true`.

Se verá si P1 puede, de alguna forma, entrar a su SC1.
Hay 2 casos para considerar:

1. P0 entró a su SC0 porque `estado[1]` era `false`: Esto indica que P1 no está intentando entrar en su SC1. Por lo tanto, si desea entrar, debe poner primeramente `turno = 0` y como `estado[0]` es `true`, P1 cumple el test del lazo de espera y, por lo tanto, no podrá entrar en su SC1.
2. P0 entró a su SC0 porque turno era 0: Esto implica que, independientemente de dónde se encuentra P1 ejecutando, encontrará `estado[0] = true` y `turno = 0;` por lo tanto, no podrá ingresar a su SC1.

40

Soluciones al Problema de la Sección Crítica

Solución por Hardware

Existen diversas soluciones. Aquí se van a considerar solo dos.

1. **Método más simple:** proveer exclusión mutua inhabilitando las IRQ al procesador cuando un proceso se encuentra en su SC. (usado por UNIX en las 1ras. versiones).

   El esquema es el siguiente:

```
Pi:
    inhabilitar IRQ's
    < SCi >
    habilitar IRQ's
```

- Ventaja: simplicidad y posibilidad de resolución para todos los procesos.
- Desventajas: **1.** No es aplicable en sistemas multiprocesador. Sería necesario inhabilitar las IRQ para todos los CPU's, lo que bajaría la eficiencia del procesamiento.

41

Soluciones al Problema de la Sección Crítica

Solución por Hardware *(cont.)*

1. **Método más simple** *(cont.)*:

   Desventajas *(cont.)*: **2.** No es aplicable a SO's de tiempo real donde las IRQ's son un recurso esencial para las tareas de ejecución crítica.

2. **Solución que usa la función test&set**

   Algunos procesadores tienen instrucciones atómicas. Esto significa que:

- El chequeo y modificación del contenido de una variable es realizada en forma **indivisible.**
- Si una IRQ se presenta durante la ejecución de una instrucción atómica, ésta es atendida sólo cuando finaliza el procesamiento de dicha instrucción.

42

Soluciones al Problema de la Sección Crítica

Solución por Hardware *(cont.)*

2. Solución que usa la función test&set *(cont.)*

Una implementación puede ser la siguiente:

```
boolean testset(int i)
{
    if(i==0)
    {
        i = 1;
        return true;
    }
    else
    {
        return false;
    }
}
```

43

Soluciones al Problema de la Sección Crítica

Solución por Hardware *(cont.)*

2. Solución que usa la función test&set *(cont.)*

- `testset` comprueba el valor de `i`. Si es 0, entonces cambia el valor a 1 y devuelve `true`. En caso contrario, el valor no se cambia y devuelve `false`.
- `testset` es ejecutada atómicamente, es decir, no está sujeta a interrupción.
- Entonces, un protocolo para implementar la exclusión mutua con esta instrucción será el siguiente:

44

Soluciones al Problema de la Sección Crítica

Solución por Hardware *(cont.)*

2. Solución que usa la función test&set *(cont.)*

```
int cerrojo = 0;   // Inicializa variable compartida.

void P()
{
    while(true)
    {
        while(!testset(cerrojo))
            /*no hacer nada*/;
        /*SECCIÓN CRÍTICA*/;
        cerrojo = 0;
        /*programa*/;
    }
}
```

45

Soluciones al Problema de la Sección Crítica

Solución por Hardware *(cont.)*

Ventajas:

- Se aplica a cualquier número de procesos tanto en sistemas monoprocesador como en sistemas multiprocesador.
- Es simple y, por lo tanto, muy fácil de verificar.
- Puede ser utilizada para soportar múltiples secciones críticas.

Desventajas:

- En su implementación, normalmente se emplean lazos de espera donde la CPU utiliza ciclos sin computación útil.
- En ciertas circunstancias es posible que provoque inanición de procesos.
- Es posible que aparezca bloqueo indefinido (“deadlock”), dependiendo del tipo de planificador de CPU que utilice el SO.

46

Soluciones al Problema de la Sección Crítica

Solución por Hardware *(cont.)*

Debido a las desventajas expuestas, las soluciones por hardware **no son** generalmente empleadas en los SO modernos.

47

Primitivas de Semáforo

Espera Ocupada.

- Técnica donde un proceso repetidamente verifica una condición, tal como esperar una entrada de teclado o si el ingreso a una sección crítica está habilitado.
- Puede ser una estrategia válida en algunas circunstancias especiales, sobre todo en la sincronización de procesos.
- En general, debe ser evitada, ya que consume tiempo de CPU sin realizar ninguna operación.

48

Primitivas de Semáforo

Operaciones P y V *(cont.)*

Dijkstra introdujo en 1965 una solución basada en dos primitivas que:

- Simplificó considerablemente la comunicación, la sincronización entre procesos y el problema de la sección crítica.
- Solucionó el inconveniente mencionado en el punto 2 anterior. Se debe aclarar, no obstante, que la eficacia de la solución que se va a estudiar a continuación depende de cómo se implemente.

49

Primitivas de Semáforo

Operaciones P y V *(cont.)*

En su forma abstracta estas primitivas, denominadas **P** y **V**, operan con variables enteras no negativas llamadas **semáforos.**

Definición: Sea **s** una variable tipo semáforo.

Las operaciones **P** y **V** se definen como:

V(s): Incremente **s** en 1 en una operación atómica;
es decir, **V(s): s** = **s** + 1 ← operación atómica

P(s): Decremente **s** en 1, si es posible.
Si un proceso invoca **P(s)** con **s** = 0:

- la operación no puede ser realizada ya que por definición **s** pertenece al rango de los enteros nonegativos.
- por lo tanto **debe esperar** hasta que sea posible ejecutar el decremento. **P(s)** es también una operación atómica.

50

Primitivas de Semáforo

Operaciones P y V *(cont.)*

Condiciones para completar la definición de **P** y **V**:

- Si varios procesos invocan simultáneamente operaciones **P** y **V** sobre la misma variable semáforo, estas operaciones se ejecutarán **secuencialmente** en un orden arbitrario.

  La ejecución secuencial se explica debido a la atomicidad de las primitivas.
- Si más de un proceso se encuentra esperando para ejecutar una operación **P** y el semáforo en cuestión pasa a ser positivo (obviamente porque se ejecutó una operación **V**), el proceso que pasará a reanudar su procesamiento se selecciona de un modo arbitrario.

51

Primitivas de Semáforo

Operaciones P y V *(cont.)*

De las definiciones anteriores se desprende que:

- La operación **P** puede considerarse como una primitiva que conlleva una potencial espera del proceso que la invoca.
- Por el contrario, la primitiva **V** puede considerarse como la operación que activa algún proceso que estaba en espera. (de allí que a la operación **P** también se la denomina ***wait()*** y a la operación **V**, ***signal().*** (**P** "espera" una "señal" de **V**).

Con la definición de la variable tipo semáforo, y las operaciones que actúan sobre ella:

- es posible solucionar el problema de la sección crítica.
- también es posible sincronizar procesos, que es otro de los objetivos de la concurrencia de procesos expuesta en la introducción de este módulo.

52

Primitivas de Semáforo

Solución al Problema de la Sección Crítica con Semáforos

Las operaciones **P** y **V** proveen una solución simple y directa al problema de la sección crítica.

Presentación de la solución:

Sea **s** una variable tipo semáforo que será utilizada por los procesos para proteger sus respectivas SC's.

La solución del problema de la SC para **n** procesos sería:
(sigue en la próxima diapositiva)

53

Primitivas de Semáforo

Solución al Problema de la Sección Crítica con Semáforos

```
semaforo s = 1; // Definición e inicialización del semáforo.

void P0()                          ...   void Pn()
{                                        {
  while(true)                              while(true)
  {                                        {
     …                                        …
    P(s);                                    P(s);
    /*SECC. CRÍTICA 0*/;                     /*SECC. CRÍTICA n*/;
    V(s);                                    V(s);
     …                                        …
  }                                        }
}                                        }
```

54

Primitivas de Semáforo

Análisis del pseudocódigo anterior. Obsérvese que:

- Cuando cualquier proceso está en su SC, el valor del semáforo **s** es 0; en caso contrario es 1.
- Hay exclusión mutua porque una vez que un proceso ejecuta a **P**, sólo él puede decrementar el valor de **s** a cero (por ser **P(s)** definida como atómica).
- Estando un proceso en su SC, todos los demás procesos que intenten entrar en sus SC no lo podrán hacer, teniendo que esperar para poder hacerlo (por definición de **P(s)**).
- Se elimina la posibilidad de inanición y bloqueo mutuo[*] entre procesos, ya que intentos simultáneos de ingresar a sus respectivas SC cuando **s** = 1, se traducen en operaciones **P** secuenciales. (por ser **P** una operación atómica).

[*] Condiciones 2 a 4, de las 4 condiciones enunciadas en diap. 34, que deben ser cumplidas por cualquier solución del problema de la SC.

55

Implementación de Operaciones con Semáforos (P y V)

Implementación con Espera Ocupada.

- Implementaremos las primitivas *P* y *V* utilizando un nuevo tipo de semáforos: **semáforos binarios**, es decir, que sólo pueden tomar valores 0 (*false*) ó 1 (*true*).
- Para ello vamos a definir sendas operaciones restringidas, $\mathbf{P_b}$ y $\mathbf{V_b}$, y un nuevo tipo de semáforo, $\mathbf{s_b}$.

Definición de P_b y V_b:

- **P_b :** si s_b es *false*, espera hasta que sea *true*; cuando es *true* (y por lo tanto se desbloquee) cambia el valor de s_b a *false*.
- **V_b :** simplemente, s_b = *true.*

Ambas instrucciones se deben ejecutar en forma atómica.

56

Implementación de Operaciones con Semáforos (P y V)

Implementación de Pb y Vb

$P_b(s_b)$: `while not test&set(sb) do nop` *// lazo de espera.*

$V_b(s_b)$: `sb = true`

- La instrucción que implementa V_b es en sí atómica en todos los procesadores; siempre se ejecuta como una unidad completa, aún si existe una interrupción en el medio de su ejecución.
- Para la operación P_b, cuando el semáforo s_b tenga el valor *false*, un proceso que ejecute P_b entrará en el lazo de espera.

57

Implementación de Operaciones con Semáforos (P y V)

Semáforos *mutex* y *delay*.

- Para implementar las operaciones generales **P** y **V** sobre un semáforo **s** podemos utilizar las operaciones restringidas a semáforos binarios. Para ello utilizaremos dos semáforos binarios: *mutex* y *delay*.
- El propósito de *mutex* es solamente implementar la exclusión mutua o atomicidad de las instrucciones **P** y **V**.
- Por su parte el semáforo *delay* es donde quedarán en espera los procesos que invoquen **P** cuando el valor de **s** sea menor que cero.

58

Implementación de Operaciones con Semáforos (P y V)

```
semaforo mutex = true;  //semáforo binario inicializado en true.
semaforo delay = false; //semáforo binario inicializado en false.

Operación P(s):
              Pb(mutex);
              s = s – 1;
              if s < 0 then
                     Vb(mutex);
                     Pb(delay);
              Vb(mutex);


Operación V(s):
              Pb(mutex);
              s = s + 1;
              if s <= 0 then
                     Vb(delay);
              else
                     Vb(mutex);
```

59

Implementación de Operaciones con Semáforos (P y V)

Observaciones sobre la implementación.

- Con esta implementación se pueden llegar a tener valores negativos de *s*, lo cual contradice la definición inicial de semáforo como una variable *no-negativa*.
- Sin embargo, esta definición resulta útil, ya que el semáforo negativo representa el número de procesos que están en espera sobre ese semáforo.
- El semáforo binario *delay* se utiliza únicamente para la sincronización de los procesos; es sobre éste semáforo donde los procesos son realmente puestos en espera cuando s es menor o igual que cero.
- En *P(s)* se utilizan dos operaciones $V_b(mutex)$, se cumpla o no la condición del *if* … **PORQUÉ???**

60

Implementación de Operaciones con Semáforos (P y V)

Evitando la Espera Ocupada.

- La espera ocupada de la sección crítica no disminuye la performance cuando las secciones críticas son breves, sin embargo lo ideal es evitar el lazo de espera que consume ciclos de CPU (*do nop*).
- Se propone entonces el siguiente esquema:
  - El proceso que intente acceder a su SC, estando accedida por otro, se bloquea a sí mismo, permitiendo el uso de CPU a otro proceso.
  - El proceso bloqueado será reactivado posteriormente por el proceso que terminó de utilizar la SC.
- De esta manera se puede evitar el uso del semáforo *delay*, ya que sobre él es donde se ejecuta la espera ocupada que consume la mayor cantidad de ciclos de CPU*.

*La operación **Pb** sobre mutex sólo consume algunos ciclos de CPU ya que la sección crítica que protege (acceso a **s**) es de unas cuantas instrucciones.

61

Implementación de Operaciones con Semáforos (P y V)

- La implementación de las operaciones **P(s)** y **V(s)** contemplarán el uso de una cola de procesos bloqueados en el semáforo **s**. Éstos serán extraídos mediante una operación **V**, lo que cambiará el estado del proceso a *ready* ubicándolo en la cola correspondiente.
- A grandes rasgos tendrán la forma:

```
P(s):  s = s - 1;
       if s < 0 //cambio de proceso.
          insertar proceso en cola de procesos bloqueados en s;
          bloquear proceso;
       end


V(s):  s = s + 1;
       if s <= 0
          sacar un proceso de cola de procesos bloqueados en s;
          desbloquear el proceso; //cambiar el estado.
       end
```

62

Implementación de Operaciones con Semáforos (P y V)

Semáforo como estructura de datos.

- Se puede considerar ahora a cada semáforo como una estructura de datos con la forma:

```
struct semaforo
{
 int value;   //valor propiamente dicho del semáforo.
 lista L;     //lista de procesos asociada con el semáforo s.
};
```

- Cada semáforo tendrá un valor entero y una cola de procesos, con un puntero* a los procesos que están bloqueados en dicho semáforo.

*En realidad el puntero apunta a los PCB de los procesos.

63

Implementación de Operaciones con Semáforos (P y V)

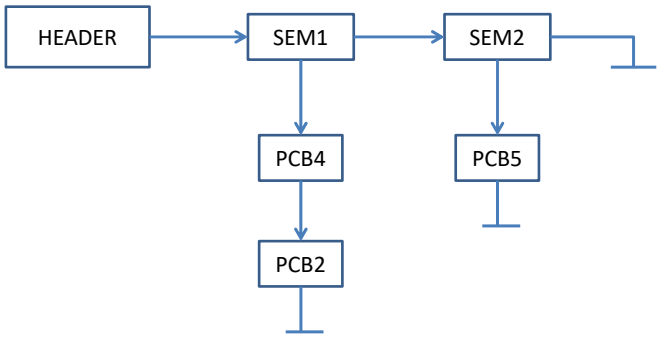


Diagrama esquemático de la Lista de Semáforos Activos.

64

Implementación de Operaciones con Semáforos (P y V)

La implementación de P(s) y V(s) será entonces:

```
struct semaforo{
       int valor;
       lista L;
       }s;

semaforobin mutex = true;//semáforo binario para exclusión mutua
struct PCB *q;      //puntero a una estructura de tipo proceso.

P(s):  Pb(mutex);
       s.valor = s.valor – 1;
       if s.valor < 0 then        //cambio de proceso.
          marcar estado del proceso como bloqueado;
          q = delete(Cola_Ready); //saca un proc. de cola ready.
          Vb(mutex);
          transferir control al proceso q;
       else
          Vb(mutex);
```

65

Implementación de Operaciones con Semáforos (P y V)

```
V(s):  Pb(mutex);
       s.valor = s.valor + 1;
       if s.valor <=0 then //activar un proceso.
          q = delete(s.L); //sacar proceso de cola asociada a s.
          if (hay CPU libre) then //apropiado para SMP.
             comenzar a ejecutar q en un CPU libre;
          else
             insert(q, Cola_Ready); //si CPU's ocupados, a cola
                                    //ready.
       Vb(mutex);
```

66

Implementación de Operaciones con Semáforos (P y V)

Análisis de la implementación.

- Estas implementaciones de **P(s)** y **V(s)** son bastante generales, contemplando incluso su uso en sistemas con multiprocesadores.
- Las funciones delete() e insert() extraen e insertan respectivamente un proceso en las colas que se especifican como argumento.
- Finalmente, no se han eliminado por completo los lazos de espera ocupados, ya que se utiliza la operación $\mathbf{P_b}$. Sin embargo, esta espera es muy breve.

67

Primitivas de Semáforo

Sincronización de Procesos con Semáforos.

Se vio que en los SO's se puede modelar una gran diversidad de interacciones entre procesos mediante procesos del tipo productor/consumidor, en donde el consumidor utiliza (consume) cierta cantidad de recursos, y el productor crea (produce) otra cantidad de recursos.

Uso de semáforos en procesos del tipo productor/consumidor.

En este caso los semáforos proveen un método eficiente para:

- mantener un contador de los recursos utilizados por los diversos procesos y, además
- permitir sincronizar los procesos que utilizan dichos recursos.

68

Primitivas de Semáforo

Sincronización de Procesos con Semáforos. *(cont.)*

El problema del Buffer Limitado, introducido por Dijkstra en 1968 muestra cómo se pueden utilizar semáforos para sincronizar procesos cooperativos.

(Este sería el caso de procesos indirectamente dependientes, de acuerdo a la clasificación presentada).

Problema del Buffer Limitado.

Se lo va a estudiar en 3 etapas:

1. Planteo del problema.
2. Posible implementación de los procesos productor/consumidor.
3. Análisis de la implementación.

69

Primitivas de Semáforo

Problema del Buffer Limitado.

1. Planteo del problema.

Se trata de una situación de procesos productor/consumidor:

- Un proceso productor genera información que es puesta en un buffer de almacenamiento.
- Concurrentemente, un proceso consumidor extrae información de este buffer (es consumida mediante procesamiento).

70

Primitivas de Semáforo

Problema del Buffer Limitado. *(cont.)*

1. Planteo del problema. *(cont.)*

Se considera que el buffer está compuesto por un número limitado de ***n*** celdas, donde cada celda puede almacenar un registro de datos producido por el proceso productor.

La solución al problema consiste en coordinar los procesos productor y consumidor de tal forma que:

- el buffer nunca sea saturado (rebalse) por el proceso productor,
- el proceso consumidor debe esperar en caso de que el buffer se encuentre vacío.

71

Primitivas de Semáforo

Problema del Buffer Limitado. *(cont.)*

1. Planteo del problema. *(cont.)*

En la solución del problema se emplearán los siguientes semáforos como contadores de números de registros del buffer:

e = número de celdas vacías.

f = número de celdas ocupadas o llenas.

- El semáforo ***e*** representa celdas disponibles para ser llenadas por el productor.
- El semáforo ***f*** representa celdas disponibles para ser consumidas por el consumidor.

La figura siguiente muestra esquemáticamente el buffer compartido y las variables semáforo definidas.

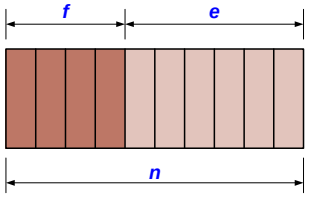
72

Primitivas de Semáforo

Problema del Buffer Limitado. *(cont.)*

1. Planteo del problema. *(cont.)*

Diagrama esquemático del buffer.



Obsérvese que el buffer es un recurso compartido de uso exclusivo durante las siguientes situaciones:

- agregado de datos a una celda por parte del productor,
- extracción de datos de una celda por parte del consumidor.

Por lo tanto, ambas operaciones deben ser protegidas mediante la implementación de las secciones críticas correspondientes.

73

Primitivas de Semáforo

Problema del Buffer Limitado. *(cont.)*

2. Posible implementación de los procesos productor/consumidor

```
semaforo e = n ; f = 0 ;   // todas las celdas están vacías.
semaforo s = 1 ;   // utilizado para exclusión mutua.
void Productor()
{
   while(true)
   {
      /*Producir el próximo registro*/;
      P(e);  // decrementa celdas vacías, o espera si no hay vacía.
      P(s);  // decrementa s / se está por ingresar a la sección crítica.
      /*AGREGAR REGISTRO AL BUFFER*/;
      V(s);  // se salió de la sección crítica.
      V(f);  // incrementa celdas llenas / despierta consumidor.
             // consumidor en espera por celda llena.
   }
}
```

74

Primitivas de Semáforo

Problema del Buffer Limitado. *(cont.)*

2. Posible implementación de los procesos productor/consumidor *(cont.)*

```
void Consumidor()
{
   while(true)
   {
      P(f); // decrementa valor de celdas llenas, o espera si no
            // hay celda llena.
      P(s); // se está por ingresar a la sección crítica.
      /*EXTRAER REGISTRO DEL BUFFER*/;
      V(s); // se salió de la sección crítica.
      V(e); // incrementa # celdas vacías y despierta productor.
            // productor en espera por celda vacía.
      /*Procesar registro extraído*/;
   }
}
```

75

Primitivas de Semáforo

Problema del Buffer Limitado. *(cont.)*

3. Análisis de la implementación propuesta.

- Los semáforos ***e*** y ***f*** han sido utilizados para sincronización. (Cuando incrementa ***f***, despierta al consumidor; cuando se incrementa ***e***, despierta al productor).
- ***e*** (número de celdas vacías) debe ser considerado por el productor ya que si no existen celdas vacías (***e*** = 0), éste no puede depositar el registro generado; por lo tanto, debe esperar hasta que el consumidor libere una de las celdas.
- ***f*** (número de celdas llenas) debe ser considerado por el consumidor, puesto que si no hay celdas llenas debe esperar a que el productor agregue datos al menos a una celda.
- Es evidente que las operaciones de incremento y decremento de estas variables deben ser atómicas, lo cual justifica aún más el hecho de que estas variables sean del tipo semáforo.

76

Primitivas de Semáforo

Problema del Buffer Limitado. *(cont.)*

3. Análisis de la implementación propuesta. *(cont.)*

- La variable semáforo ***s*** es utilizada solamente para garantizar acceso exclusivo al buffer, es decir, para proteger la SC.

  Es importante notar que:

  - ***s*** garantiza exclusión mutua de los procesos en la manipulación de los punteros que apuntan a la próxima celda vacía o llena del buffer, pero no al acceso a la celda.
  - El **acceso a la celda** propiamente dicho es permitido o inhibido por la sincronización de los procesos que es implementada por P(e) y V(f) en productor, y P(f) y V(e) en consumidor.

77

Conclusiones sobre las herramientas vistas

Todas las herramientas vistas hasta aquí para proveer a los procesos:

- Exclusión mutua.
- Sincronización.
- Comunicación.

sólo son aplicables a sistemas sobre arquitecturas de hardware que tienen **memoria compartida**, es decir, una única memoria principal, pudiendo poseer uno o más procesadores cuya característica principal es que:

- están sobre un único bus que conecta a la memoria única
- ejecutan una única copia de sistema operativo que está en memoria.

78

Entornos de ejecución concurrente

Existen arquitecturas con **memoria no compartida** que se pueden clasificar en dos tipos:

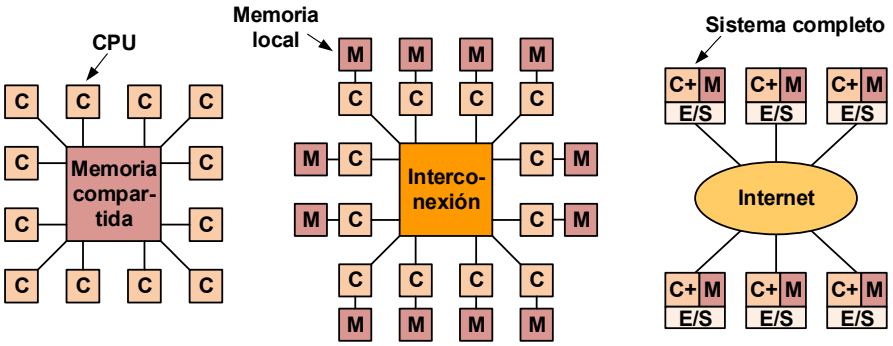
- Sistemas Multicomputadores o Clusters (**MC**).
- Sistemas Distribuidos (**SD**).

(Ver diapositiva siguiente)

79

Entornos de ejecución concurrente

Sistemas de Múltiples Procesadores



(a) **Multiprocesador con Memoria compartida**

(b) **Multicomputador con paso de mensajes**

(c) **Sistema distribuido de área extensa**

80

Sincronización en MC y SD usando Mensajes

Características operativas de los procesos en MC y SD.

- Los procesos sólo acceden a la memoria propia de cada nodo, pues no hay memoria física que sea compartida entre nodos.
- Aclaración para provisión de exclusión mutua:

  Valen los mecanismos vistos antes, solo para procesos que están en un mismo nodo, compartiendo recursos de ese nodo y usando variables compartidas **sólo en ése ámbito**.

Sincronización y comunicación entre procesos de nodos diferentes:

- **No son válidos** los procedimientos basados en semáforos.
- Es necesario usar **mensajes** entre procesos en nodos distintos.

Mecanismos más usuales para generar mensajes:

- Primitivas ***send*** y ***receive***.
- **Llamada a Procedimiento Remoto (RPC: Remote Proc. Call)**.

81

Sincronización en MC y SD usando Mensajes

Primitivas *send* y *receive*.

Se utilizan para enviar y recibir datos de cualquier tipo entre los procesos para proveer sincronización y comunicación.

En forma genérica, la sintaxis de estas primitivas es:

- ***send (p, msg)***
- ***receive (q, msg)***

send: se usa para enviar un mensaje.

- ***p:*** proceso destinatario del mensaje.
- ***msg:*** puede ser una estructura de datos específica, o un lugar de la memoria en dónde se almacenará el mensaje.

receive: se usa para recibir un mensaje.

- ***q:*** proceso emisor del cual se espera recibir el mensaje.
- ***msg:*** puede ser una estructura de datos específica o un lugar de la memoria adonde se depositará el mensaje.

82

Sincronización en MC y SD usando Mensajes

send	Bloqueante	No bloqueante
Nombramiento **Explícito**	Enviar el mensaje ***msg*** al receptor ***p*** y esperar hasta que ***msg*** sea aceptado	Enviar el mensaje ***msg*** al receptor ***p*** y continuar ejecutando
Nombramiento **Implícito**	Enviar ***msg*** a todos los procesos y esperar hasta que ***msg*** sea aceptado	Enviar ***msg*** a todos los procesos y continuar ejecutando
receive	**Bloqueante**	**No bloqueante**
Nombramiento **Explícito**	Esperar por mensaje ***msg*** del proceso emisor ***q***	Si hay mensaje ***msg*** del emisor ***q***, recibirlo; sino continuar ejecutando
Nombramiento **Implícito**	Esperar por mensaje ***msg*** de cualquier emisor	Si hay mensaje ***msg*** de cualquier emisor, recibirlo, sino continuar

83

Sincronización en MC y SD usando Mensajes

Evaluación de las primitivas *"send"* y *"receive".*

Ventaja:

- Son dos de las tantas primitivas utilizadas en la sincronización de procesos en sistemas que no tienen memoria compartida.

Desventaja:

- Son de bajo nivel; esto significa que el programador se ve forzado a intercalar en el medio de sus procedimientos de alto nivel las primitivas ***send*** y ***receive***.

Existen otras primitivas de alto nivel que son utilizadas para la sincronización de procesos que no necesitan compartir variables y no tienen la desventaja de las primitivas *send* y *receive.*

Estas primitivas se basan en invocaciones a procedimientos remotos llamadas comúnmente **RPC** (**R**emote **P**rocedure **C**all).

84

Llamada a Procedimiento Remoto (RPC)

- Las **RPC** se utilizan para la **sincronización** de procesos en sistemas MC y SD. (Es una variante de las primitivas de pasaje de mensajes).

Esencia del esquema basado en RPC.

- Las **RPC** permiten hacer interactuar a procesos de **distintos computadores** utilizando la **misma semántica** usada para llamadas a procedimientos locales.
- Es decir, actúan como si los programas que interactúan estuvieran en la misma máquina, con la salvedad de que no están.
- Los esquemas de sincronización basados en **RPC** son usados en la mayoría de los SO's (Ej: Windows NT, UNIX, entre otros).

85

Llamada a Procedimiento Remoto (RPC)

Visión de una RPC.

- Si se compara una **RPC** con las primitivas ***send()*** y ***receive()***, la primera puede ser vista como una primitiva de alto nivel de pasaje de mensajes.
- Desde el punto de vista del programador: la invocación a un proceso remoto tiene el mismo efecto que una invocación a un procedimiento estándar local; es decir, se transfiere el control a otro procedimiento (remoto), mientras se suspende al programa que lo invoca.
- Una sentencia ***"return"*** ejecutada por el procedimiento invocado, transfiere nuevamente el control al procedimiento original, en dónde se continúa con la ejecución de la instrucción que sigue luego del llamado al procedimiento remoto.

86

Llamada a Procedimiento Remoto (RPC)

Características del procedimiento local y remoto.

Diferencia:

- El procedimiento remoto ejecuta en un espacio de direcciones totalmente separado; por lo tanto, el proceso local que invoca una **RCP** no puede compartir ninguna variable global con el procedimiento remoto.

Intercambio de información:

- El proceso que invoca una **RPC** debe pasar todos los valores al procedimiento remoto como parámetros de entrada.
- De igual manera, los resultados son devueltos al proceso local que invocó la **RPC** como parámetros de salida.

87

Llamada a Procedimiento Remoto (RPC)

Las RPC desde el punto de vista de la implementación.

- Varían totalmente respecto de los procedimientos normales: debido a que un procedimiento remoto ejecutará en un **espacio diferente de direcciones** (en otro computador), **no puede ser parte** (proceso padre o hijo) del proceso que lo invoca. Por lo tanto se debe crear un proceso separado en el otro computador para ejecutar la RPC.

- Todos éstos temas (***send()*** y ***receive()***, ***RPC***) se desarrollarán con mayor detalle en el Módulo IV – Sistemas de Múltiples Procesadores.

88

Resumen del Módulo I – Concurrencia

Definición de concurrencia.

Está relacionada con la interacción de procesos:

- Comunicación.
- Acceso ordenado a recursos de uso exclusivo.
- Sincronización de procesos.
- Asignación de tiempo de procesador.

Entornos de ejecución concurrente de procesos.

- Multiprocesadores con Memoria Compartida (**MP**).
- Multicomputadores con paso de mensajes (**MC**).
- Sistemas distribuidos (**SD**).

89

Resumen del Módulo I – Concurrencia

Tipos de interacción de procesos.

- Competencia.
- Cooperación.

Clasificación de procesos según el tipo de interacción.

- Procesos independientes entre sí.
- Procesos indirectamente dependientes entre sí.
- Procesos directamente dependientes entre sí.

Conceptos fundamentales.

- Exclusión mutua.
- Recursos compartidos de uso exclusivo.

90

Resumen del Módulo I – Concurrencia

Problema de la Sección Crítica.

- Escenario general.
- Soluciones por software.
- Soluciones por hardware.
- Solución con semáforos.
  - Operaciones P y V.
  - Semáforos binarios.
  - Espera ocupada.
  - Uso de semáforos en sincronización de procesos.

91

Resumen del Módulo I – Concurrencia

Entornos con Memoria No Compartida.

- Multicomputadores con paso de mensajes (MC).
- Sistemas distribuidos (SD).

Concurrencia en entornos con Memoria No Compartida.

- Primitivas ***send*** y ***receive***.
- Llamadas a procedimientos remotos (***RPC***).

92