

SINCRONIZACIÓN Y COMUNICACIÓN ENTRE PROCESOS

SISTEMAS OPERATIVOS

Prof. PhD Mirella Herrera

SINCRONIZACIÓN ENTRE PROCESOS

Sincronización

- Es un conjunto de protocolos y mecanismos utilizados para preservar la integridad y consistencia del sistema, cuando varios procesos concurrentes comparten recursos que son reutilizables en serie (sólo pueden ser utilizados por un proceso a la vez). Su estado y operación pueden resultar corrompidos si son manipulados concurrentemente y sin sincronización por más de un proceso. Ej: variables compartidas para Lectura-Escritura, dispositivos físicos como impresoras, etc.

Tipos de procesos

Un sistema operativo multiprogramado permite

Ejecución concurrente compartición de recursos entre varios procesos activos

Tipos de procesos:

Procesos independientes

- Su estado no es compartido
- Son deterministas
- Son reproducibles
- Pueden ser detenidos y reiniciados
- sin ningún efecto negativo

Procesos cooperantes

- Su estado es compartido
- Su funcionamiento no es determinista
- Su funcionamiento puede ser irreproducible
- Si son detenidos y posteriormente reiniciados puede que se produzcan efectos negativos

Comunicación entre procesos

- Los procesos concurrentes pueden comunicarse con propósitos tales como: intercambiar datos, transmitir información sobre los progresos respectivos y acumular resultados colectivos.
- Una memoria compartida, accesible a todos los procesos, proporciona un medio sencillo y habitual para la comunicación.

Condición de Competencia

Mem. Pcpal.

In=7
Out=4

Proceso A

Proceso B

5	Prog1
	Prog2
6	
7	

Direct

Mem. Pcpal.

In=7
Out=4

Proceso A
ProxA=In
Fin Quantum

Mem. Pcpal.

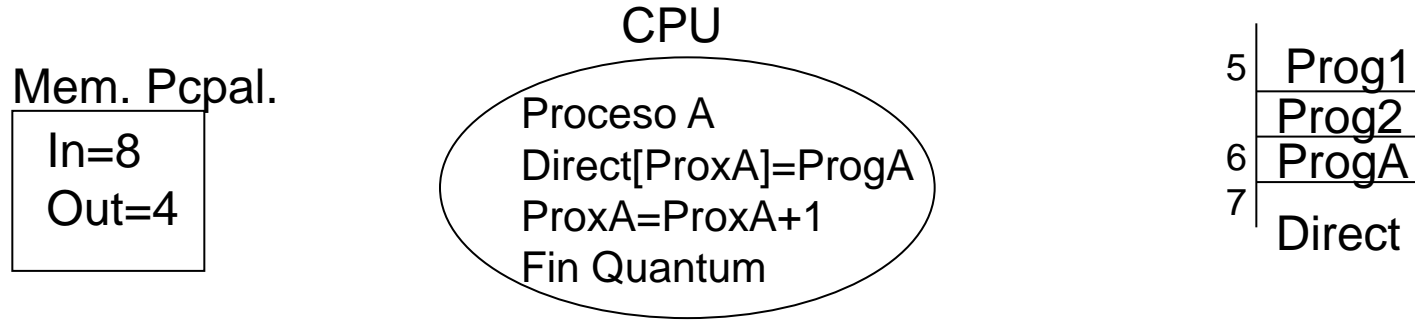
In=8
Out=4

Proceso B
ProxB=In
Direct[ProxB]=ProgB
ProxB=ProxB+1
Fin Quantum

5	Prog1
	Prog2
6	ProgB
7	

Direct

Condición de Competencia



Condición de Competencia: Situación en la que dos o mas procesos intentan realizar alguna operación (Lectura-Escritura) en ciertos datos compartidos (In, Out) y el resultado final depende de quién ejecuta qué y en qué momento

Conclusión: Se alcanza este estado incorrecto por que se permite que ambos procesos manipulen la variable In en forma concurrente generando inconsistencia en los valores de la variable compartida Para remediar esta dificultad necesitamos asegurar que sólo un proceso en cada ocasión pueda manipular la variable In y esto requiere de algún tipo de sincronización entre los procesos A y B

Exclusión Mutua

Cuando un proceso ejecuta código que manipula datos compartidos (o recursos), se dice que el proceso está en su **Sección Crítica (SC)** (para esos datos compartidos)

La ejecución de secciones críticas debe ser *mutuamente excluyente*:

- En cualquier momento, sólo a un proceso se le permite ejecutar en su **Sección Crítica** (incluso con múltiples CPUs)
- Entonces cada proceso debe requerir el permiso para entrar en su sección crítica (SC)

Exclusión Mutua

- La sección de código que lleva a cabo este requerimiento se llama la sección de entrada. **Protocolo Negociación**
- La sección crítica (CS) podría ser seguida por una sección de salida. **Protocolo de Liberación**
- El código restante es el resto de la sección

El problema de la sección crítica consiste en diseñar un protocolo que los procesos puedan usar, para que su acción no dependa del orden en el cual su ejecución es entrelazada (posiblemente en muchos procesadores)

Exclusión Mutua

Toda solución debe cumplir las siguientes condiciones:

1. **Exclusión mutua:** No puede haber dos procesos simultáneamente en la región crítica. Si un proceso P_i se está ejecutando en su región crítica, ningún otro proceso se puede ejecutar en dicha región crítica.
2. **Progreso:** Ningún proceso fuera de la región crítica puede bloquear a otros procesos.
3. **Espera acotada:** Ningún proceso debe esperar infinitamente para entrar en su región crítica.
4. **Los procesos se ejecutan a una velocidad $<>0$**
 - Su velocidad relativa no influye
 - No se realiza ninguna asunción sobre la velocidad de los procesos o sobre el número de CPUs

Exclusión Mutua

- Estructura general de un proceso:

repeat

entry section *Protocolo de negociación*

critical section

exit section *Protocolo de Liberación*

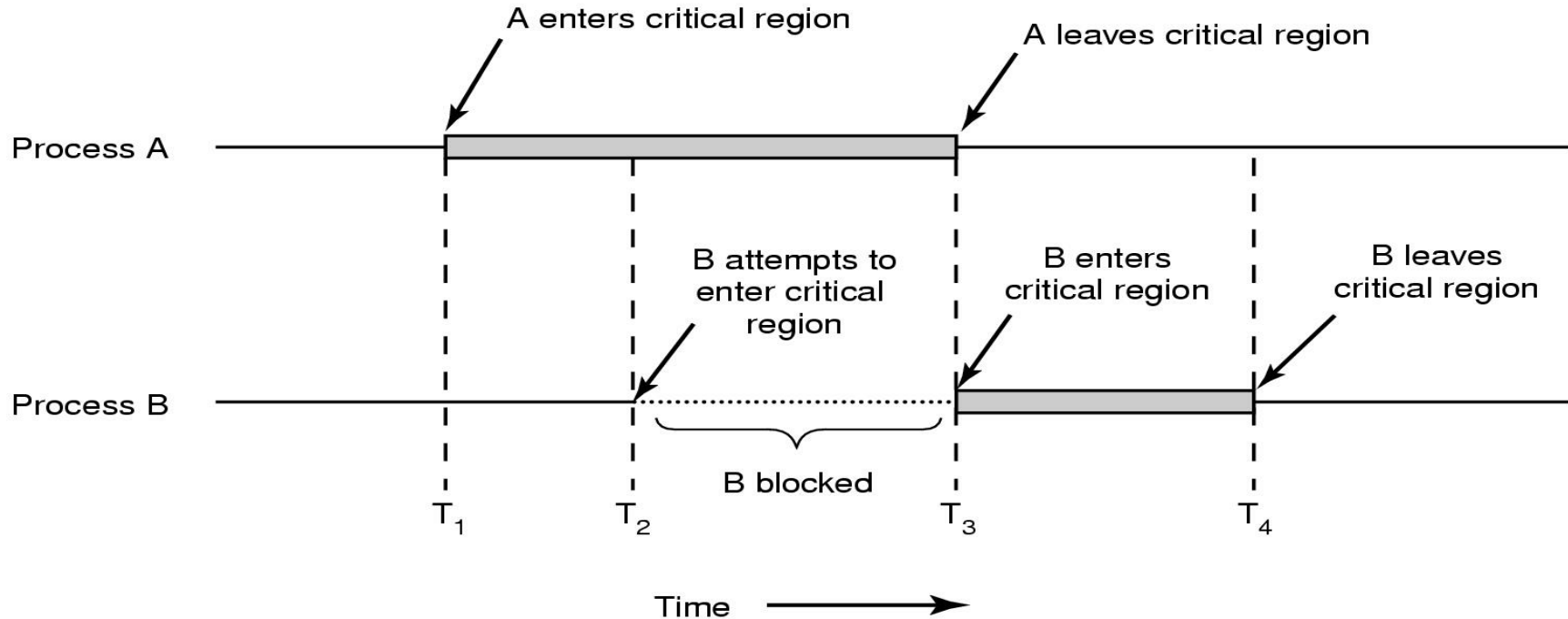
remainder section

forever

Exclusión Mutua

- Cada proceso se ejecuta a una velocidad diferente de cero, pero no se puede asumir nada con respecto a la velocidad relativa de n procesos
- Muchos CPUs pueden estar presentes pero el hardware de memoria previene el acceso simultáneo a la misma localización de memoria
- No se puede asumir nada sobre el orden de ejecución entrelazada
- Para las soluciones:
 - ❖ Se necesita especificar secciones de entrada y de salida

Exclusión Mutua



Soluciones a la Exclusión Mutua

Soluciones

1. Basadas en variables de control: Solución de Dekker, Peterson y la generalización para n procesos de Lamport
2. Basadas en instrucciones máquina: HI-DI, test-and-set, compare-and-swap
3. Basadas en primitivas del SO: Semáforos
4. Basadas en lenguajes de alto nivel: Regiones críticas y monitores para lenguajes de programación como Ada, Java, Pascal

SOLUCIONES SOFTWARE

Soluciones Software

- ✓ **Progresión de las soluciones de Dekker**
 - Intentos 1 al 5 – Algoritmo de Dekker
- ✓ **Algoritmo de Peterson**
- ✓ **Algoritmo de la Panadería de Lamport para n procesos**

SOLUCIONES HARDWARE

Soluciones Hardware

- ✓ **Habilitación/deshabilitación de interrupciones**
- ✓ **Test & Set**
- ✓ **Compare& Swap**

Soluciones Hardware

Habilitar/Deshabilitar Interrupciones

Muchos sistemas proporcionan soporte hardware para secciones críticas

Uniprosesadores

- Se pueden deshabilitar las interrupciones
- El código se ejecuta sin expulsión: poco recomendable
- Generalmente muy ineficiente en multiprosesadores
- Hace que el SO no escale bien

Soluciones Hardware

- **Habilitar/Deshabilitar Interrupciones**

Process Pi

repeat

disable interrupts

critical section

enable interrupts

remainder section

forever

Soluciones Hardware

Instrucciones Hw especiales de ejecución atómica:

Atómica = no-interrumpible

- ❑ Lee una palabra o bit y escribe un valor (TAS o TSL)
- ❑ Intercambia el contenido de dos posiciones de memoria o una posición de memoria y un registro (SWAP)

SEMÁFOROS

Semáforos

- ✓ **Semáforos Binarios**
- ✓ **Semáforos Generales o Contadores**
- ✓ **Implantación de semáforos:**
 - Con Espera Ocupada
 - Con colas
- ✓ **Señalización con Semáforos**

Semáforos

- Es un mecanismo propuesto por Dijkstra (1965), para la exclusión mutua entre un número arbitrario de procesos
- Un semáforo S es una variable entera, sobre la cual se definen tres (3) operaciones:
 1. Inicialización en un valor no negativo
 2. Wait (s) → Disminuye el valor del semáforo
P (Proberen) → Probar en holandés
 3. Signal (s) → Incrementa el valor del semáforo
V (Verhogen) → Incrementar en holandés

Semáforos

Consta de dos operaciones primitivas: Señal-Signal-V y Espera-Wait- P que operan sobre el tipo especial de variable semáforo S:

- **Wait (S):** Decrementa el valor de su argumento semáforo S, en tanto que el resultado no sea negativo. La concusión de la operación *wait*, una vez tomada la decisión de decremento a su argumento semáforo, debe ser indivisible
- **Signal (S):** Incrementa el valor de su argumento semáforo S, en una operación indivisible

Semáforos Binarios

- Pueden tomar valor 0 si el RC está ocupado y 1 si está libre
- **Wait (s)** es la espera hasta que la variable semáforo sea igual a LIBRE, seguido de su modificación indivisible para que indique OCUPADO antes de devolver el control al invocador. La operación *wait* implementa por tanto la fase de negociación del protocolo de EM
- **Signal (s)** pone el valor de la variable en Libre y representa la fase de liberación

Semáforos Generales o Contadores

- ✓ Pueden tomar cualquier valor entero no negativo
- ✓ Son particularmente útiles cuando hay que asignar un recurso a partir de un conjunto de recursos idénticos
- ✓ Tienen como valor inicial el número de recursos del conjunto
- ✓ Cada operación Wait(s) decrementa el semáforo s en 1 indicando que se ha retirado un recurso del grupo y lo está utilizando algún proceso
- ✓ Cada operación Signal (s) incrementa en 1 el semáforo lo que indica la devolución de un recurso al grupo y que el recurso puede ser reasignado
- ✓ Si se intenta una operación Wait cuando el semáforo tiene valor 0, el proceso deberá esperar hasta que se libere un recurso, mediante una operación Signal

Semáforos con Espera Ocupada

Implantación

Wait(s): while not ($s > 0$) do; {espera}
 $s := s - 1$;

Signal(s): $s := s + 1$;

Semáforos con Espera Ocupada

Program

Exclusion_Semaforo

Var mutex: semaforo;

Process P1

```
begin
    wait(mutex)
    seccion_critica_1;
    signal(mutex);
    otras_tareas_P1;
end;
end;
```

Process P2

```
begin
    wait(mutex)
    seccion_critica_2;
    signal(mutex);
    otras_tareas_P2;
end;
end;
```

Process P3

```
begin
    wait(mutex)
    seccion_critica_3;
    signal(mutex);
    otras_tareas_P3;
end;
end;
```

{Programa Principal}

```
begin
    mutex:= 1; {LIBRE}
    initiate P1, P2, P3;
end;
```

Semáforos con Espera Ocupada

Tiempo	Estado/Actividad de los Procesos				Procesos en SC
	P1	P2	P3	1=L,0=O	
M1	-----	-----	-----	1	---;---
M2	Wait(mutex)	Wait(mutex)	Wait(mutex)	0	--;P1,P2,P3
M3	SC	esperando	esperando	0	P1,P2,P3
M4	Signal(mutex)	esperando	esperando	1	--;P2,P3
M5	Otras_tareas_p1	SC	esperando	0	P2;P3
M6	Wait(mutex)	SC	esperando	0	P2;P3,P1
M7	esperando	Signal(mutex)	esperando	1	--;P3,P1
M8	SC	Otras_tareas_p2	esperando	0	P1;P3

Semáforos con Cola

Implantación

Wait(s): Si no ($s > 0$) entonces

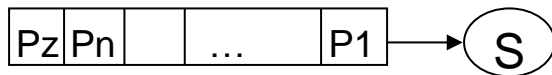
Suspender al proceso invocador en S
en caso contrario

$s := s - 1;$

Signal(s): Si no cola vacía (al menos un proceso espera) entonces

Reanudar uno de los procesos en la cola de s
en caso contrario

$s := s + 1;$



Estructura del Semáforo

Semáforos con Cola

- Las operaciones de semáforos se pueden incluir en el núcleo del SO para evitar la espera activa
- Un semáforo se implanta con una variable protegida y una cola de PCBs en la cual los procesos esperan por operaciones V
- Cuando un proceso intenta una operación P sobre un semáforo en 0, renuncia al procesador y se bloquea a sí misma para esperar por una operación V sobre el semáforo
- El SO inserta el PCB del proceso en la cola del semáforo asociado al recurso con una política FIFO

Semáforos con Cola

- Cuando el recurso quede finalmente disponible, una operación V sobre el semáforo permite reanudar al proceso suspendido (quita el proceso de la cola de bloqueados y lo pasa a la cola de listos)
- La operación V intenta despertar un proceso que espera y solo cuando no hay ninguno incrementa la variable semáforo (la variable semáforo se encuentra en estado ocupado mientras haya procesos esperando en la cola)
- Un proceso nuevo que ejecute P será insertado en la cola siempre que el semáforo esté ocupado

Señalización con Semáforos

Program Exclusion_Semaforo

Var evento: semaforo;

Process P1

```
begin
  tareas_prelim_p1
  wait(evento)
  otras_tareas_P1;
end;
```

Process P2

```
begin
  tareas_prelim_p2
  signal(evento)
  otras_tareas_P2;
end;
```

PROGRAMA PRINCIPAL

```
begin
  inicia_semaforo(evento,0)
  initiate P1,P2;
end;
```

Esta idea esta basada en el hecho de que los semáforos son muy útiles y convenientes para intercambiar señales de sincronización entre procesos cooperativos

La pareja wait y signal se usa por separado entre dos procesos

P1 ejecuta wait para esperar a que ocurra el evento. P2 ejecuta signal para señalar que ha ocurrido el evento lo que permite que P1 continúe

Este mecanismo funciona aunque P2 realice signal sin que P1 ejecute wait

Semáforos

Ventajas del uso de Semáforos

- Potente Mecanismo para la sincronización y señalización entre procesos
- Es una herramienta simple y relativamente fácil de comprender

Desventajas del uso de Semáforos

- No son estructurados. La sincronización e integridad del sistema dependen de la estricta adherencia de todos los programadores al orden de las operaciones wait y signal. Invertir el orden de y signal, olvidar alguna de ellas o simplemente saltarlas puede corromper o bloquear fácilmente el sistema entero
- No soportan abstracción de datos. Solo pueden proteger el acceso a la SC y no pueden restringir el tipo de operaciones sobre el RC
- No están relacionados sintácticamente a los recursos que protegen

MONITORES

Monitores

Definición

Son estructuras de un lenguaje de programación que ofrecen funcionalidad equivalente a los semáforos pero mas fáciles de controlar. Definidos por Hoare (1974), se han implantado en lenguajes como Pascal Concurrente, Modula-2, Modula-3, Pascal Plus

Es un modulo software que consta de uno o más procedimientos, una secuencia de inicialización y unos datos locales

Monitores

Se caracterizan por

- Las variables de datos solo son accesibles a los procedimientos del monitor y no por los procedimientos externos
- Un proceso entra al monitor invocando uno de los procedimientos
- Un solo procedimiento puede estar ejecutándose en el monitor en un instante dado; cualquier otro proceso que haya invocado al monitor quedara suspendido, esperando que el monitor se libere

Monitores

Mecanismo de Sincronización

Es provisto a través de las variables de tipo ***condition***, con dos operaciones

cwait(c): Suspende la ejecución del proceso llamado bajo la condición c. El monitor queda disponible para ser usado por otro proceso

csignal(c): Reanuda la ejecución de algún proceso suspendido después de un *cwait* bajo la misma condición. Si hay varios procesos elige uno de ellos y si no hay no se hace nada.

Monitores

Type monitor-name = monitor

Variable declarations

Procedure entry P1 (...);

begin end;

Procedure entry P2(...);

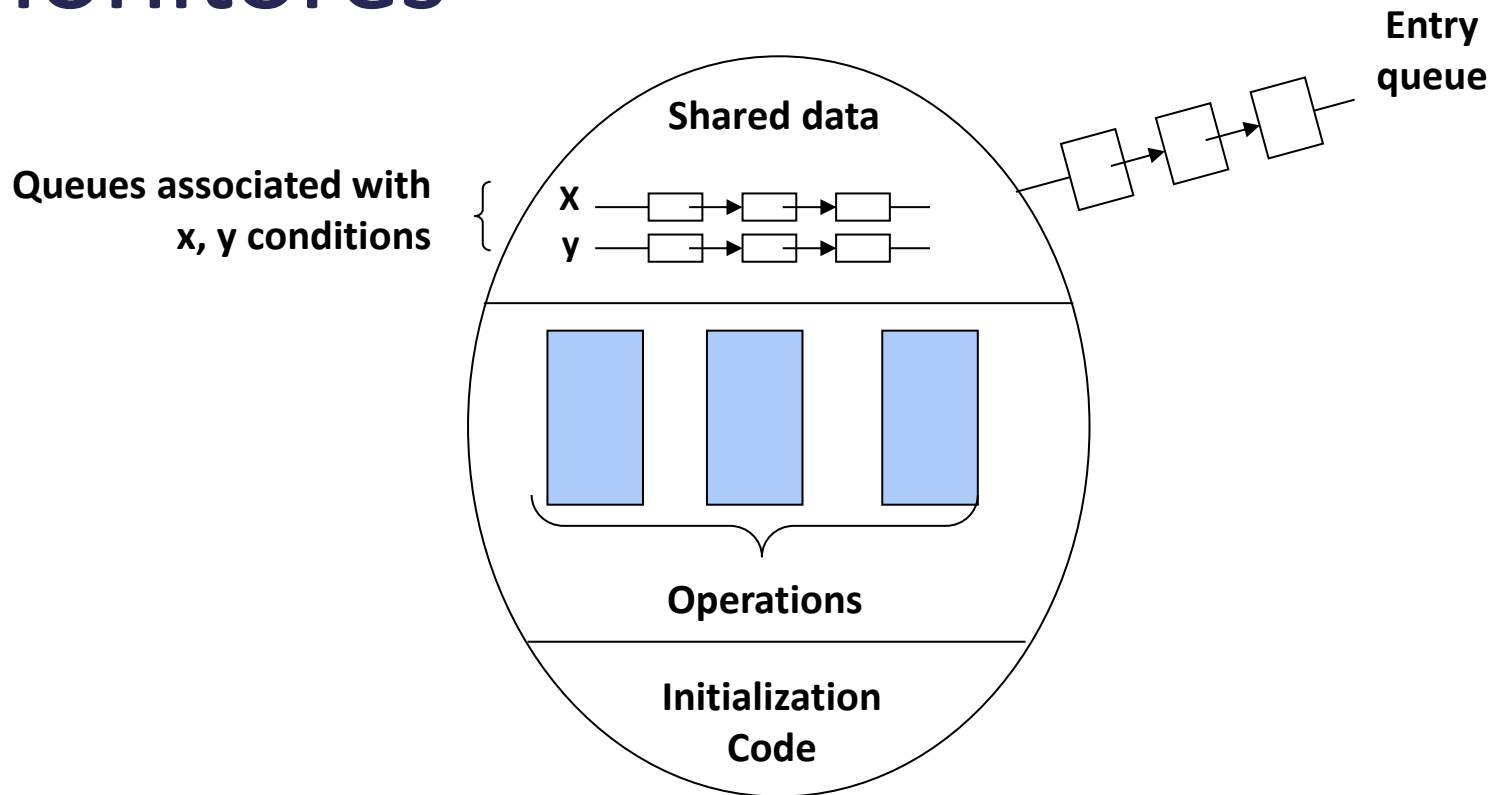
begin end;

begin

initialization code

end;

Monitores



COMUNICACIÓN ENTRE PROCESOS

Paso de Mensajes

- ✓ **Definición**
- ✓ **Aspectos de Implementación**
 - Denominación
 - Copia
 - Intercambio sincronizado o asíncrono de mensajes
 - Longitud
- ✓ **Protocolo EM**

Paso de Mensajes

Definición

- Son un mecanismo relativamente sencillo, adecuado para garantizar tanto la EM como la comunicación entre procesos en entornos centralizados y distribuidos
- El envío y recepción de mensajes es una forma estándar de comunicación entre nodos de una red de computadores
- Un **mensaje** es una colección de información que puede ser intercambiada entre un proceso emisor y receptor, puede contener datos, ordenes de ejecución e incluso código a transmitir entre dos o mas procesos
- Un mensaje se caracteriza por tipo, longitud, identificadores de emisor y receptor y un campo datos.

Aspectos de Implementación

Denominación

- **Denominación Directa:** Cuando se invoca a una operación de mensajes, cada emisor debe designar al receptor específico y a la inversa, cada receptor debe designar a la fuente desde la cual se desea recibir un mensaje, por lo cual es una comunicación simétrica
Problema: Inconveniente para rutinas de servicio, por ejemplo un gestor de impresoras que debe especificar el id de cada cliente.
- **Denominación Indirecta:** Los mensajes son enviados y recibidos a través de depósitos especializados llamados buzones. Es necesario disponer de servicios del sistema para el mantenimiento de los buzones (`crear_buzon`, `eliminar_buzon`)
 - La operación `send` deposita el mensaje producido en el `buzon1` y la operación `receive` retira un mensaje del `buzon1` y lo entrega al proceso receptor.

Aspectos de Implementación

Copia

El intercambio de mensajes entre procesos transfiere el contenido del mensaje desde el espacio de direcciones del emisor al del receptor. Debe haber un compromiso entre seguridad y eficiencia

- **Copia todo el mensaje al espacio de direcciones del receptor:** Se transfiere el mensaje por valor. Es inevitable en sistemas distribuidos sin memoria común. Las modificaciones del emisor y el receptor sobre el mensaje se realizan sobre la copia local

Problema: Consume memoria y ciclos de procesador ya que pueden requerir que el mensaje sea copiado desde el espacio del emisor a un buffer del SO y desde allí al espacio del receptor. El SO debe disponer de un espacio de memoria dinámico para el proceso de copia

Aspectos de Implementación

Copia

- **Pasa un puntero al mensaje entre los dos procesos:** Se transfiere el mensaje por referencia. Es la solución menos segura pero mas rápida, proporciona al receptor una ventana al espacio de direcciones del emisor
 - Problema:** Mientras el receptor este utilizando el mensaje el receptor no puede modificarlo.
- **Híbrido:** Inicialmente el SO intenta optimizar el rendimiento compartiendo una sola copia del mensaje que es accesible tanto al espacio de direcciones del emisor como al receptor siempre que ambos accedan solo para lectura. Si alguno de ellos intenta modificar el mensaje el SO interviene y crea una copia física separada del mensaje para cada proceso.

Aspectos de Implementación

Intercambio sincronizado o asíncrono de mensajes

- **Sincronizado:** El emisor y el receptor deben proceder juntos a completar la transferencia. La operación *send* es bloqueante, es decir, el emisor queda suspendido hasta que el receptor acepte el mensaje.
 - **Ventaja:** Facilidad de implementación, el emisor está seguro de la recepción de su mensaje cuando prosigue la instrucción *send*.
 - **Desventaja:** Obligatoria operación sincronizada de emisores y receptores.

Aspectos de Implementación

Intercambio síncrono o asíncrono de mensajes

- **Asíncrono:** El emisor no se bloquea por *receive* pendientes. El SO almacena temporalmente los mensajes pendientes por *receive*. El emisor continúa su ejecución independientemente de los receptores.

Problemas:

- Posibilidad útil pero peligrosa si se abusa de ella pues un proceso pudiera generar tantos mensajes sin control y agotar la memoria temporal del sistema. Una solución es asignar un espacio de almacenamiento temporal para cada emisor/receptor.
- Aplazamiento indefinido o Bloqueo entre procesos cuando se envía un mensaje y nunca se recibe por que el receptor se cancela o cuando el receptor espera por un mensaje que nunca se produce.

Otra solución pudiera ser *receive* sin bloqueo o con espera temporizada en este ultimo se especifica un límite de tiempo durante el cual debe completarse un intercambio de mensaje, sino llega ninguno el SO devuelve el control al receptor indicando que el plazo ha finalizado.

Aspectos de Implementación

Longitud de los mensajes

- Debe haber un compromiso entre el recargo y la flexibilidad. No es de gran importancia la longitud en sistemas donde la transferencia se realiza mediante punteros pero cuando los mensajes son copiados debe ser cuidadosamente evaluado.
- **Fija:** Producen bajo recargo dado que los buffers del sistema serán también de tamaño fijo, lo que hace la asignación sencilla y eficaz.

Problema: Los mensajes son naturalmente de tamaño variable y encajarlos en buffers de tamaño mas pequeño desperdician espacio y los mas largos deben ser divididos y enviados por partes.

- **Variable:** Crea buffers dinámicamente para que se ajusten al tamaño de cada mensaje individual.

Problema: La gestión del fondo de memoria dinámica por parte del SO, lo cual es costos en términos de tiempo de CPU y también puede producir fragmentación de la memoria.

Exclusión Mutua con PM

Protocolo de Exclusión Mutua con Paso de Mensajes

Receive (msj,...n) *Protocolo de Negociación*

Sección Crítica

Send (msj,...,n) *Protocolo de Liberación*