

IPC

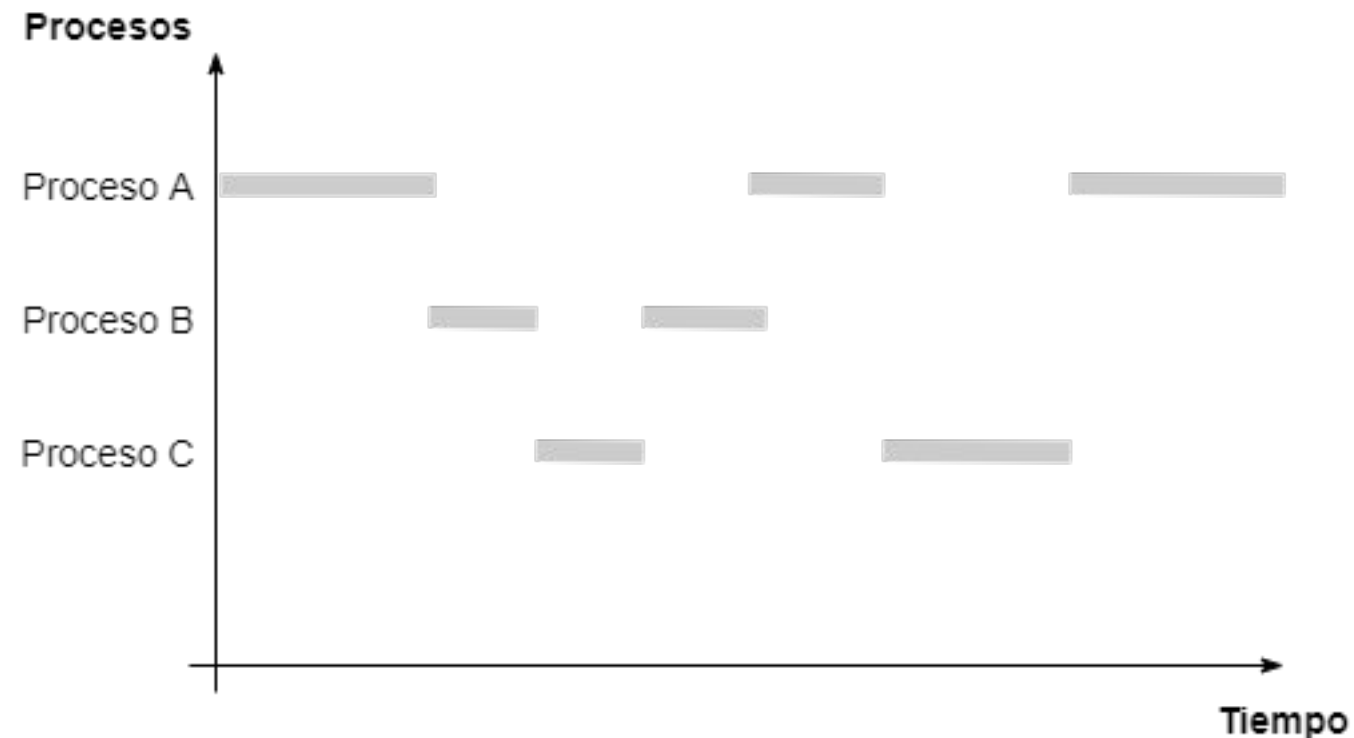
Interprocess

Communication

Adaptación (ver referencias al final)

Modelos de concurrencia

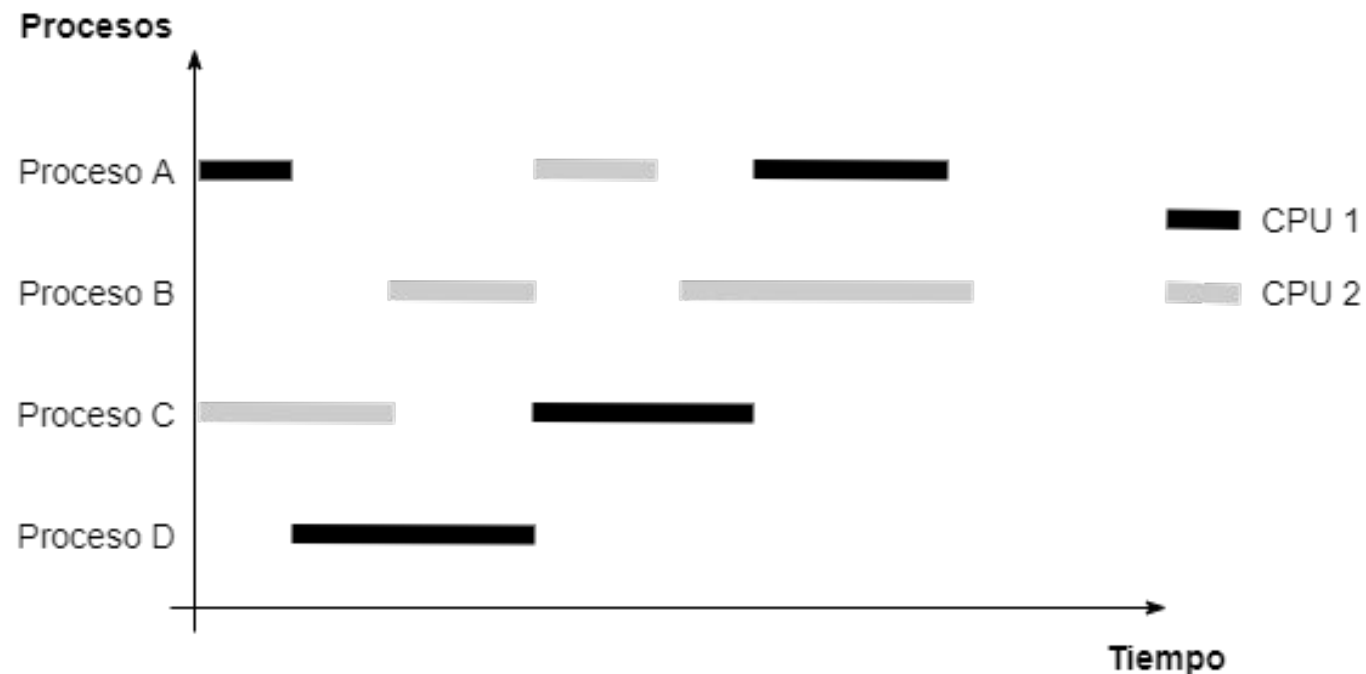
- **Multiprogramación con un solo procesador**
 - Todos los procesos se ejecutan sobre un único procesador
 - Sistema operativo reparte tiempo de procesador entre procesos



Modelos de concurrencia

- **Multiprocesador**

- Múltiples núcleos de procesamiento que comparten memoria principal
- Se intercalan ejecuciones de procesos
- Se superponen ejecuciones de procesos (en diferentes procesadores)



Motivaciones para la concurrencia

- **Facilita la programación**

- Múltiples procesos que cooperan entre sí para alcanzar un objetivo común
- Proceso 1: Compilador genera código ensamblador
- Proceso 2: Ensamblador que genera código de máquina

- **Compartir información**

- Aplicaciones que están interesadas en la misma información
- P. Ej.: copiar – pegar entre diferentes aplicaciones

- **Uso interactivo**

- Varios usuarios trabajando en forma simultánea en varios terminales

Motivaciones para la concurrencia

- **Mejorar el desempeño computacional**

- Necesidad de que una tarea se haga en menos tiempo
- División de procesos que se ejecutan en paralelo (requiere paralelismo)
- Es difícil y no siempre es posible

- **Modularidad**

- Separar funcionalidades en módulos

- **Aprovechamiento de los recursos**

- Fases de E/S de unos procesos se pueden aprovechar para realizar fases de procesamiento de otros.
- Multiplexión de la CPU en tiempos.

Modelos de concurrencia

- **Computación distribuida**

- Múltiples computadores agrupados (nodos) que se ven como uno solo
- Cada computador tiene su(s) procesador(es) y memoria
- Los nodos se comunican entre sí a través de una red de comunicaciones
- Es posible ejecutar múltiples procesos sobre los diferentes procesadores

Procesos independientes y cooperativos

- Procesos que se ejecutan concurrentemente en el S.O pueden ser
 - Independientes
 - Cooperativos
- **Independientes**
 - No comparten datos con ningún otro proceso.
 - P. Ej.: las *shell* de Linux son procesos independientes. Se pueden tener varias en ejecución concurrente pero cada proceso es independiente
- **Cooperativos**
 - Afectan o son afectados por otros procesos en ejecución en el S.O

Interacciones entre procesos

- **Procesos que comparten o compiten**

- Dos procesos independientes que compiten por el acceso a disco
 - S.O debe gestionar el orden de acceso al recurso por el que compiten
 - S.O debe evitar conflictos de acceso al recurso por el que compiten
- Dos procesos que intentan modificar el mismo registro en una base de datos
 - Gestor de BD debe gestionar el orden de acceso al registro

- **Procesos que se comunican y se sincronizan**

- Procesos que se comunican y se sincronizan entre sí para alcanzar un objetivo común
- P: Ej.: proceso de compilación y ensamblado.

Comunicación entre procesos

- ¿Cómo un proceso puede pasar información a otro proceso?
 - Mecanismos de IPC
- ¿Cómo hacer que dos o más procesos no se interpongan entre sí?
 - Dos procesos que intentan actualizar información a la vez.
- ¿Cómo mantener la secuencia apropiada cuando hay dependencias?
 - Proceso A produce datos que proceso B consume

Modelos de IPC

- **Memoria compartida**

- Se designa una sección de memoria que todos los procesos cooperativos comparten mediante operaciones de lectura y escritura.

- **Archivos compartidos**

- Procesos que comparten un archivo común para leer y escribir datos en él.

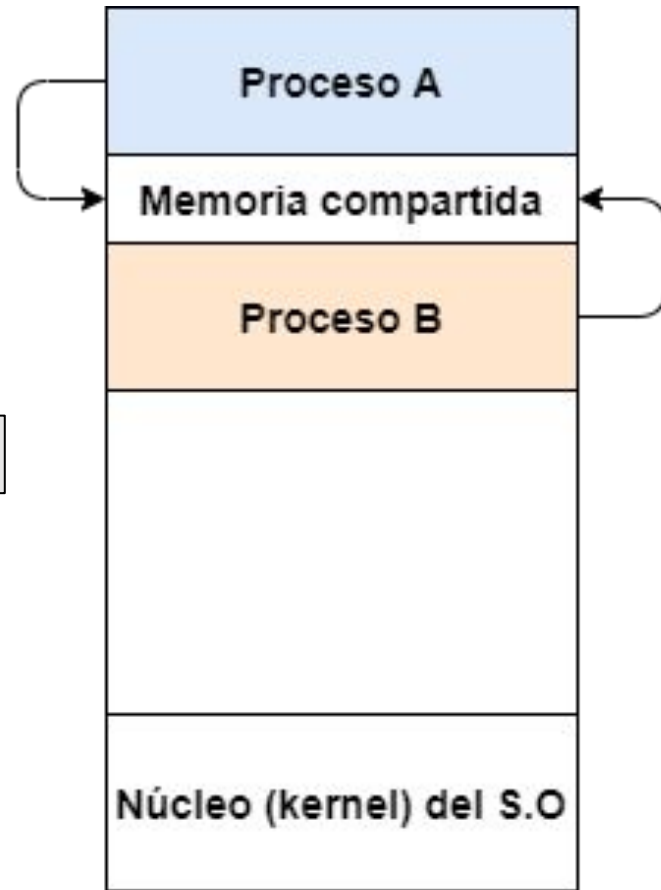
- **Paso de mensajes**

- Mecanismo por el cual los procesos se pasan mensajes entre sí.

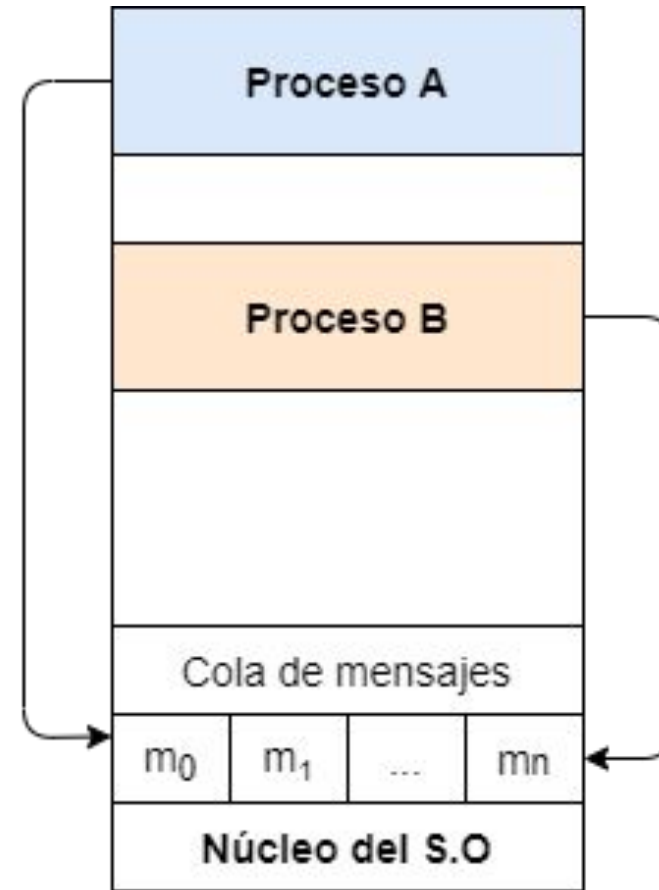
- **Tuberías**

- Mecanismo por el cual un proceso pone datos de un extremo de la tubería y el otro los lee del otro extremo.

Modelos de IPC

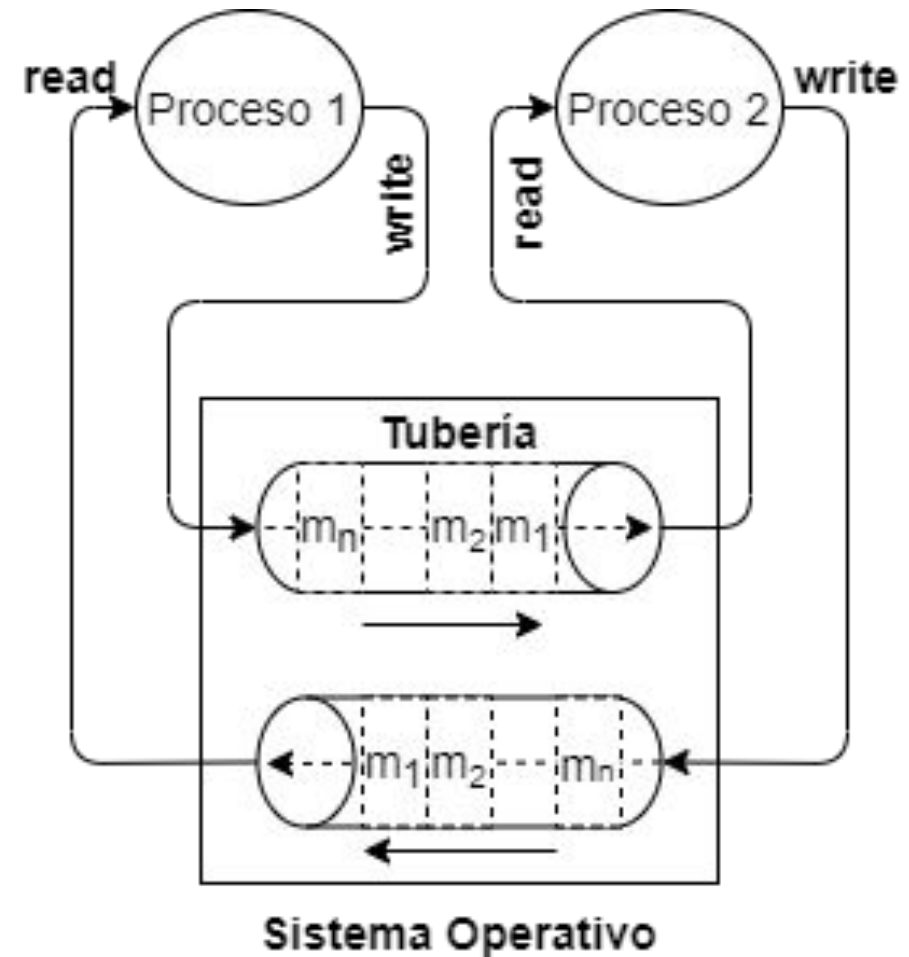
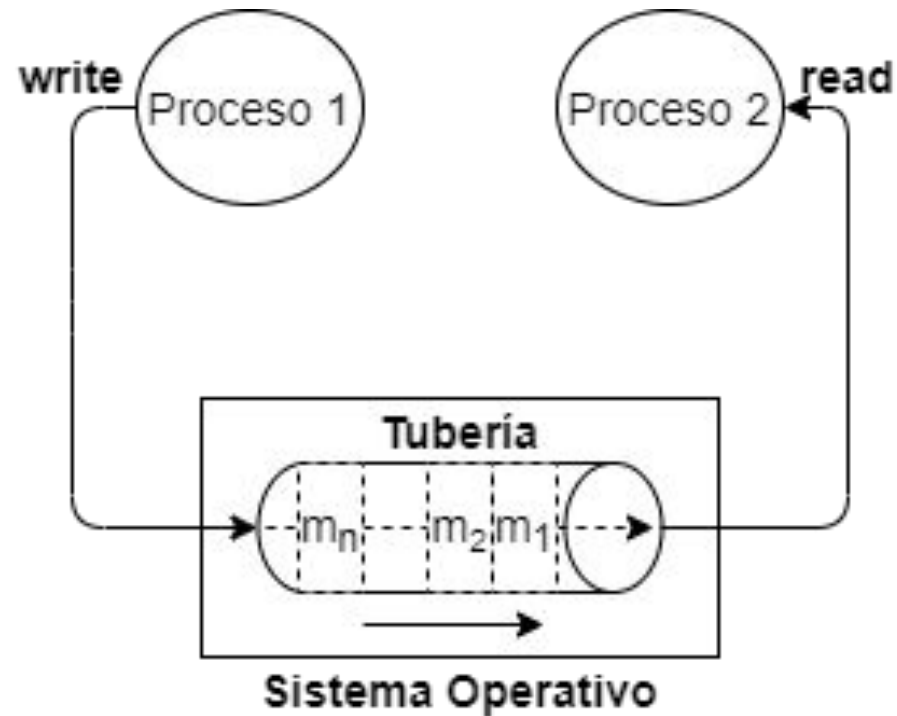


Memoria



Memoria

Tuberías (paso de mensajes)



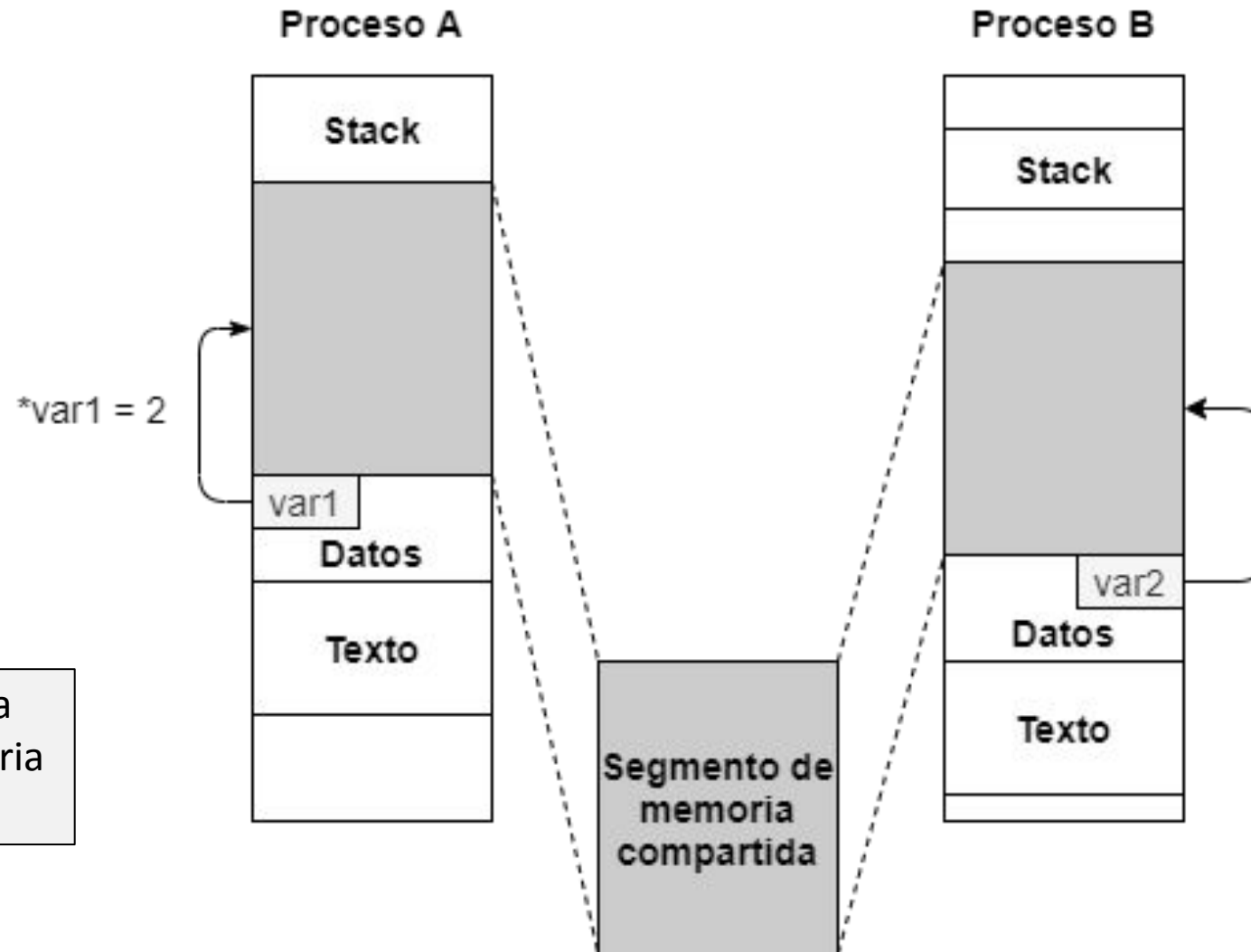
Problema del productor – consumidor

- Uno o más procesos **productores** generan/producen datos
- Uno o más procesos **consumidores** consumen datos producidos
- Por ejemplo:
 - Compilador produce código en ensamblador.
 - Ensamblador consume código en ensamblador y produce archivo(s) objeto(s).
 - Cargador consume archivos objetos para crear procesos.
- Se necesita mecanismo de cooperación entre procesos.
- Se necesitan mecanismos de sincronización entre procesos.
 - Evitar interacciones problemáticas
 - ¿Qué pasa si no se han consumido lo producido? ¿Qué pasa si no hay nada para consumir?

Memoria compartida

- La región de memoria compartida reside en el espacio de memoria del proceso **que crea** el segmento compartido
- Procesos que se quieran comunicar deben ***enganchar*** esta región compartida **a su propio** espacio de memoria
- Los procesos cooperantes mediante este mecanismo:
 - Responsables de evitar que se escriba la misma posición de memoria a la vez
 - Establecer forma de los datos y ubicación
 - S.O no tiene mucho control sobre lo que sucede el área compartida

Memoria compartida



var1 es un apuntador a una variable en memoria compartida

var2 es un apuntador a una variable en memoria compartida

Memoria compartida

- **Se debe disponer de un búfer**

- El productor pone ítems en el búfer.
- El consumidor consume/saca los ítems del búfer.

- El búfer se dispone en memoria compartida por los dos procesos.

- Se debe sincronizar proceso productor y consumidor

- Evitar que el consumidor consuma un ítem que no ha sido puesto aún por el productor.

- **Dos tipos de búfer**

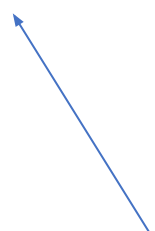
- Ilimitado: consumidor siempre puede producir nuevos ítems.
- Limitado: productor espera si búfer lleno, consumidor espera si búfer vacío.

Memoria compartida: búfer

```
#define TAMANO_BUFFER 10
typedef struct {
    . . .
} item;
item bufer[TAMANO_BUFFER];
int in = 0;
int out = 0;
```

- `bufer` es un arreglo circular.
- `in` apunta a la siguiente posición libre en el búfer.
- `out` apunta a la primera posición llena del búfer.
- Búfer vacío: `in == out`
- Búfer lleno:

`((in + 1) % TAMANO_BUFFER) == out`



Esta información la comparten ambos procesos

Memoria compartida: productor

```
item sgte_item_producido; /* ítem producido */
while (true) {
    while (((in + 1) % TAMANO_BUFFER) == out)
        ; /* no hacer nada */
    /* pone next_item en el búfer */
    bufer[in] = sgte_item_producido;
    /* ajustar el búfer */
    in = (in + 1) % TAMANO_BUFFER;
}
```

Memoria compartida: consumidor

```
item sgte_item_consumido;  
while (true) {  
    while (in == out)  
        ; /* no hacer nada */  
    /* consumir el item */  
    sgte_item_consumido = bufer[out];  
    /* ajustar el búfer */  
    out = (out + 1) % TAMANO_BUFFER;  
}
```

Paso de mensajes

- Permite comunicar procesos y sincronizar sus acciones sin necesidad de compartir el mismo espacio de memoria.
- Dos operaciones/primitivas de operación
 - `send(message)`
 - `receive(message)`
- `message` puede ser de tamaño fijo o tamaño variable
- Mecanismos lógicos de implementación
 - Comunicación directa o indirecta
 - Comunicación sincrónica o asincrónica
 - *Buffering* (almacenamiento) automático o explícito

Paso de mensajes: comunicación directa

- **Nombrado**

- Se debe especificar explícitamente el nombre del proceso con el que se establece la comunicación.
- **send(P, message)** □ Enviar un mensaje al proceso **P**
- **receive(Q, message)** □ Recibir un mensaje del proceso **Q**

- **Características del enlace de comunicaciones**

- Se establece automáticamente.
- Se asocia exactamente **a dos procesos únicamente**.
- Existe un solo enlace entre cada par de procesos.
- Simétrico: ambos procesos conocen la identidad de su par.

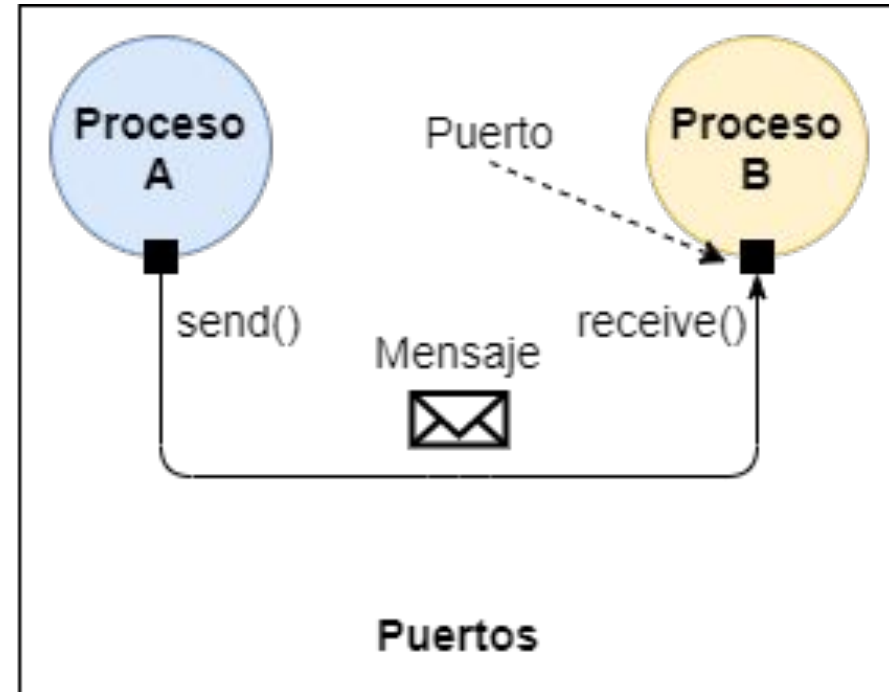
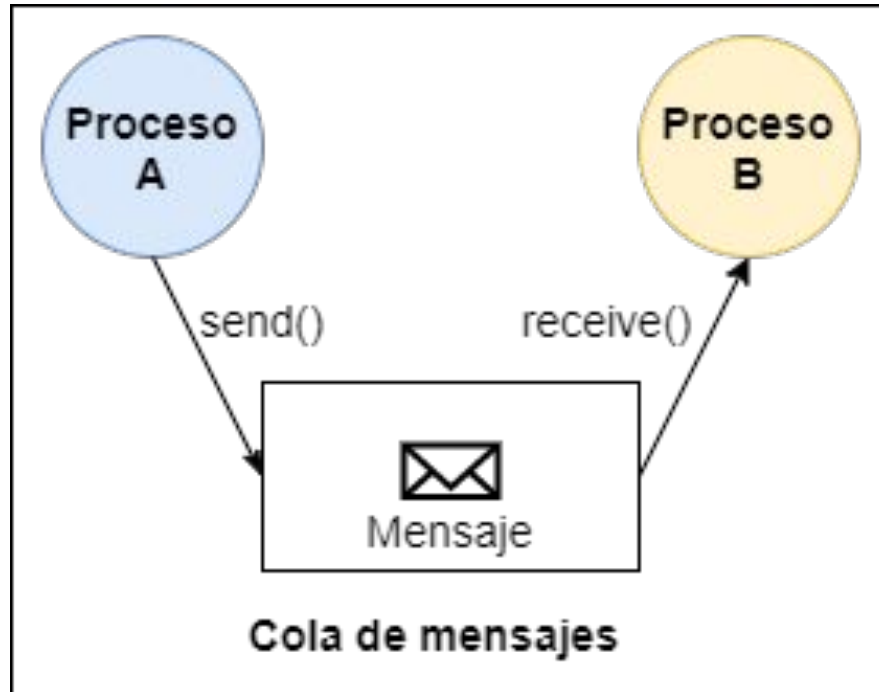
Paso de mensajes: nombrado

- También se puede especificar que el receptor puede recibir mensajes de cualquier proceso
 - `receive(ANY, message)` □ Recibir un mensaje del cualquier proceso
- El **receptor** **no** necesita conocer la identidad del emisor.
- El proceso **emisor** **si** necesita conocer la identidad del receptor.
- Características del enlace de comunicaciones
 - **Asimétrico**: solo una identidad de proceso es requerida

Paso de mensajes: comunicación indirecta

- Los mensajes son enviados y recibidos a través de colas de mensajes (**MQ**) o puertos.
- Una **MQ** es un objeto en el que se ponen y se quitan mensajes
 - Cada MQ tiene un identificador único
- Primitivas/operaciones
 - **send(A, message)** □ Envía un mensaje a la **MQ** o puerto **A**.
 - **receive(A, message)** □ Recibe un mensaje de la **MQ** o puerto **A**.
- Procesos se pueden comunicar con otros a través de diferentes **MQ**.
- Dos procesos se pueden comunicar **si** comparten **una misma MQ**.

Paso de mensajes: comunicación indirecta



- **MQ** o puertos son estructuras indirectas entre los dos procesos.
- Los puertos es una estructura asociada a cada proceso.
 - Proceso termina, puerto desaparece
- Cola de mensajes usualmente implementada en el S.O

Paso de mensajes: comunicación indirecta

- Características del enlace de comunicaciones
 - Se establece si los dos procesos comparten la misma **MQ**
 - Se puede asociar a más de dos procesos.
 - Pueden existir más de un enlace de comunicaciones entre pares de procesos.
 - Cada enlace de comunicaciones está asociado a un buzón.

Paso de mensajes: productor

```
/* Producir un item en sgte_msg_producido */  
message sgte_msg_producido;  
while (true) {  
    /* Enviar ítem producido */  
    send(sgte_msg_producido);  
}
```

Paso de mensajes: consumidor

```
message sgte_msg_consumido;  
while (true) {  
    /* consumir el ítem en sgte_msg_consumido */  
    receive(sgte_msg_consumido);  
}
```

Paso de mensajes: sincronización

- **Envío bloqueante**

- Proceso emisor se bloquea hasta que el mensaje es recibido por el proceso receptor o por la MQ.

- **Envío no bloqueante**

- Proceso emisor envía mensaje y continua su ejecución.

- **Recepción bloqueante**

- Proceso receptor se bloquea hasta que recibe el mensaje.
- Proceso emisor envía mensaje y continua su ejecución.

- **Recepción no bloqueante**

- Proceso receptor puede recibir un mensaje válido o NULL.

Paso de mensajes: *buffering*

- Implementación **sin capacidad** para almacenar mensajes
 - Enlace no puede tener mensajes almacenados
 - Comunicación debe ser síncrona ente procesos
 - Envío bloqueante y recepción bloqueante
 - Sistema sin almacenamiento
- Implementación **con capacidad** para almacenar mensajes
 - Tamaño limitado para almacenar mensajes
 - Si **MQ** no está llena se pone en cola y proceso emisor sigue su ejecución
 - Si **MQ** está llena proceso emisor se bloquea esperando a que se libere espacio.
 - Almacenamiento (*buffering*) automático.

Referencias

- Carretero Pérez, J., De Miguel Anasagasti, P., García Carballeira, F., & Pérez Costoya, F. (2001). Comunicación y sincronización de procesos. In *Sistemas operativos. Una Visión Aplicada* (pp. 223–257). McGraw Hill.
- Silberschatz, A., Galvin B., P., & Gagne, G. (2018). Interprocess Communication. In *Operating Systems Concepts* (10th ed., pp. 123–132). John Wiley & Sons, Inc.