# Sistemas Operativos

Grado en Ingeniería Informática del Software

## Tema 3: Concurrencia

*Miguel Riesco Albizu*

# *Tabla de contenidos*

# *Revisión de conceptos de programación concurrente*

**Concurrencia**

- Ejecución de tareas simultáneamente en el sistema y pueden tener que compartir recursos.

**Paralelismo: Concurrencia real**

- La ejecución de los procesos se realiza en diferentes núcleos o procesadores

**Concurrencia aparente**

- La simultaneidad se simula. Los procesos existen simultáneamente e  intercalan su ejecución en el tiempo.
- Modelo con un único procesador

# *Revisión de conceptos de programación concurrente*

## Tipos de Procesos Concurrentes

### Independientes

- Se ejecutan sin requerir la ayuda o cooperación de otros procesos
- En general la mayor parte de los procesos se ejecutan independientemente de ningún otro.

### Cooperantes

- Diseñados para trabajar conjuntamente en alguna actividad.
- Deben ser capaces de comunicarse e interactuar entre ellos

# *Revisión de conceptos de programación concurrente*

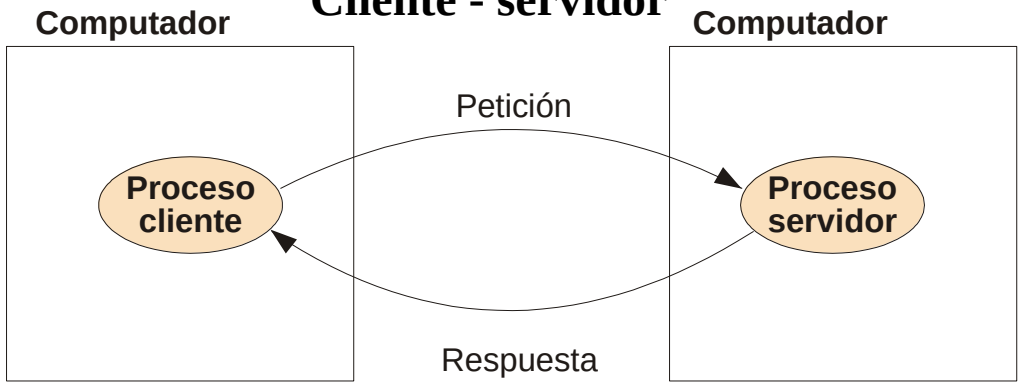
**Interacciones entre procesos** (de cualquier tipo)

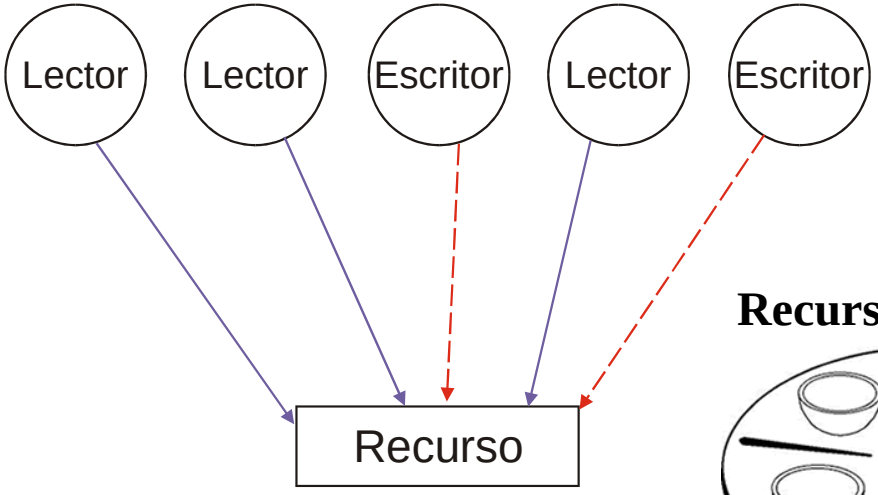
- **Comparten o compiten** por el acceso a un recurso físico o lógico
  - Ej. Dos procesos independientes pueden competir por el acceso a disco
  - Ej. Dos procesos desean modificar el contenido de un registro de la base de datos
- Se **comunican o sincronizan** para alcanzar un objetivo común
  - Ej. (a+b)*(a-b) El proceso que realiza el producto no puede empezar hasta que los que realizan la suma y resta no hayan acabado.
  - Ej $| V | = \sqrt{v1^2 + v2^2 + \cdots + vn^2}$

# *Revisión de conceptos de programación concurrente*

## Problemas clásicos

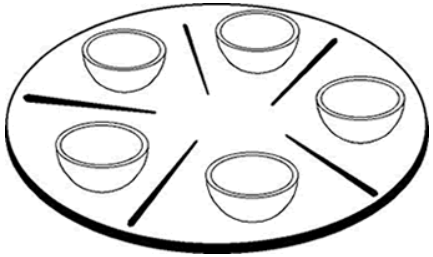
### Cliente - servidor

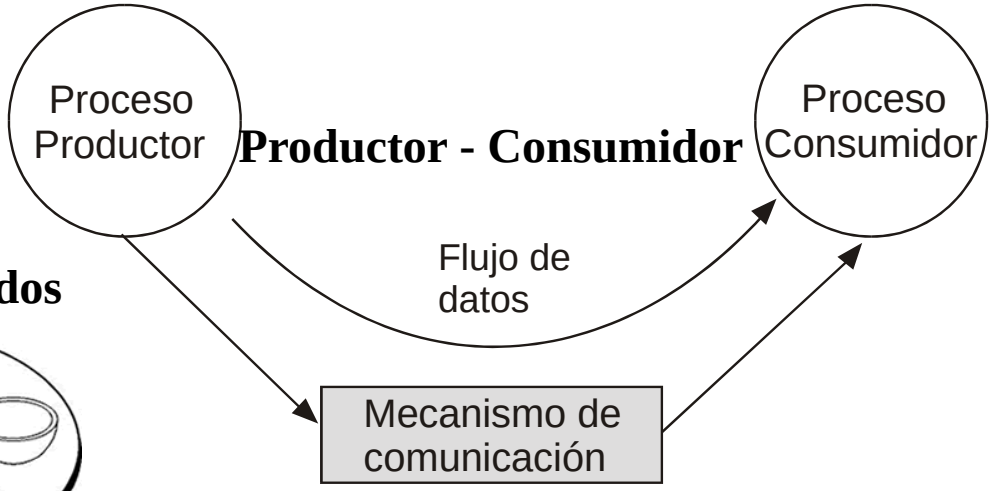



**Lectores - escritores**

### Recursos limitados





# *Revisión de conceptos de programación concurrente*

## Problemas que presenta la concurrencia

1. ***Condiciones de carrera:*** El resultado final de la ejecución de varios procesos concurrentes depende de la secuencia de ejecución.
2. ***Interbloqueo***: bloqueo permanente de varios procesos que compiten por recursos o se sincronizan entre sí.

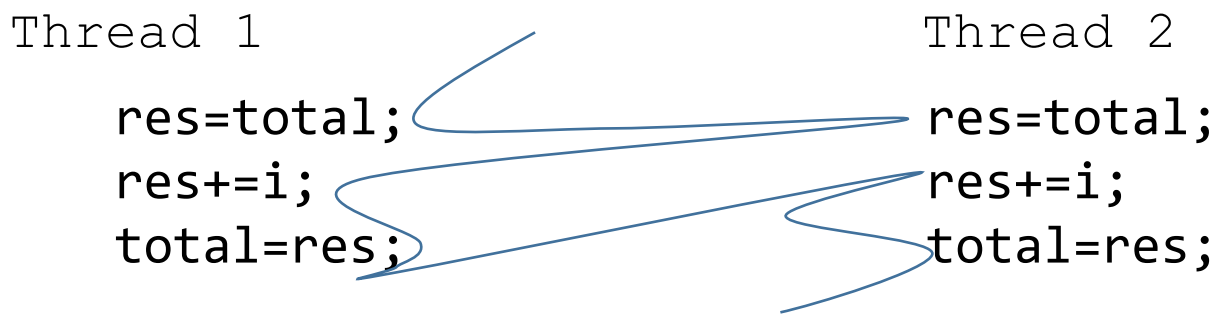
# *Revisión de conceptos de programación concurrente*

## Condiciones de carrera

```
void  partial_addition(int from, until) {

 for (i= from; i<=until; i++) {
  res=total; // get the value of total
  res+=i;    // calculate the new value for total
  total=res; // update the value of total
 }
}

int main() {
  pthread_create(&tid[0], NULL, partial_addition,(void *) a1);
  pthread_create(&tid[1], NULL, partial_addition,(void *) a2);
  pthread_join(tid[0],NULL);
  pthread_join(tid[1],NULL);

 printf("The final sum is: %d\n", total);
}
```

# *Revisión de conceptos de programación concurrente*

## Condiciones de carrera




*El SO puede hacer un cambio de contexto entre dos instrucciones cualesquiera*

- **Solución:** ejecutar la sección crítica en ***exclusion mútua*** (ejecución ***atómica*** de operaciones "peligrosas")
- **Sección crítica:** *Segmento de código de un programa que accede a recursos compartidos con otros procesos, necesitándose un acceso exclusive al recurso.*

# *Revisión de conceptos de programación concurrente*

**Sección crítica**

```
void  partial_addition(int from, until) {
  for (i= from; i<=until; i++) {
    res=total; // get the value of total
    res+=i;    // calculate the new value for total
    total=res; // update the value of total
  }
}
```

# *Mecanismos de sincronización*

- Los primeros intentos de resolver el problema de la exclusion mútua se realizaron sin herramientas especiales, usando solo las estructuras de control tradicionales.
- Hay varios intentos de solución a este problema, cada uno de los cuales presenta diferentes inconvenientes (descritos en el libro de texto).
- Se encontraron diversas soluciones: algoritmos de Dekker, de Peterson, de Lamport, ...
- La solución no es evidente, y presenta siempre espera activa.
- Se desarrollaron algunas instrucciones hardware especiales (como test-and-set lock TSL), para facilitar el desarrollo de algoritmos de sincronización, pero siempre había espera activa.

*(páginas 120-123 del libro)*

- **Era esencial desarrollar herramientas de sincronización.**

# *Tabla de contenidos*

# *Papel del Sistema Operativo en la concurrencia*

- El Sistema operativo es en sí mismo un programa concurrente.
  - Mientras el SO está ejecutando algo, puede aparecer una interrupción y hacer que se ejecute otra cosa, dejando inconsistente, tal vez, alguna estructura de datos del SO.
- Una posible solución es inhabilitar las interrupciones mientras se está ejecutando el SO (mientras la CPU está modo privilegiado), haciendo todo el SO una sección crítica.
  - Pero esto no permitiría implementar “llamadas al sistema en 2 etapas”.
- Solución: implementar un *kernel re-entrante:* El SO puede ejecutar concurrentemente varios hilos de ejecución, llevando a cabo varias llamadas de distintos procesos.
- Esto hace el SO más complicado de implementar: todas las estructuras de datos críticas deben ser ejecutadas de manera atómica (deben estar dentro de una *sección crítica).*

# *Papel del Sistema Operativo en la concurrencia*

**¿Cómo se incorpora la sincronización en un programa?**

1. Desde el lenguaje usando **bibliotecas del lenguaje**
2. Usando directamente **llamadas al sistema operativo** para usar los mecanismos primitivos de sincronización del sistema

En último término es el sistema operativo el que proporciona mecanismos para

1. Sincronización
2. Comunicación

# *Papel del Sistema Operativo en la concurrencia*

**Mecanismos de Sincronización**

- Semáforos
- Mutex
- Variables condicionales
- Señales
- Regiones críticas condicionales (lenguaje)
- Monitores (lenguaje)

**Mecanismos de Communicación**

- Archivos
- Pipes
- Memoria compartida
- Paso de mensajes

# *Tabla de contenidos*

# *Mecanismos de sincronización*

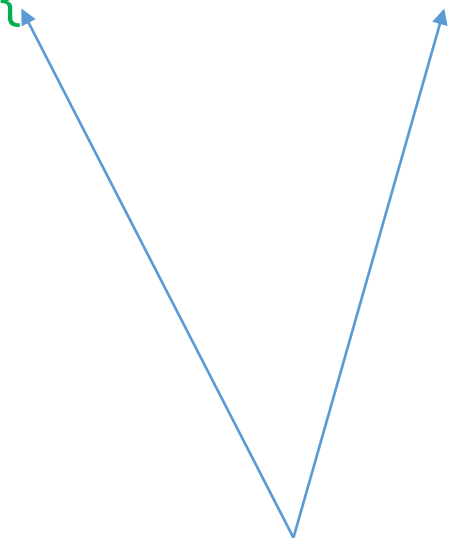
**Free=true;**

Thread 1

```
while (! free) {
  free=false;
  res=total;
  res+=i;
  total=res;
  free=true;
}
```

Thread 2

```
while (! free) {
  free=false;
  res=total;
  res+=i;
  total=res;
  free=true;
}
```



***Protocolo de entrada***

***Protocolo de salida***

**Espera activa**

# *Mecanismos de Sincronización*
## *Semáforos*

- Ideados por Dijkstra en 1965
- Estructura con tres operaciones atómicas:
  - Inicialización, *P, y V.*
- **Todas las operaciones son atómicas**.

*Normalmente se usan **wait(s)** y **signal (s)** en lugar de P(s) y V(s) respectivamente*

***No presentan espera activa.** El SO gestiona una cola de bloqueados*

```
wait(s)
{ s=s-1;
  if (s<0)
     Bloquear ;
}
```

```
signal(s)
{ s=s+1;
  if (s<=0)
     Despertar a un proceso
bloqueado sobre s;
}
```

***El Sistema Operativo implementa estas operaciones y ofrece los servicios***

# *Mecanismos de Sincronización*

## *Semáforos*

**Utilidad**

1. **Solución a la sección crítica en exclusión** mutua *(Inicializando a 1)*
2. **Limitación del número de procesos o hilos que acceden concurrentemente a una sección crítica** (control de las unidades de un recurso que están siendo utilizadas simultáneamente) *(Inicializando al número de recursos)*
3. **Sincronización**

```
S = 1;
```

```
wait(s);
   SECCION CRÍTICA
signal(s);
```

```
S = 4;
```

```
wait(s);
   Uso de recurso
signal(s);
```

```
S = 0;
```

| Hilo 1 | Hilo 2 |
|---|---|
| A = x + y | wait(s); |
| signal(s); | Print(A); |

# *Mecanismos de Sincronización*

## *Mutex*

- Similar a un semáforo inicializado a 1

- Se usa sólo para la exclusión mutua en secciones críticas

- Operaciones:
  - Lock() (como wait() en sem.)
  - Unlock() (como signal() )

```
lock(m);
   <sección crítica>
unlock(m);
```

```
lock(m);
 res=total;
 res+=i;
 total=res;
unlock(m);
```

# *Mecanismos de Sincronización*
## *Variables condicionales*

- Variable de sincronización asociada a un mutex
  - Se usa dentro de una sección crítica cuando necesitamos esperar por una condición
- Permite bloquear el hilo hasta que se produzca la condición
  - Cuando se bloquea porque no se cumple la condición, la variable mutex se libera
  - Cuando la condición se cumple los hilos que duermen sobre la cola de esa condición se despiertan.
    - Estos hilos compiten de nuevo por el mutex
- Operaciones:
  - Lock(), unlock() para el mutex,
  - c_wait(<variable>, <mutex>)
  - c_signal(<variable>)

# *Mecanismos de Sincronización*

## *Variables condicionales*

```
lock(m);
   <sección crítica>
   while condition == FALSE {
        c_wait(c, m)
   }
  <resto de sección crítica>
unlock(m);
```

```
lock(m);
   < sección crítica >
   condition = TRUE
   c_signal(c)
unlock(m);
```

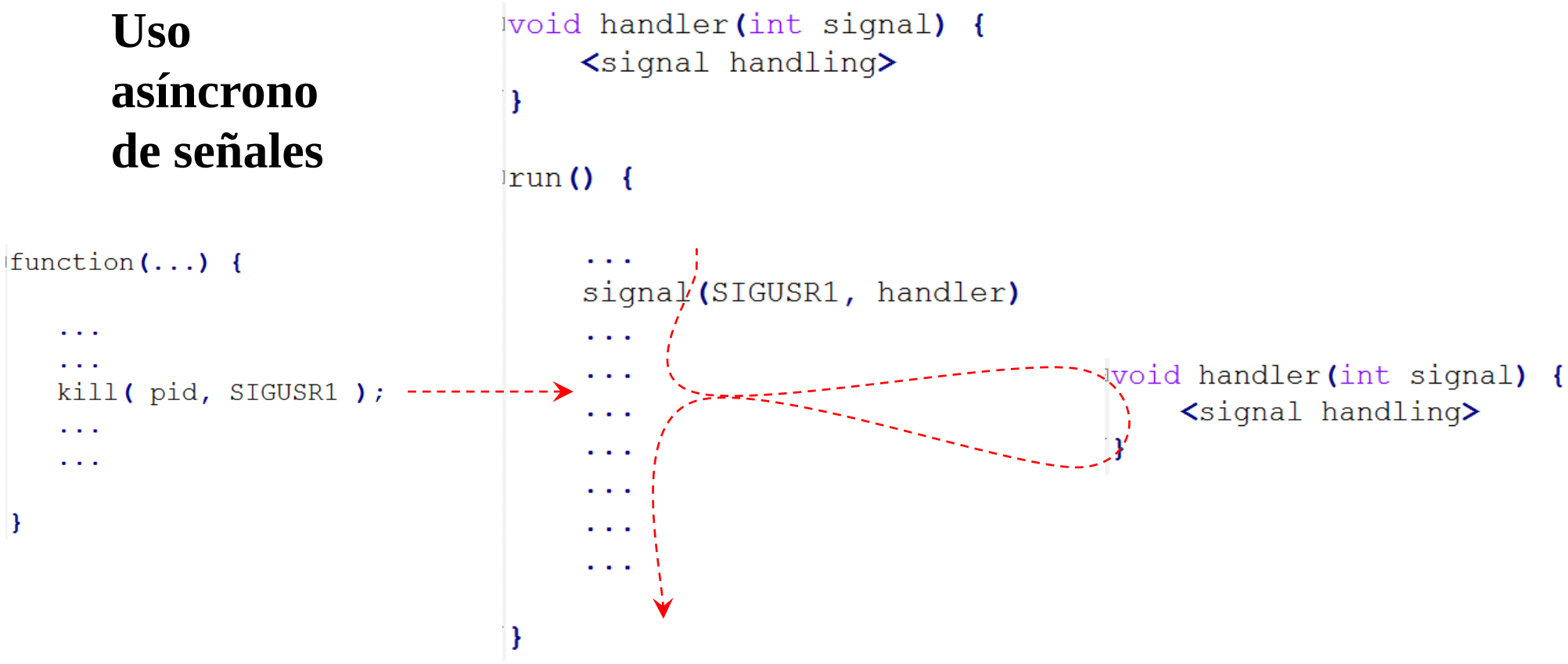
**Envío de señales entre procesos**

- Un proceso puede bloquearse (dormirse) hasta que reciba una señal (llamada ***pause***).
- Un proceso puede despertar a otro enviando una señal (***kill***).

**Consideraciones sobre señales**

- Un proceso puede recibir señales aunque no esté esperando por ellas (son asíncronas).
- Las señales no se encolan. Si hay una señal pendiente de tratar y se recibe otra del mismo tipo, sólo queda constancia de que ha llegado una señal.
- Las señales pueden producir condiciones de carrera.

**Uso asíncrono de señales**



# *Tabla de contenidos*

# *Mecanismos de Comunicación*

**Mecanismos de Sincronización**

- Semáforos
- Mutex
- Variables condicionales
- Señales

- Regiones críticas condicionales (lenguage)
- Monitores (lenguage)

**Mecanismos de Comunicación**

- Archivos
- Pipes
- Memoria compartida
- Paso de mensajes

# *Mecanismos de Comunicación*

## *Ficheros*

• Los ficheros pueden utilizarse para compartición de datos por parte de varios procesos.

• Son fáciles de usar, al basarse la comunicación en las operaciones de leer y escribir.

• Pueden comunicar un número potencialmente ilimitado de procesos.

• Es poco eficiente, al ser operaciones lentas.

# *Mecanismos de Comunicación*

## *Tuberías*

• Es un mecanismo tanto de comunicación como de sincronización usado en el estándar Posix

• Se utilizan las mismas llamadas para leer y escribir que en ficheros. Ambas operaciones son atómicas.

• Es un fichero FIFO: cuando se lee un dato, se lee el que más tiempo lleva en la tubería **y este dato desaparece** de la tubería.

• Si un proceso intenta leer de una tubería **vacía**, el S.O. **duerme al proceso** hasta que haya datos.

• Si la tubería está **llena** y se intenta escribir, el proceso se **dormirá** hasta que haya sitio.

• Puede ser utilizado por múltiples procesos, tanto leyendo o escribiendo.

# *Mecanismos de Comunicación*

## *Paso de mensajes*

- Para comunicación entre procesos de *distintas máquinas*
- Los procesos envían y reciben mensajes para comunicarse y sincronizarse.
- **Una implementación que se utiliza mucho son los Sockets**. Los más utilizados en aplicaciones distribuidas
- Se utilizan dos operaciones básicas :
  - *send(destino, mensaje)*
  - *receive(origen, mensaje)*

Tipos de comunicación:

- **Síncrona total** (envío y recepción bloqueante): el emisor se duerme hasta que el receptor recibe el mensaje. El receptor se duerme si el mensaje no ha llegado.
- **Síncrona intermedia** (envío no bloqueante y recepción bloqueante): el emisor no se duerme pero el receptor se bloquea hasta que recibe el mensaje.
- **Asíncrona** (envío y recepción no bloqueante): Nadie espera.

**Modos de comunicación**

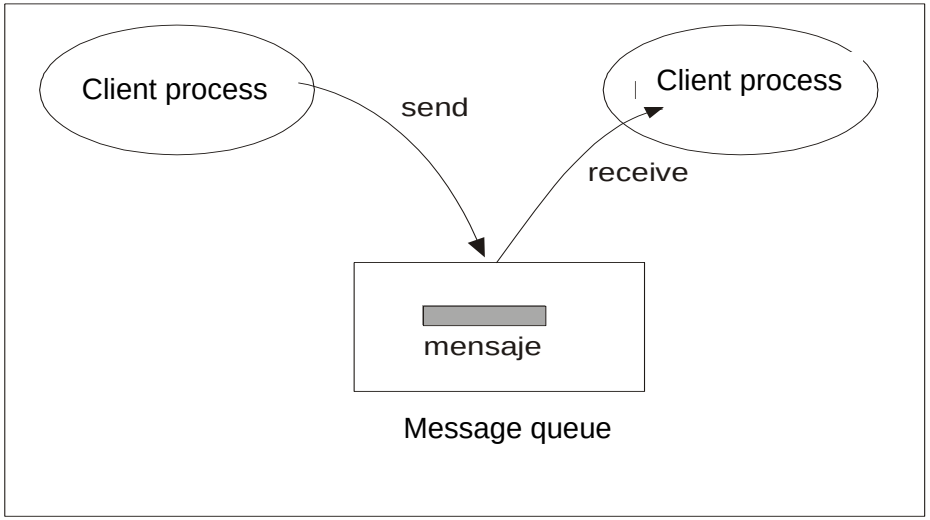
1. **Comunicación directa.** Se envía el mensaje al proceso
2. **Comunicación indirecta.** Se envía el mensaje a una estructura de la que el proceso recoge la información
    - **Colas de mensajes o buzones**
    - **Puertos**

**Colas de mensajes.** El destino y el origen identifican una entidad intermedia: un buzón. Puede haber múltiples emisores y múltiples receptores.

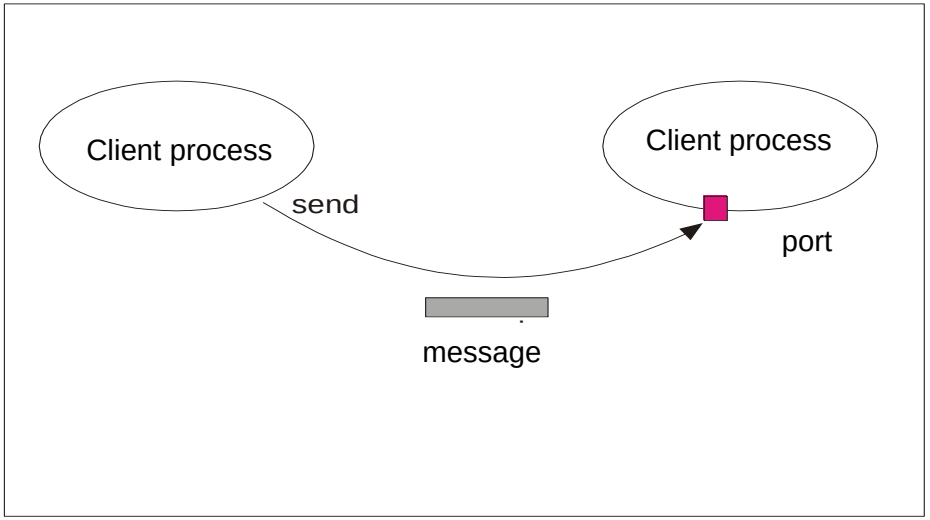
**Puertos.** Caso específico de un buzón, donde sólo hay un receptor, pudiendo haber múltiples emisores. El puerto suele pertenecer al proceso receptor. **Los Sockets son un ejemplo de ello**.

# *Mecanismos de Comunicación*

## *Paso de mensajes*




Comunicación con colas de mensajes




Comunicación con puertos

# *Mecanismos de Comunicación*
## *Memoria compartida*

• La memoria compartida se utiliza para comunicar (no para sincronizar) procesos dentro de la misma máquina.

• El sistema operativo permite a varios procesos acceder a la misma zona de memoria a través de llamadas al sistema específicas de creación de memoria compartida.

• Los procesos que comparten memoria pueden utilizar esa zona para dejar datos que deben ser accesibles por todos ellos.

• En un sistema basado en *hilos* todos los hilos del mismo proceso comparten la memoria sin necesidad de intervención del S.O.

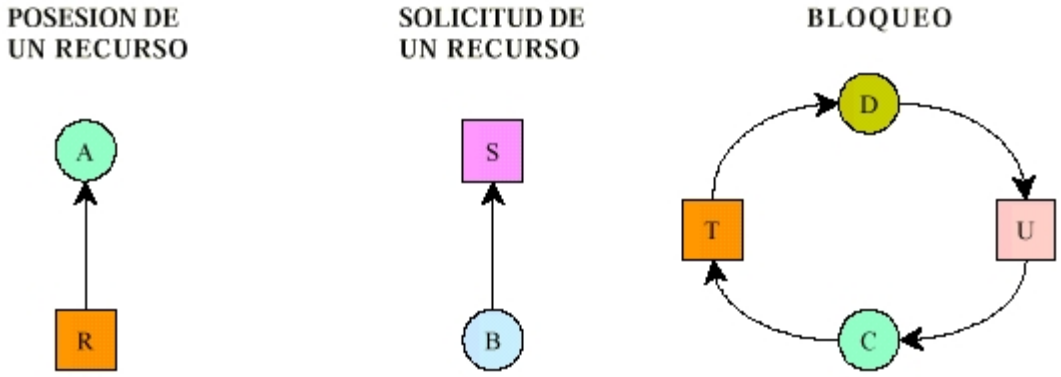
# *Tabla de contenidos*

## Problemas que presenta la concurrencia

1. ***Condiciones de carrera:*** El resultado final de la ejecución de varios procesos concurrentes depende de la secuencia de ejecución.
2. ***Interbloqueo***: bloqueo permanente de varios procesos que compiten por recursos o se sincronizan entre sí.

## Interbloqueo

Es una anomalía que se produce cuando un proceso se encuentra esperando a un suceso que no puede ocurrir.




Figura~6.2: Gráficas de asignación de recursos.

• La causa fundamental es el uso exclusivo de recursos compartidos por varios procesos. En el interbloqueo suele haber implicados *conjuntos de procesos o hilos.*

• No hay que confundirlo con *inanición* o *aplazamiento indefinido*. Este problema se da por una mala política de asignación de recursos.

## De un solo proceso

1. El proceso espera por un evento que no va a ocurrir. Por ejemplo, *dormir hasta que sean las 12.00 del 1 de enero de 2000.*

## De dos procesos

| Proceso 1 |
| --- |
| Solicitar GrabadoraDVD |
| Solicitar Impresora |
| USAR |
| Liberar Impresora |
| Liberar GrabadoraDVD |

| Proceso 2 |
| --- |
| Solicitar Impresora |
| Solicitar GrabadoraDVD |
| USAR |
| Liberar GrabadoraDVD |
| Liberar Impresora |

**De dos procesos.** Sea un sistema con 200 KB de memoria

| Proceso 1 |
| --- |
| Solicitar 80 KB |
| … |
| Solicitar 60 KB |
| … |
| Liberar 140 KB |

| Proceso 2 |
| --- |
| Solicitar 70 KB |
| … |
| Solicitar 80 KB |
| … |
| Liberar 150 KB |

• Los procesos pueden quedar bloqueados si se atiende la primera petición de cada proceso.

# *Tratamiento del interbloqueo*

Hay cuatro estrategias para tratar con el interbloqueo:

1. Ignorar el problema.
2. Detección y recuperación
3. “Evitación” (predicción), mediante una cuidadosa asignación de recursos.
4. Prevención, negando una de las condiciones para que se produzca.

Un estudio detallado del tema puede encontrarse en el capítulo 6 del libro.