

# Sincronización

Gustavo Romero

Arquitectura y Tecnología de Computadores

19 de octubre de 2010

# Índice

## 1 Introducción

## 2 Problemas

## 3 Señales

- Definición
- Asíncronas
- Síncronas
- Semáforos
- UNIX

# Lecturas recomendadas

- J. Bacon      Operating Systems (9, 10, 11)
- A. Silberschatz      Fundamentos de Sistemas Operativos (4, 6, 7)
- W. Stallings      Sistemas Operativos (5, 6)
- A. Tanuenbaum      Sistemas Operativos Modernos (2)

# Introducción

# Concurrencia (1)

- Niveles:
  - **Multiprogramación:** gestión de múltiples procesos.
  - **Multiprocesamiento:** gestión de múltiples hebras dentro de un proceso.
  - **Procesamiento distribuido:** gestión de múltiples procesos sobre múltiples máquinas (sin memoria compartida).
- Contexto:
  - Entre aplicaciones: originalmente ideada con este fin.
  - Dentro de una aplicación: conveniente en ciertos tipos de aplicaciones.
  - Sistema operativo: las mismas ventajas que aporta a las aplicaciones son deseables aquí.
- Tipos:
  - **Monoprocesador:** los procesos se **entrelazan** en el tiempo para ofrecer la apariencia de ejecución simultánea.
  - **Multiprocesador:** la ejecución de múltiples procesos se **solapa** en el tiempo.

# Concurrencia (2)

- No puede predecirse la velocidad relativa de ejecución:
  - Eventos.
  - Políticas del sistema operativo.
- Problemas:
  - La **compartición** de recursos está plagada de peligros.
  - La **gestión** óptima de los recursos es complicada.
  - La localización de **errores** de programación es muy complicada debido a su naturaleza no determinista y no reproducible.
- Todos los sistemas comparten este tipo de problemas aunque algunos pueden verse agravados en máquinas con más de un procesador.

# Conceptos **importantes**

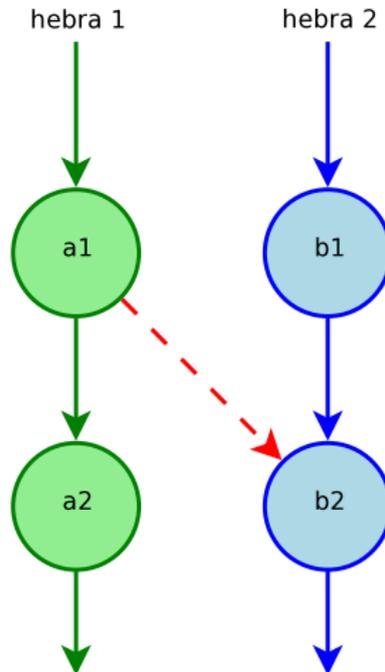
- **Condición de carrera:** situación en la que el resultado de una operación depende del orden de ejecución de varias hebras.
- **Sección crítica:** sección de código que no pueden ejecutar varias hebras de forma concurrente.
- **Exclusión mutua:** requisito de acceso exclusivo a recursos durante la ejecución de una sección crítica.
- **Interbloqueo** (*“deadlock”*): situación en la que varias hebras son incapaces de avanzar porque todas ellas están esperando que alguna de las otras haga algo.
- **Inanición:** situación en la que una hebra preparada es mantenida en dicho estado indefinidamente.
- **Circulo vicioso** (*“livelock”*): situación en la que varias hebras cambian su estado pero sin llegar a realizar trabajo útil.

# ¿Por qué sincronizar?

Las hebras **cooperan** porque...

- Necesitan **coordinar** su ejecución.
  - El botón de detener del navegador cancela la descarga de una página web.
  - La hebra del interfaz del navegador necesita avisar a la hebra encargada de descargar la página web.
- Necesitan acceder a **recursos compartidos**.
  - Varias hebras de un servidor web pueden acceder a una caché común en memoria para ahorrar espacio y tiempo.

# Coordinación



b2 tiene que esperar hasta que a1 haya completado su ejecución.

# Recursos compartidos

- Problema básico:
  - 2 hebras acceden a una variable compartida.
  - Si la variable compartida es **leída y escrita** por ambas hebras tendremos que coordinar el acceso.
- En los próximos 2 temas estudiaremos...
  - Mecanismos para controlar el acceso a recursos compartidos:
    - Bajo nivel: cerrojos.
    - Alto nivel: semáforos, monitores, variables condición.
  - Protocolos para coordinar el acceso a recursos compartidos.
- La gestión de la concurrencia es complicada y propensa a errores.
  - En Informática se dedica una asignatura completa a su estudio.
  - Incluso las soluciones de algunos libros de texto son erróneas.

# Ejemplo: cuenta bancaria (1)

procedimiento para **retirar** fondos de una cuenta bancaria

```
int retirar(CCC_t cuenta, int cantidad)
{
    int saldo = conseguir_saldo(cuenta);
    saldo = saldo - cantidad;
    almacenar_saldo(cuenta, saldo);
    return saldo;
}
```

procedimiento para **ingresar** fondos en una cuenta bancaria

```
int ingresar(CCC_t cuenta, int cantidad)
{
    int saldo = conseguir_saldo(cuenta);
    saldo = saldo + cantidad;
    almacenar_saldo(cuenta, saldo);
    return saldo;
}
```

## Ejemplo: cuenta bancaria (2)

- Supongamos un saldo inicial de 1.000€.
- Estudiaremos el caso en que dos personas intentan retirar 100€ a la vez de la misma cuenta desde dos cajeros automáticos.
- Imaginemos que cada procedimiento se ejecuta como una hebra diferente.
- La ejecución de las 2 hebras podría entrelazarse debido a la planificación del sistema operativo.
- ¿Existe algún problema?
- ¿Cuál será el saldo de la cuenta tras retirar fondos?

# Ejemplo: cuenta bancaria (3)

hebra 1	hebra 2
saldo = conseguir_saldo(cuenta);	
saldo = saldo - cantidad;	
almacenar_saldo(cuenta, saldo);	
return saldo;	
	saldo = conseguir_saldo(cuenta);
	saldo = saldo - cantidad;
	almacenar_saldo(cuenta, saldo);
	return saldo;

**saldo final: 800€**

## Ejemplo: cuenta bancaria (4)

hebra 1	hebra 2
saldo = conseguir_saldo(cuenta); saldo = saldo - cantidad;	
	saldo = conseguir_saldo(cuenta); saldo = saldo - cantidad;
	almacenar_saldo(cuenta, saldo);
	return saldo;
almacenar_saldo(cuenta, saldo); return saldo;	

**saldo final: 900€**

# Condiciones de carrera

El problema ocurre porque dos hebras acceden a un **recurso compartido** sin **sincronizar** su uso.

- Las condiciones de carrera producen resultados **impredecibles**.
- Dependen de cuándo se ejecutan las hebras, durante cuánto tiempo y de cuándo se produce el cambio entre ellas, y por supuesto, de lo que estén haciendo.

El acceso concurrente a recursos compartidos debe ser controlado.

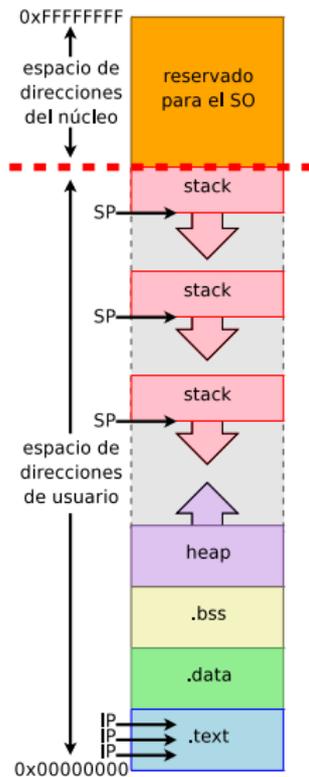
- Es la única forma de que podamos comprender el comportamiento de los programas.
- Necesitamos reintroducir el **determinismo** y la **predecibilidad**.

# Operaciones atómicas

- Para comprender un programa concurrente debemos conocer qué **operaciones** deberían ser **indivisibles**.
- **Operación atómica**: una operación que siempre se ejecuta hasta completarse o que por el contrario no inicia su ejecución.
  - Es indivisible: no puede ser detenida a mitad y luego continuar.
  - Son la base fundamental sobre la que se crean programas multihebra y sin las que no habría forma de hacer que funcionasen de manera correcta.
- En la mayoría de los procesadores las operaciones de lectura y escritura de una palabra de memoria son atómicas.
- Sin embargo existen muchas instrucciones que no son atómicas:
  - El almacenamiento de reales grandes puede no serlo.
  - Las instrucciones de sobre cadenas o vectores.

# ¿Qué recursos son compartidos?

- Las **variables locales** de los procedimientos **no** son compartidas.
  - Existen en la pila.
  - La pila es privada para cada hebra.
  - No podemos pasar un puntero a una variable local a otra hebra ¿Por qué?
- Las **variables globales si** son compartidas.
  - Se almacenan en la zona de datos estáticos.
  - Son accesibles por todas las hebras.
- Los **datos alojados dinámicamente si** son compartidos:
  - Se almacenan en el cima/montón ("*heap*").
  - Son accesibles por todas las hebras.
- ¿Hay algún problema en utilizar punteros a objetos compartidos? ↔ ¿dónde se almacenan?



# Ejemplo: recursos compartidos

- ¿Qué pasará al ejecutar dos copias de esta hebra?

```
void* saludo(void*)  
{  
    while(true)  
        cout << "hola, soy " << pthread_self() << endl;  
    return NULL;  
}
```

- esto...

```
hola, soy 918374512  
hola, soy 649128354
```

- o esto...

```
hola, soy hola, soy 918374512  
  
649128354
```

- Condición de carrera: ambas hebras acceden al terminal que es un recurso compartido.

# Ejemplo: datos compartidos (1)

```
int a = 0, b = 0; // variables compartidas
```

```
void *suma(void*)  
{  
    while(true)  
    {  
        a = a + 1;  
        b = b + 1;  
    }  
    return NULL;  
}
```

```
void *resta(void*)  
{  
    while(true)  
    {  
        a = a - 1;  
        b = b - 1;  
    }  
    return NULL;  
}
```

- Tras un tiempo... ¿seguirán a y b siendo iguales?

## Ejemplo: datos compartidos (2)

```
int a = 0, b = 0; // variables compartidas
```

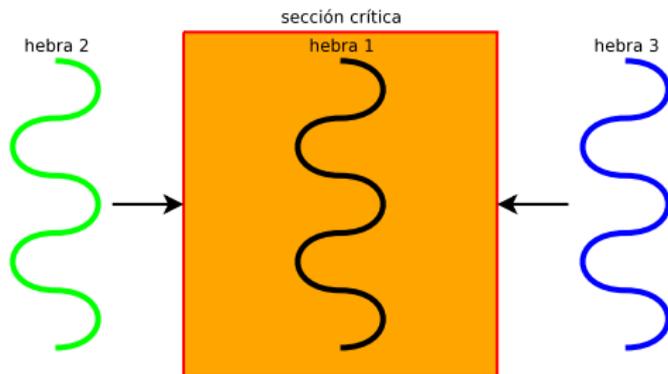
```
void *suma(void*)
{
    while(true)
    {
        a = a + 1;
        b = b + 1;
    }
    return NULL;
}
```

```
void *resta(void*)
{
    while(true)
    {
        a = a - 1;
        b = b - 1;
    }
    return NULL;
}
```

- El código que accede a variables compartidas son **secciones críticas** → tras cierto tiempo  $a \neq b$
- Compartir datos y recursos induce problemas similares.
- Problemas: ¿CFLAGS = -O3? ¿volatile?

# Exclusión mutua

- Debemos utilizar **exclusión mutua** para sincronizar el acceso a los recursos compartidos, por ejemplo, **serializando** el acceso.
- Cualquier código que utilice exclusión mutua para sincronizar su ejecución es denominado **sección crítica**.
  - Sólo una hebra puede ejecutar su sección crítica.
  - El resto de hebras son forzadas a esperar a la entrada.
  - Cuando una hebra abandona su sección crítica otra puede entrar.

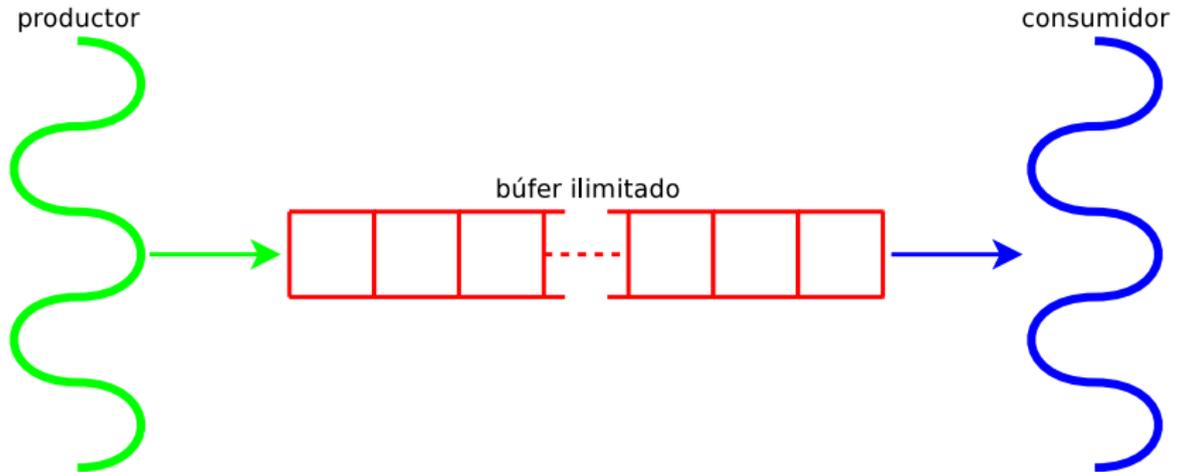


# Interacción entre hebras

percepción	relación	influencia	problemas
se desconocen	competencia	resultados independientes, temporización afectable	interbloqueo, inanición
percepción indirecta (recursos compartidos)	cooperación por compartición	resultados dependientes, temporización afectable	exclusión mutua, interbloqueo, inanición, coherencia de datos
percepción directa (comunicación)	cooperación por comunicación	resultados dependientes, temporización afectable	interbloqueo, inanición

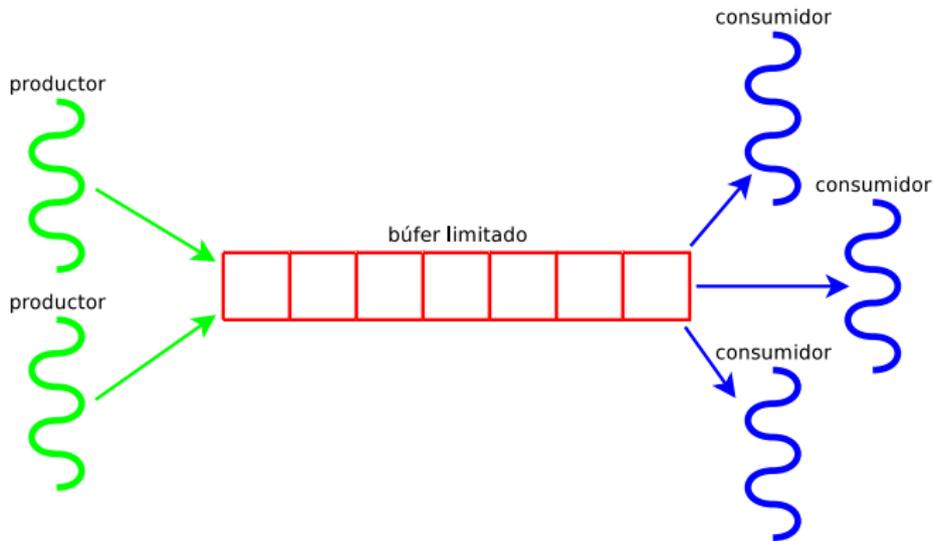
# Problemas

# Productor/consumidor con búfer ilimitado



- Dos procesos repiten indefinidamente su trabajo: el productor genera datos que introduce en un búfer. El consumidor recupera los datos y hace algo con ellos.
- ¿Hay un problema de concurrencia?  $\implies$  si, el consumidor no puede consumir desde un búfer vacío.

# Productor/consumidor con búfer limitado



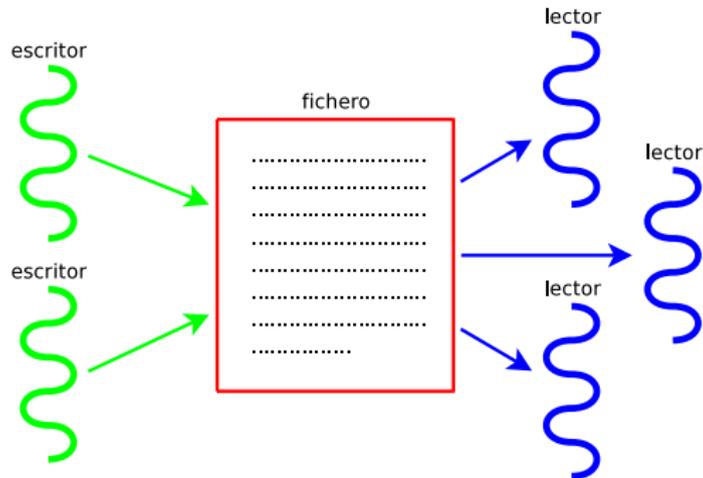
- ¿Aparecen nuevos problemas si el búfer no es infinito?  $\implies$  si, el productor no puede introducir datos en un búfer lleno.
- Si hay más de un productor y/o consumidor  $\implies$  Deben competir por el uso del búfer con sus iguales.

# El problema del barbero durmiente

- En una barbería trabaja un barbero que tiene un único sillón de trabajo y varias sillas para esperar. Cuando no hay clientes, el barbero se sienta en una silla y se duerme. Cuando llega un nuevo cliente, éste o bien despierta al barbero o, si el barbero está afeitando a otro cliente, se sienta en una silla. Si todas las sillas están ocupadas por clientes esperando se va.



# El problema de los lectores/escritores



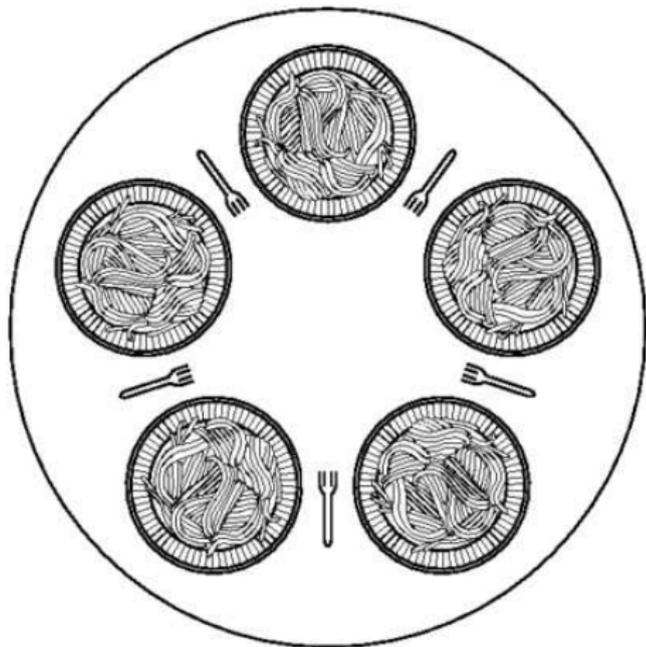
- Cierta número de hebras acceden a memoria, unas para leer y otras para escribir. Mientras una escriba ninguna otra podrá acceder. Sí se permite a varias hebras leer simultáneamente.
- ¿Problemas? → Inconsistencia y competición.
- 3 versiones: prioridad lectores, prioridad escritores y justa.

# El problema de la cena de los filósofos

## Vida de un filósofo

```
while(true)
{
    pensar();
    coger_tenedores();
    comer();
}
```

- Requisitos:
  - Evitar el intebloqueo.
  - Evitar la inanición.



# Niveles de sincronización

programas	procesos/hebras cooperativos
API de alto nivel	cerrojos, semáforos, monitores, paso de mensajes
hardware	load&store, deshabilitar interrupciones, test&set, compare&swap

A continuación veremos cómo implementar varias primitivas de sincronización de alto nivel:

- Si las únicas operaciones atómicas disponibles son la carga y el almacenamiento será más complicado.
- Necesitamos primitivas útiles a nivel de usuario.

# Señales

# Primero las señales, después la exclusión mutua

- Los dos mecanismos se diferencian sólo ligeramente.
- Debemos intentar no mezclarlos.
- El uso habitual de las señales es...
  - Una hebra desea informar a...
    - otra hebra o
    - cualquier hebra o conjunto de ellas
  - de que ha alcanzado cierto punto de su código (IP = valor).
- Ejemplo: un jefe de estación tocando el silbato.

# Señales

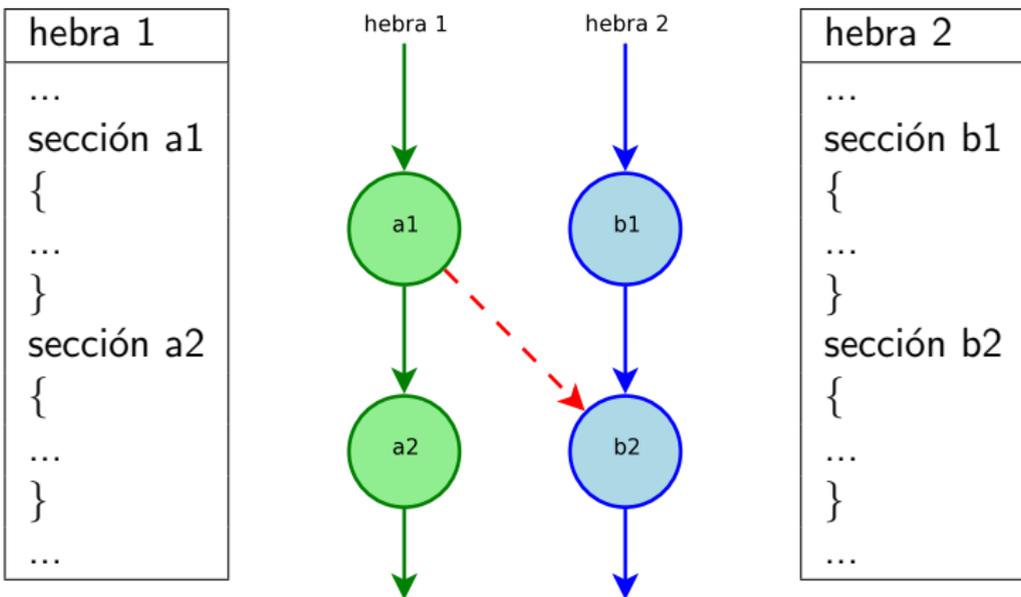
Definición: mensaje entre hebras que puede provocar la ejecución de un procedimiento en el receptor.

## Tipos:

- Bandera:
  - Valor binario: 0/1, si/no,...
  - El significado de la bandera debe ser acordado entre las hebras en comunicación.
- Contador:
  - Cada valor puede tener un significado diferente o reflejar el número de señales pendientes.

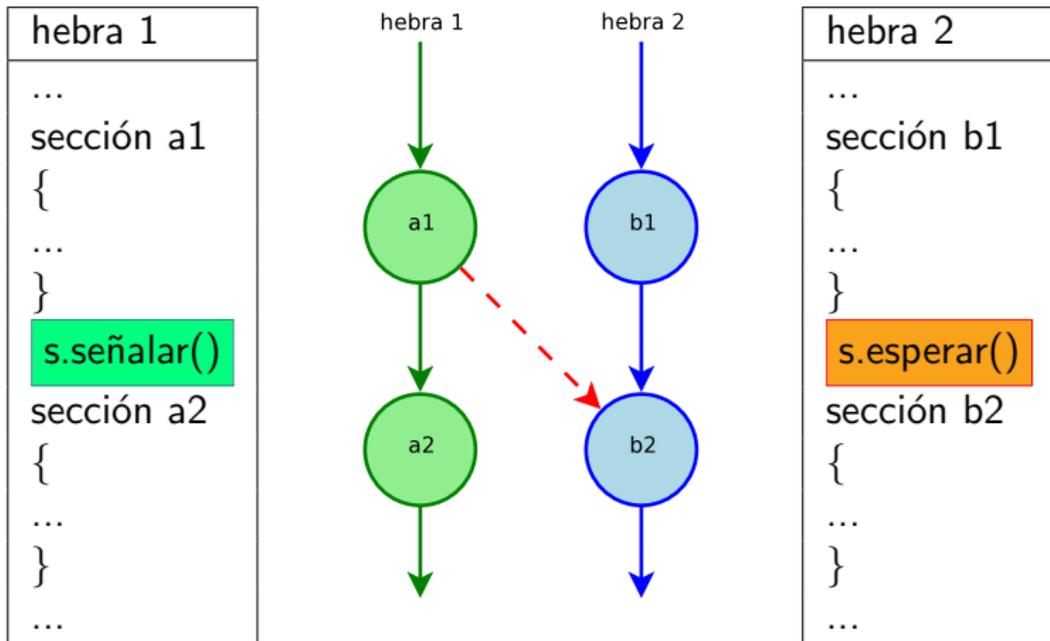
¿Cuándo es una bandera suficiente? ¿Cuándo necesitamos un contador?

# Relación de precedencia (1)



¿Cómo asegurar que la sección a1 se ejecuta antes que b2? → la sección b2 debe esperar hasta que a1 se haya ejecutado.

# Relación de precedencia (2)



¿Cómo implementar este objeto señal 1:1 (emisor/receptor)?

# Tipos de soluciones

- Soluciones a nivel de aplicación:
  - No requiere instrucciones especiales del procesador ni servicios del núcleo.
  - Implementado como funciones de biblioteca o dentro de la propia aplicación.
- Soluciones a nivel del núcleo:
  - Proporcionar llamadas al sistema que cumplan estas funciones.
- Soluciones hardware:
  - Instrucciones especiales del procesador.
  - Garantizan la ejecución atómica.
- La mayoría de los sistemas sólo proporcionan un subconjunto de estas soluciones.

# Señales en el problema productor/consumidor

Problemas que podremos resolver empleando señales:

- El productor debe **avisar** al consumidor para que pueda utilizar el objeto que acaba de introducir en el búfer.
- El consumidor debe **avisar** al productor de que ha consumido un objeto del búfer y por lo tanto ha quedado un hueco libre.
- El productor debe **esperar** cuando el búfer está lleno.
- El consumidor debe **esperar** cuando el búfer está vacío.

# Solución al problema productor/consumidor (v1)

```
const int N = 10; // tamaño del búfer
int búfer[N];    // almacén de datos
int contador = 0; // posición a producir/consumir
```

```
void* productor(void*)
{
    while (true)
    {
        while (contador == N);
        búfer[contador] = producir();
        contador = contador + 1;
    }
}
```

```
void* consumidor(void*)
{
    while (true)
    {
        while (contador == 0);
        contador = contador - 1;
        consumir(búfer[contador]);
    }
}
```

- ¿Funciona? → No, condición de carrera en la variable compartida contador.

# Solución al problema productor/consumidor (v2)

```
const int N = 10; // número de elementos
int búfer[N];    // almacén de datos
int entra = 0;   // posición para introducir
int sale  = 0;   // posición para sacar
```

```
void* productor(void*)
{
    while (true)
    {
        while ((entra + 1) % N == sale);
        búfer[entra] = producir();
        entra = (entra + 1) % N;
    }
}
```

```
void* consumidor(void*)
{
    while (true)
    {
        while (entra == sale);
        consumir(búfer[sale]);
        sale = (sale + 1) % N;
    }
}
```

- ¿Funciona? ¿es eficiente? ¿es escalable?
- ¿Podemos utilizar todos los elementos del búfer? → No, sólo  $N - 1$ .

# Escalabilidad

¿Cuándo se dice que una solución del problema productor/consumidor es escalable?

- Cuando funciona de forma efectiva para:
  - $p \geq 1$  productores.
  - $c \geq 1$  consumidores.
  - un búfer de  $0 < N < MAXINT$  elementos.
- La solución anterior no es escalable porque...
  - Sólo parcialmente correcto para  $p = 1$  y  $c = 1$ .
  - Incorrecto para  $N = 1$ .

# Eficiencia

¿Cuándo se dice que una solución del problema productor/consumidor es eficiente?

- Cuando funciona de forma rápida y con poca sobrecarga.
- La solución anterior no es eficiente porque...
  - La sincronización mediante **espera ocupada** desperdicia el tiempo del procesador.
  - El planificador puede prevenir la ineficiente espera ocupada.
  - Sólo es útil si conocemos que la espera ocupada va a ser corta.
- ¿Cómo mejorar esta solución?
  - Utilizar la API del núcleo para evitar la espera ocupada.

# Interfaz del núcleo para evitar la espera ocupada

- **block()**

- `sleep()` → bloquear/dormir de UNIX.

- **unblock()**

- `wakeup()` → desbloquear/despertar de UNIX.

- **yield()**

- `pthread_yield()` → ceder el procesador entre hebras.
- `sched_yield()` → ceder el procesador entre procesos UNIX.

# block() → bloquear() del núcleo

```
bloquear(hebra_actual)
{
    hebra_actual.estado = bloqueado ;
    siguiente_hebra = planificador();
    hebra_actual = cambio_de_hebra(siguiente_hebra);
    hebra_actual.estado = ejecutando ;
}
```

```
desbloquear(una_hebra)
{
    if (una_hebra.estado == bloqueado )
        una_hebra.estado = preparado ;
}
```

- Solución para el modelo de 3 estados.
- ¿Cómo implementaría yield? → modificando bloquear...

## API del núcleo aplicada al problema productor/consumidor

```
const int N = 100; // tamaño del búfer
int búfer[N];      // almacén
int contador = 0; // número de elementos en el búfer
```

```
void* productor(void*)
{
    while (true)
    {
        if (contador == N)
            bloquear(productor);
        búfer[contador++] = producir();
        desbloquear(consumidor);
    }
}
```

```
void* consumidor(void*)
{
    while (true)
    {
        if (contador == 0)
            bloquear(consumidor);
        consumir(búfer[--contador]);
        desbloquear(productor);
    }
}
```

- Problemas: condición de carrera (contador) e interbloqueo <sup>1</sup>.
- Mejora la eficiencia: deja de utilizar espera ocupada.

<sup>1</sup>contador == 0 + if del consumidor + N ejecuciones de productor

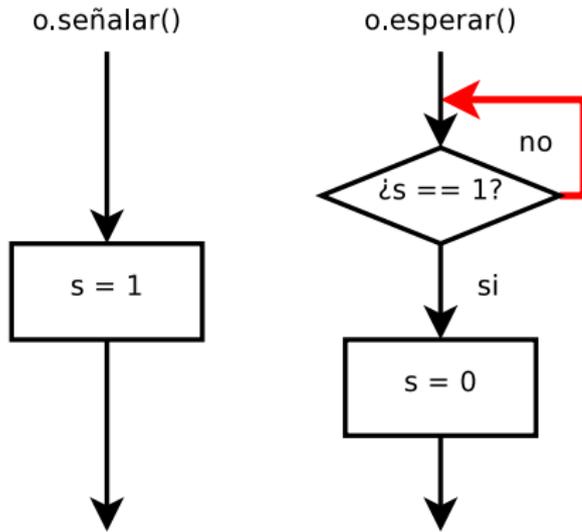
# En resumen... la API del núcleo puede ser insuficiente

- El uso del API del núcleo puede ser peligroso.
  - Ver la condición de carrera del ejemplo anterior.
- Sería mejor utilizar objetos señal u objetos sincronización con métodos apropiados.
- Hasta ahora hemos intentado enviar señales **sin éxito** mediante...
  - **espera ocupada** a nivel de usuario.
  - **bloqueo** a nivel del núcleo.

# Objeto señal

implementación a nivel de usuario

Implementación que emplea una variable lógica `s` compartida entre hebras.



¿Qué podría suceder si `o.señalar()` es invocado varias veces antes que `o.esperar()`?

→ se pierden señales.

¿Qué podría suceder si `o.esperar()` es invocado antes que `o.señalar()`?

→ se debería bloquear la hebra.

- ¿Funcionaría en cualquier tipo de sistema? → no, exige memoria compartida.
- ¿Es eficiente y escalable?

# Objeto señal: implementación a nivel usuario (v1)

- ¿Pierde señales?
- ¿Eficiente?
- ¿Escalable?

```
class señal
{
public:
    señal(): activa(false) {}

    void señalar() { activa = true; }

    void esperar()
    {
        while (activa == false);
        activa = false;
    }

private:
    bool activa;
};
```

# Objeto señal: implementación a nivel usuario (v2)

- ¿Pierde señales?
- ¿Eficiente?
- ¿Escalable?
- Parecido a la espera ocupada.

```
class señal
{
public:
    señal(): activa(false) {} // inicialización

    void señalar()
    {
        activa = true;
        yield();           // exige cooperación
    }

    void esperar()
    {
        while (activa == false)
            yield();       // evita espera ocupada
        activa = false;
    }

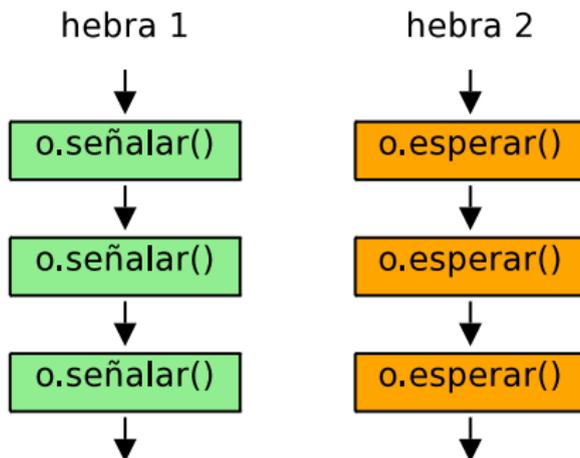
private:
    bool activa;
};
```

# ¿Son escalables estas soluciones? (1)

- ¿Puede más de una hebra señalar y/o esperar?
  - No, de otra forma ya no sería un objeto señal 1:1 sino n:m con  $n > 1$ ,  $m > 1$  o ambos.
- Múltiples usos de un mismo objeto señal dentro del un mismo par de hebras.
  - No es efectivo ni eficiente.
- Múltiples objetos señal dentro del un mismo par de hebras.
  - No es eficiente.

# ¿Son escalables estas soluciones? (2)

- Múltiples usos de un mismo objeto señal dentro del un mismo par de hebras.
  - No es efectivo ni eficiente ↔ ¿por qué?



# Objeto señal: implementación a nivel del núcleo (v3)

- ¿Pierde señales?
- ¿Eficiente?
- ¿Escalable?

```
class señal
{
public:
    señal(): bloqueada(ninguna) {}

    void señalar()
    {
        desbloquear(bloqueada);
    }

    void esperar()
    {
        bloqueada = esta;
        bloquear(bloqueada);
        bloqueada = ninguna; // ¿necesario?
    }

private:
    hebra bloqueada;
};
```

# Objeto señal: implementación a nivel del núcleo (v4)

```

class señal
{
private:
    bool activa;
    hebra bloqueada;

public:
    señal():
        activa(false),
        bloqueada(ninguna) {}

    void señalar()
    {
        activa = true;
        if (bloqueada != ninguna)
            desbloquear(bloqueada);
    }
}

```

```

void esperar()
{
    while(activa == false)
    {
        1   bloqueada = esta;
        3   bloquear(esta);
    }
    activa = false;
}
};

```

- ¡Pierde señales?
- ¡Eficiente!
- ¡Escalable!
- ¡Condición de carrera!

# Resumen

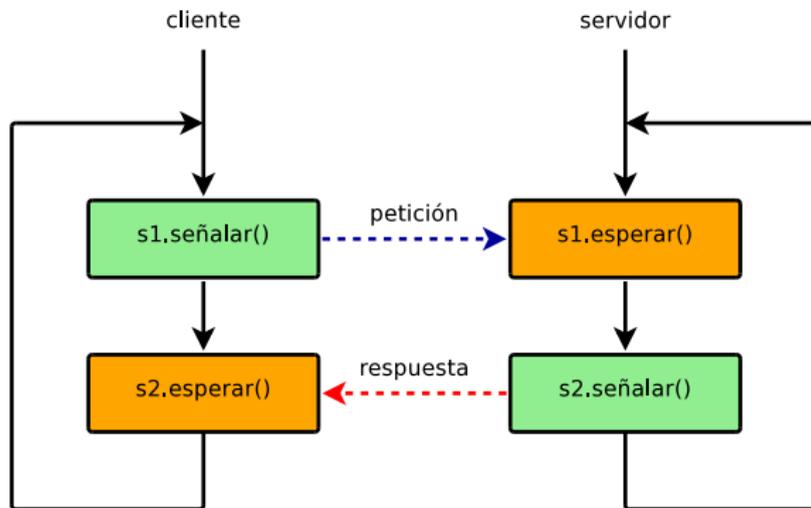
## Problemas aún sin resolver:

- Las operaciones señalar() y esperar() deberían ser **atómicas**.
- Evitar la **espera ocupada** a nivel de usuario.

## Sobre las soluciones vistas...

- Todas pueden perder señales.
  - Repetidas operaciones señalar() pueden sobrescribir señales todavía no consumidas.
- La operación señalar() es **asíncrona** (no bloqueante) en caso de que no haya otra hebra bloqueada mediante esperar().
  - Existe el peligro de **desbordar** a otra hebra (servidor).

# Reconocimiento para evitar el desbordamiento



Observaciones:

- `señalar()` es **asíncrono**.
- ¿Hay alguna solución más obvia? → sincronía.
- ¿Protege frente a clientes maliciosos? → no.

# Objeto señal síncrona

```

class señal_síncrona
{
public:
    señal_síncrona():
        activa(false),
        señaladora(ninguna),
        esperadora(ninguna) {}

    void señalar()
    {
        activa = true;
        if (esperadora != ninguna)
            desbloquear(esperadora);
        else
        {
            señaladora = esta;
            bloquear(señaladora);
            señaladora = ninguna;
        }
    }
}

void esperar()
{
    while(activa == false)
    {
        1  esperadora = esta;
        3  bloquear(esperadora);
           esperadora = ninguna;
    }
    activa = false;
    desbloquear(señaladora)
}

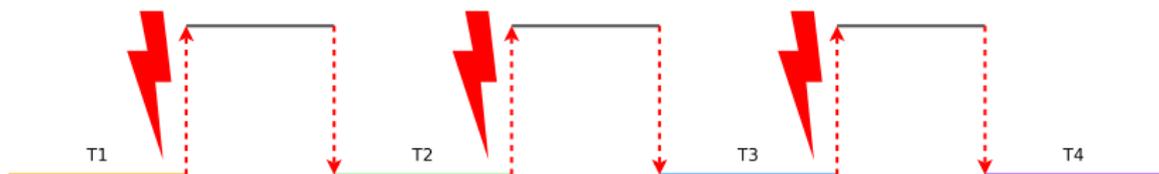
private:
    bool activa;
    hebra señaladora;
    hebra esperadora;
};

```

# ¿Son las banderas superfluas?

... dado que existen soluciones más potentes

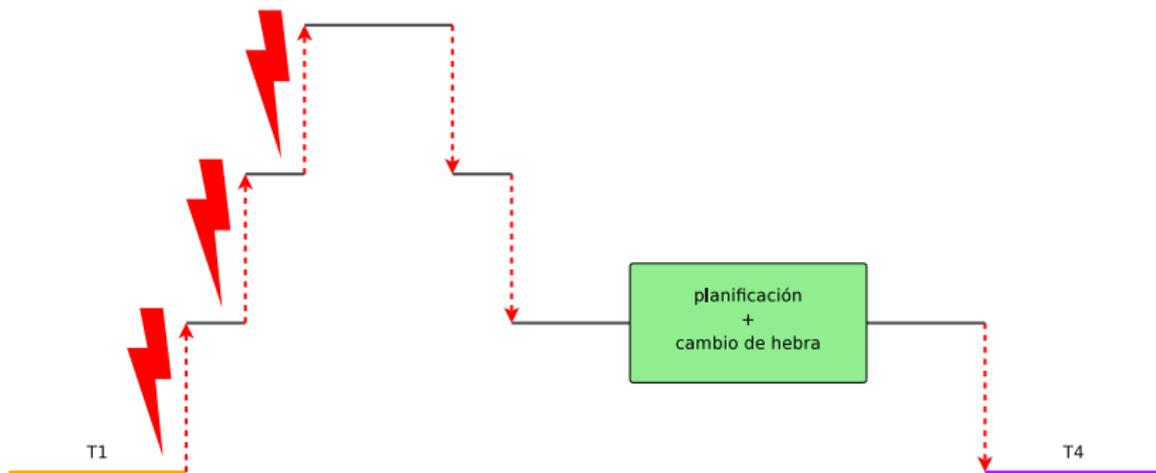
- **No.**
- Casi todos los sistemas utilizan banderas.
- Ejemplo: varias interrupciones sucesivas podría provocar varios cambios de hebra sucesivos.



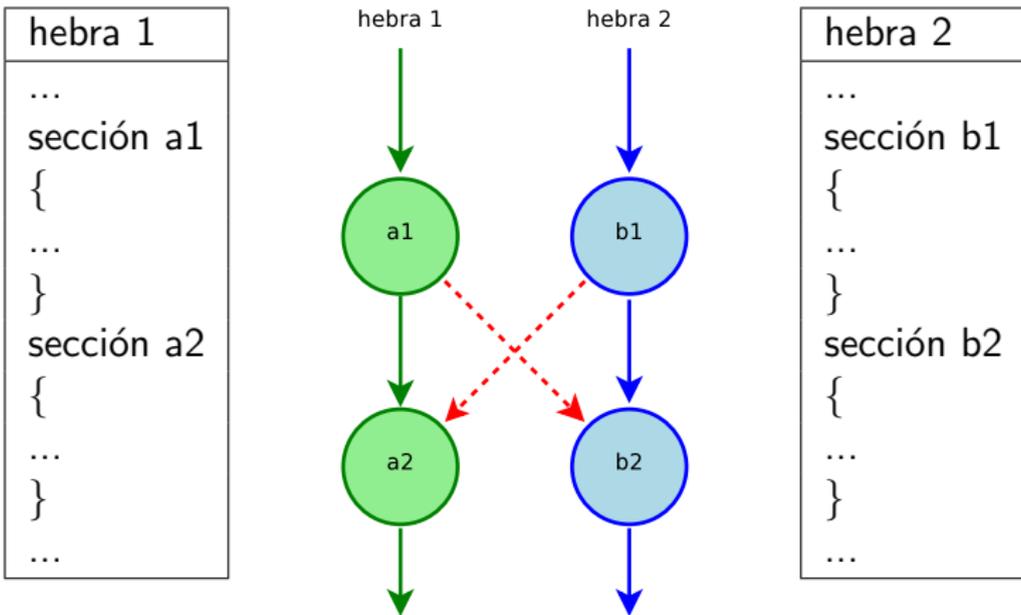
- Supongamos un sistema en que las interrupciones se producen con mucha frecuencia y las prioridades pueden anidarse → ¿Cómo puede ser beneficioso utilizar banderas?

# Ejemplo de uso de banderas

- Planificación perezosa y cambio de hebra: esperar hasta que haya que volver al espacio de usuario.
- Cada manejador de interrupción en el que pueda ser necesario planificar activa la bandera de planificación.

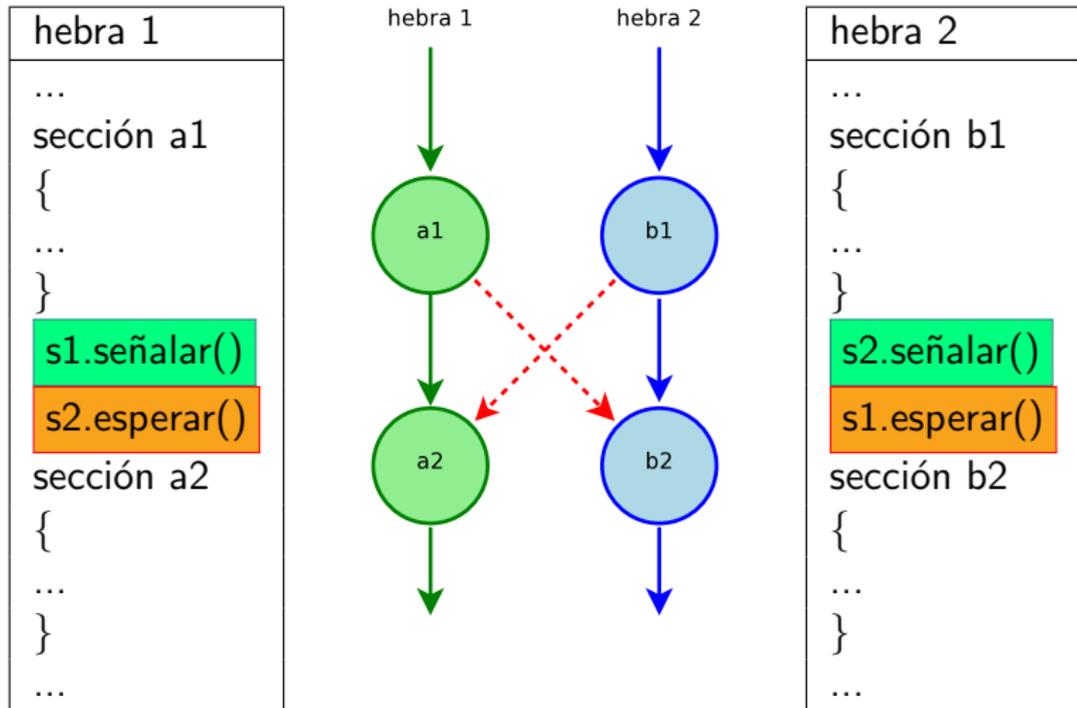


# Relación de precedencia mútua (1)



¿Cómo conseguir que  $a1 <^* b2$  y que  $b1 <^* a2$ ?

# Relación de precedencia mútua (2)



¿Funciona? ¿Puede hacerse mucho mejor?

# Relación de precedencia mútua (3)

## Ventajas:

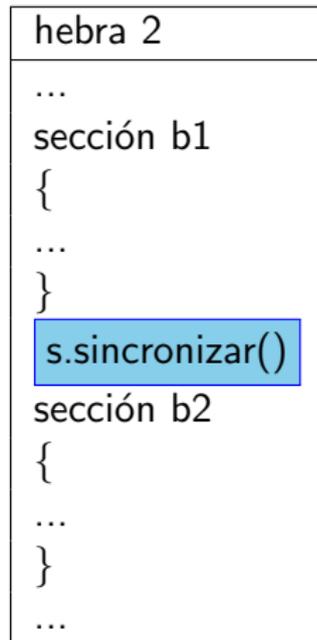
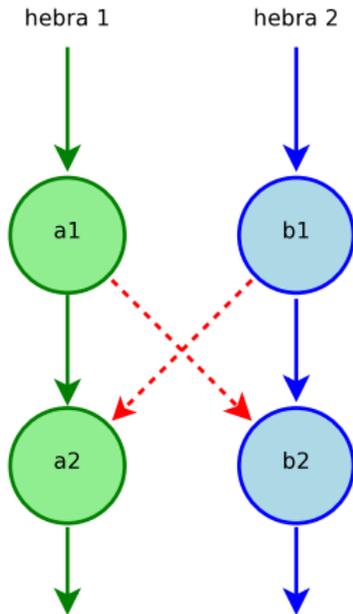
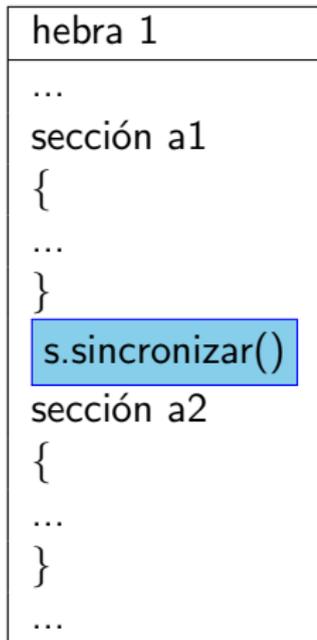
- No necesitamos nuevos mecanismos para un problema parecido.

## Inconvenientes:

- Complicado e ineficiente  
⇒ **no es escalable.**
- Es muy fácil olvidar la sincronización con una nueva hebra ⇒ engorroso y **propenso a errores.**
- Pobre rendimiento debido a que podríamos ahorrar llamadas al sistema ⇒ **no es eficiente.**

Conclusión: necesitamos un mecanismo más efectivo.

# Sincronización “pura”



Problema: ¿Cómo implementar la sincronización de dos hebras?

# Objeto sincronización para 2 hebras

```
class sincronización
{
private:
    bool activo;
    hebra bloqueada;

public:
    sincronización():
        activo(false),
        bloqueada(ninguna)
    {
    }
};

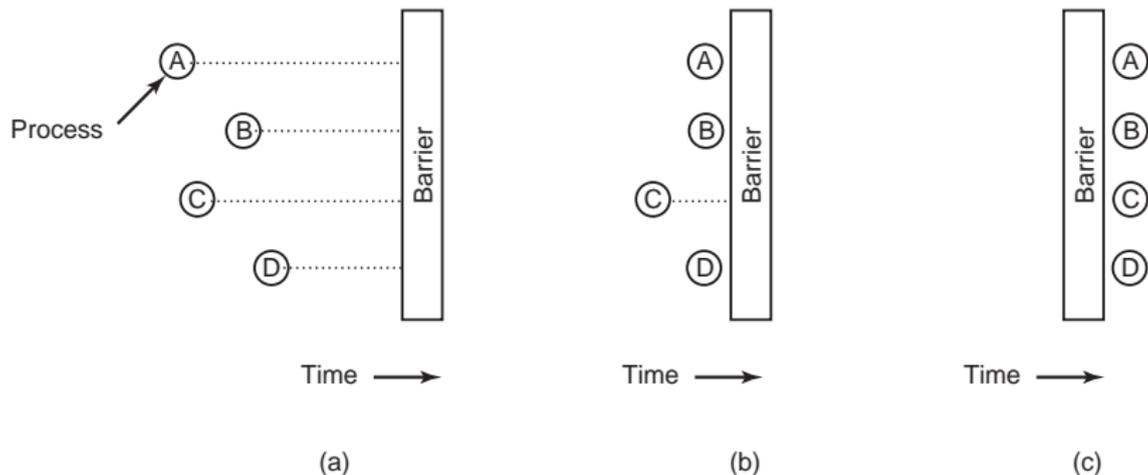
void sincronizar()
{
    if (!activo)                // soy el primero
    {
        activo = true;
        bloqueada = actual;
        bloquear(actual);      // esperar a la otra
    }
    else                          // soy el segundo
    {
        activo = false;        // reinicializar
        desbloquear(bloqueada); // liberar a la otra
    }
};
```

- ¿Es reutilizable?
- ¿Puede generalizarse para más de dos hebras?

# Ejemplo de sincronización entre n hebras (barrera)

```
// problema numérico resuelto
// mediante ecuaciones diferenciales
sincronización s;
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        {
            temp[i, j] = matriz[i - 1, j] + matriz[i + 1, j];
        }
s.sincronizar();
for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        {
            matriz[i, j] = temp[i, j];
        }
s.sincronizar();
```

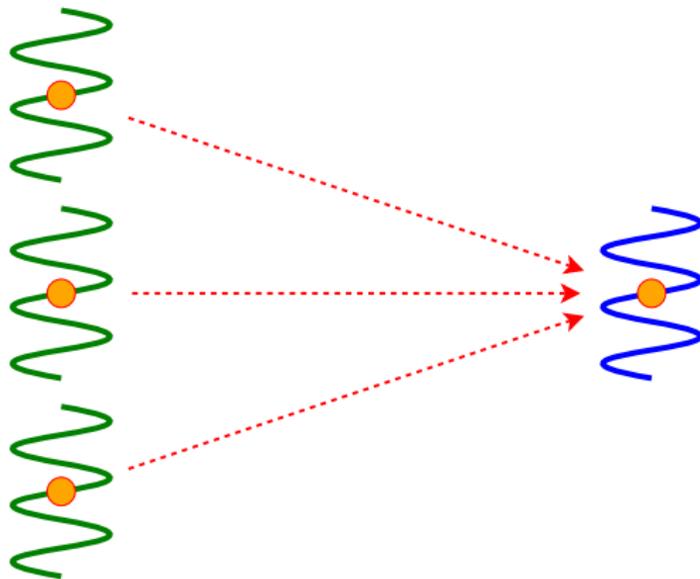
# Barrera



## Uso de una barrera:

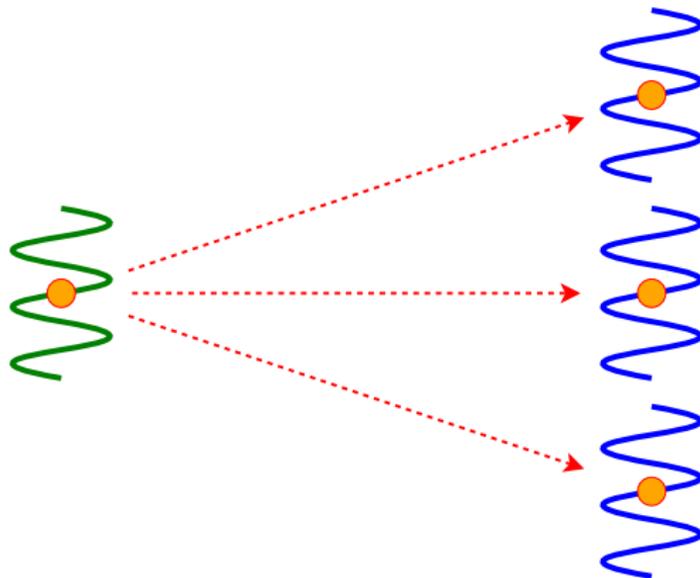
- Las hebras se aproximan a la barrera.
- Al llegar a la barrera las hebras se bloquean.
- La última hebra llega y provoca el desbloqueo de todas.

# Patrones de señalización: muchos a uno



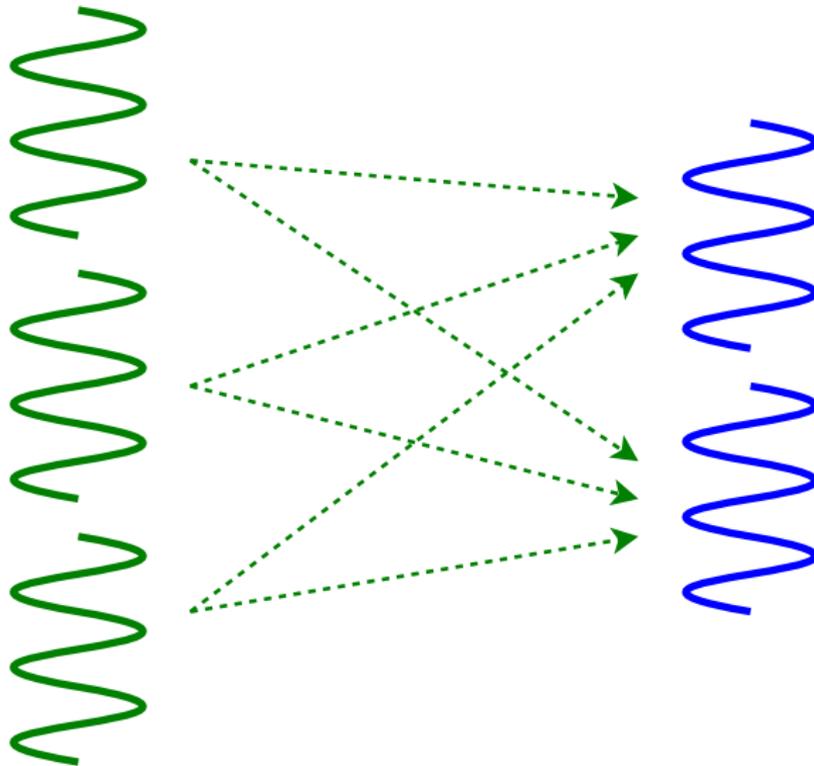
- Significado: la **hebra azul** sólo puede continuar tras un **punto de espera** si las tres **hebras verdes** han alcanzado cierto **punto específico** de su código.

# Patrones de señalización: uno a muchos

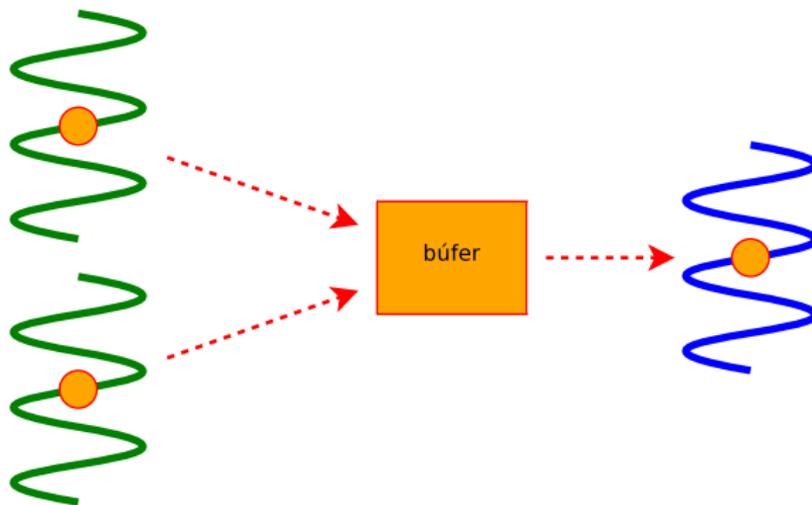


- Significado: las **hebras azules** sólo puede puede continuar si la **hebra verde** han alcanzado cierto punto de su código.

# Patrones de señalización: muchos a muchos



# Otros patrones de señalización



- La **hebra azul** sólo puede continuar si alguna de las dos **hebras verdes** alcanza cierto punto de su código.
- Problema: ¿Cómo y dónde almacenar temporalmente las señales?

# Almacenamiento temporal de señales

Una nueva señal **sobreescribe** a las anteriores: banderas o semáforos binarios.

- Ventaja: se reacciona sólo a la señal más nueva.
- Inconveniente: podríamos perder señales necesarias.

Cada nueva señal es **almacenada temporalmente** hasta que pueda ser consumida: semáforos con contador.

- Ventaja: se reacciona para toda señal.
- Inconveniente: una fuente de señales deficiente podría desbordar el sistema.

# Semáforos con contador (Dijkstra)

objetos señal del núcleo

**Definición:** un semáforo es una variable entera, que aparte de la inicialización, sólo puede ser modificada mediante dos operaciones atómicas y mutuamente excluyentes.

**$p()$  / decrementar() / esperar()**

Operación que decrementa el valor del semáforo.

**$v()$  / incrementar() / señalar()**

Operación que incrementa el valor del semáforo.

# Semáforos con contador

¿Cómo diseñar e implementar los semáforos con contador?

- Para evitar la **espera ocupada**...
- Cuando una hebra debe **esperar** ante un semáforo  $\implies$  poner a la hebra llamadora en una **cola de bloqueados** en espera de un evento.
- La ocurrencia de un evento puede comunicarse mediante el **incremento** del valor del semáforo.

# Semáforos con contador

Significado de los semáforos con contador:

- Un valor **positivo** indica el número señales emitidas.
- Un valor **negativo** indica el número de hebras que esperan por una señal  $\longrightarrow$  longitud de la cola de hebras bloqueadas en espera de una señal.
- Si el contador es **cero** ni hay señales emitidas ni hay hebras esperando que se produzca una señal.

# Semáforos con contador

```
class semáforo
{
public:
    semáforo(int _valor = 0):
        valor(_valor),
        bloq(vacia) {}

    void p() // esperar()
    {
        valor = valor - 1;
        if (valor < 0)
        {
            bloq.añadir(esta);
            bloquear(esta);
        }
    }

}

void v() // señalar()
{
    hebra h;
    valor = valor + 1;
    h = bloq.borrar_primerero();
    desbloquear(h);
}

private:
    int valor;
    cola<hebra> bloq;
};
```

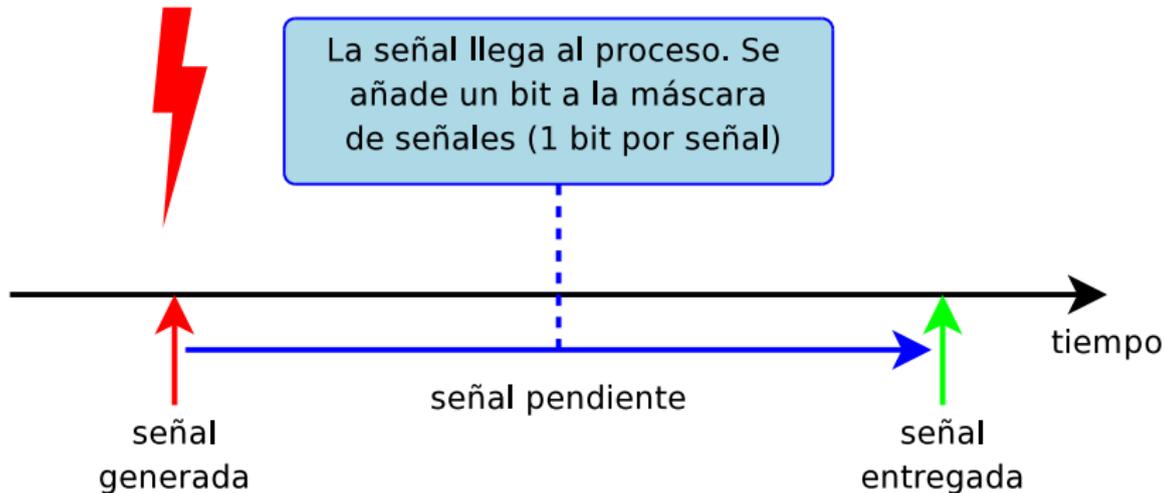
# Señales UNIX

- Pueden lanzarse mediante la orden **kill** del intérprete de órdenes y la llamada al sistema **kill()**.
- No tienen un significado ni un interfaz común.
- Hay cuatro versiones diferentes:
  - System V unreliable
  - BSD
  - System V reliable
  - POSIX
- El uso de las señales puede conducir a condiciones de carrera.
- La programación con señales es engorrosa.

# Lista de señales UNIX (POSIX.1)

Signal	Value	Action	Comment (Value = alpha/sparc,i386/ppc/sh,mips)
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

# Reparto de señales en UNIX



# Tipos de señales UNIX

- Las señales UNIX pueden recibirse...
  - síncrona
    - o
  - asíncronamente
- Las señales síncronas suelen enviarlas la propia hebra/proceso:
  - violación del espacio de direcciones.
  - excepción de división entre cero.
  - reservadas al usuario.
- Las señales asíncronas suele enviarlas otras hebras/procesos:
  - orden `kill`
  - llamada al sistema `kill()`
  - CTRL+C: `SIGINT` → terminar proceso
  - CTRL+Z: `SIGSTOP` → suspender proceso
  - expira el tiempo del temporizador

# API de las señales UNIX

```
#include <sys/types.h> // pid_t
#include <pthread.h>    // pthread_t
#include <unistd.h>     // getpid
#include <signal.h>     // kill pthread_kill raise signal

// instala la función ‘manejador’ como manejador
// de señal para la señal ‘signal’
typedef void (*sighandler_t)(int);
sighandler_t signal(int signal, sighandler_t manejador);

// envia la señal ‘signal’ a el proceso ‘pid’
int kill(pid_t pid, int signal);

// envía la señal ‘signal’ a el proceso actual
// equivalente a kill(getpid(), signal)
int raise(int signal);

// envía la señal ‘signal’ a la hebra ‘hebra’
int pthread_kill(pthread_t hebra, int signal);
```

# Manejadores de señal UNIX

- El manejador por defecto, ¿se ejecuta en modo usuario o modo núcleo? → depende de quien maneje el evento.
- Manejadores definidos por el usuario:
  - Implemente manejadores de señal cortos.
  - Tenga mucho cuidado al utilizar señales.
  - Muchos sistemas UNIX exigen reinstalar el manejador de señal para poder volver a utilizarlo por segunda vez.
    - ¿Detecta una posible condición de carrera?

## alarma.cc

```
int FRECUENCIA_ALARMA = 1; // segundos
void atrapar_alarma(int); // declaraci n anticipada

//-----

void instalar_alarma() // instalador del manejador de se al y alarma
{
    signal(SIGALRM, atrapar_alarma); // establecer manejador de la se al
    alarm(FRECUENCIA_ALARMA);      // establecer alarma
}

//-----

void atrapar_alarma(int) // manejador de se al
{
    instalar_alarma(); // reinstalar tras cada alarma
    cout << "  ALARMA  !!!" << endl;
}

//-----

int main()
{
    instalar_alarma(); // instalar alarma por primera vez

    for(;;); // bucle infinito
}
```