

SISTEMAS OPERATIVOS

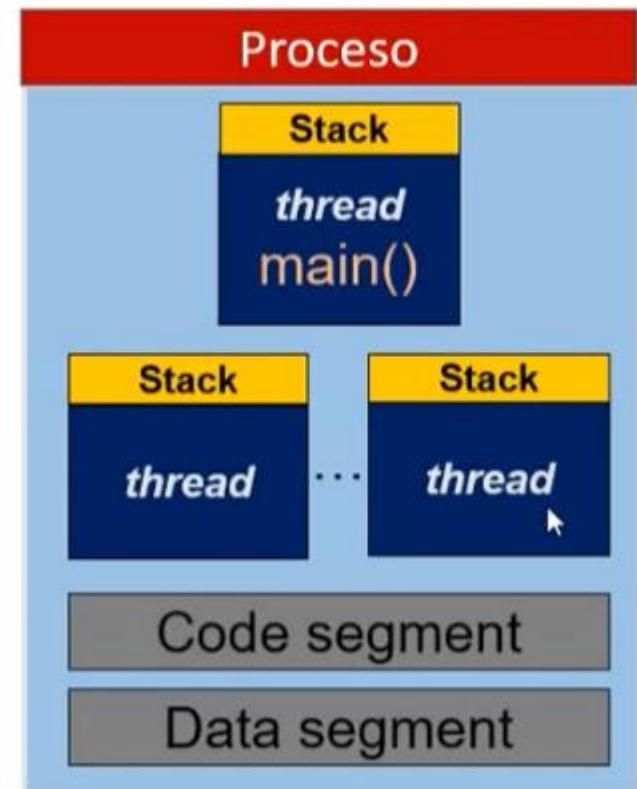
Ingeniería civil informática

GONZALO CARREÑO

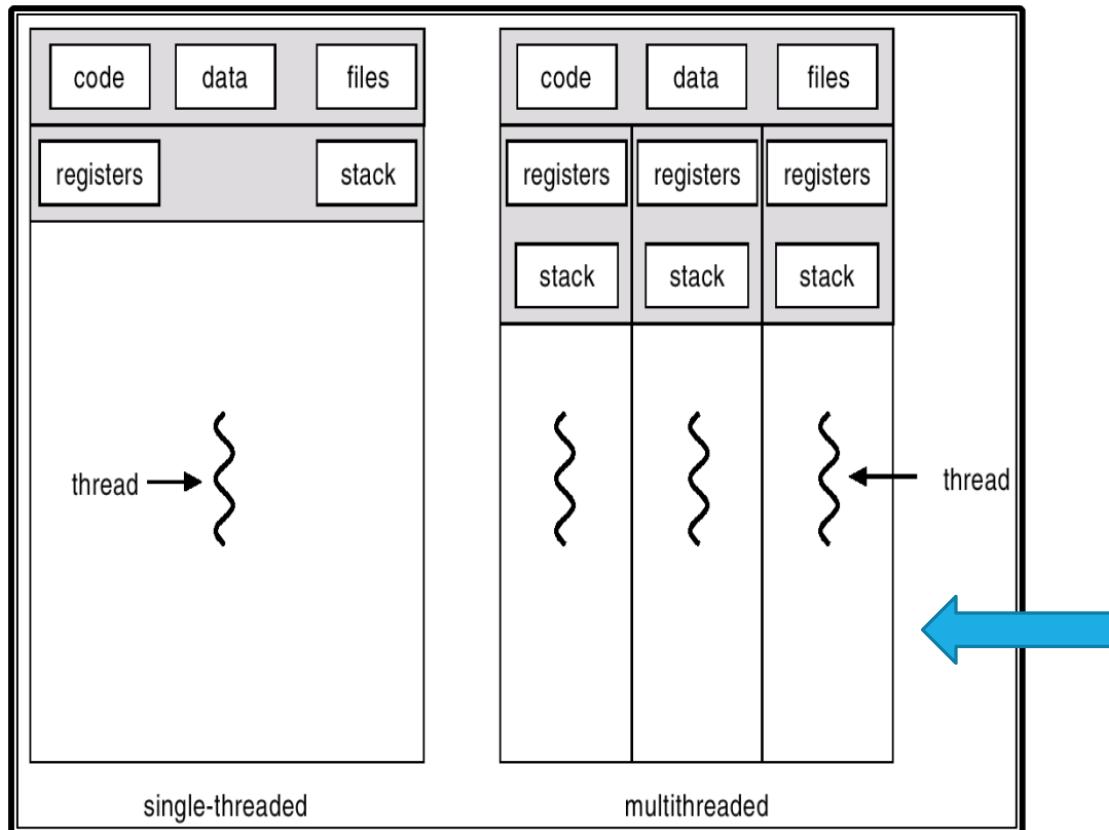
GONZALOCARRENOB@GMAIL.COM

Hilos

- La mayoría de los modernos SO proporcionan procesos con múltiples secuencias o hilos de control en su interior.
- Se considera una unidad básica de utilización de la CPU.
- Cada uno comprende:
 - Identificador de thread
 - Contador de programa
 - Conjunto de registros
 - Pila
- Comparten con el resto de hilos del proceso:
 - Mapa de memoria (sección de código, sección de datos, shmem)
 - Ficheros abiertos
 - Señales, semáforos y temporizadores.



Hilo, Proceso ligero o Thread



Un hilo comparte con sus hilos pares:

- Sección de código
- Sección de datos
- Recursos del sistema operativo

Ejemplo de código cualquiera

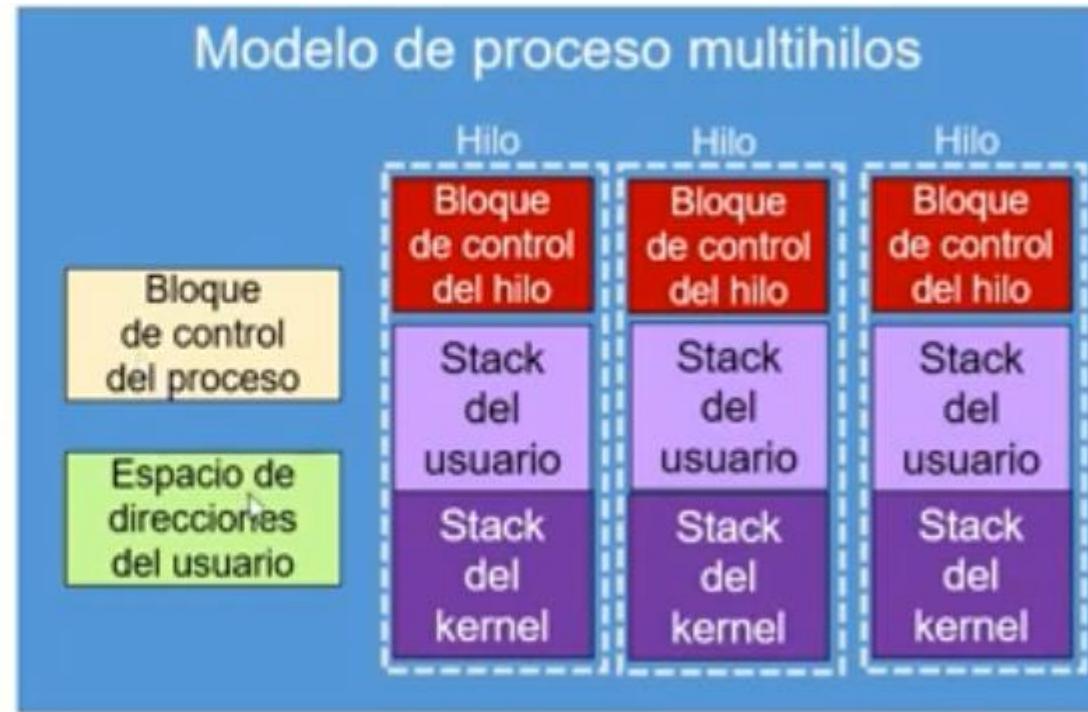
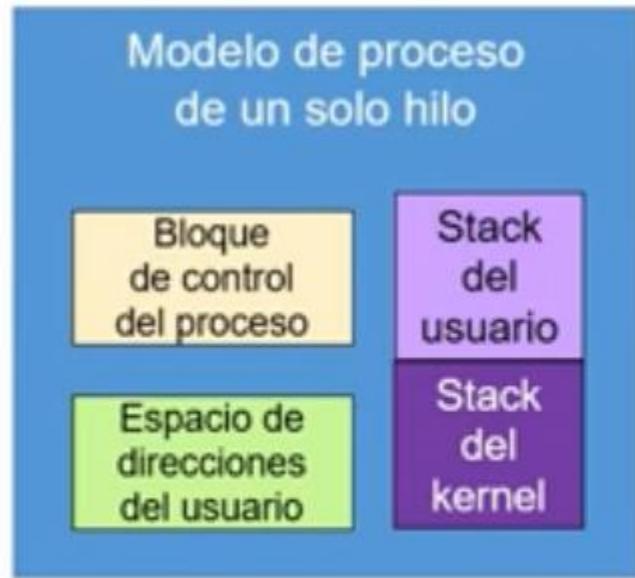
```
void *tfunc(void *args)
{
    printf("Hola mundo\n");
}

int main()
{
    int i;
    pthread_t id_hilo[NTHREADS];

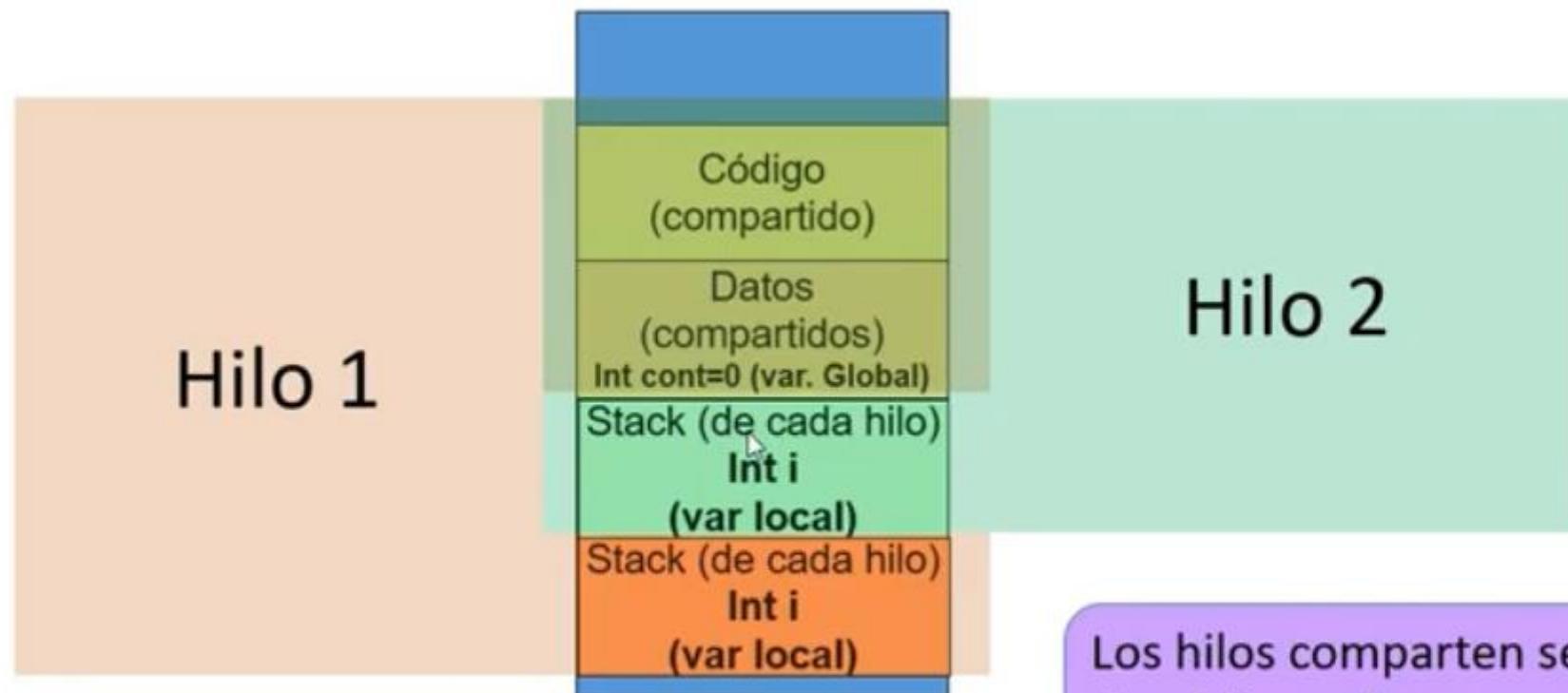
    for(i=0;i<3;i++)
        pthread_create(&id_hilo[i],NULL,tfunc,NULL);

    for(i=0;i<3;i++)
        pthread_join(id_hilo[i],NULL);
}
```

Modelo de proceso de un hilo y de muchos hilos



Hilos en memoria



Hilo 2

Los hilos comparten segmento de código y datos, pero cada hilo debe tener su stack



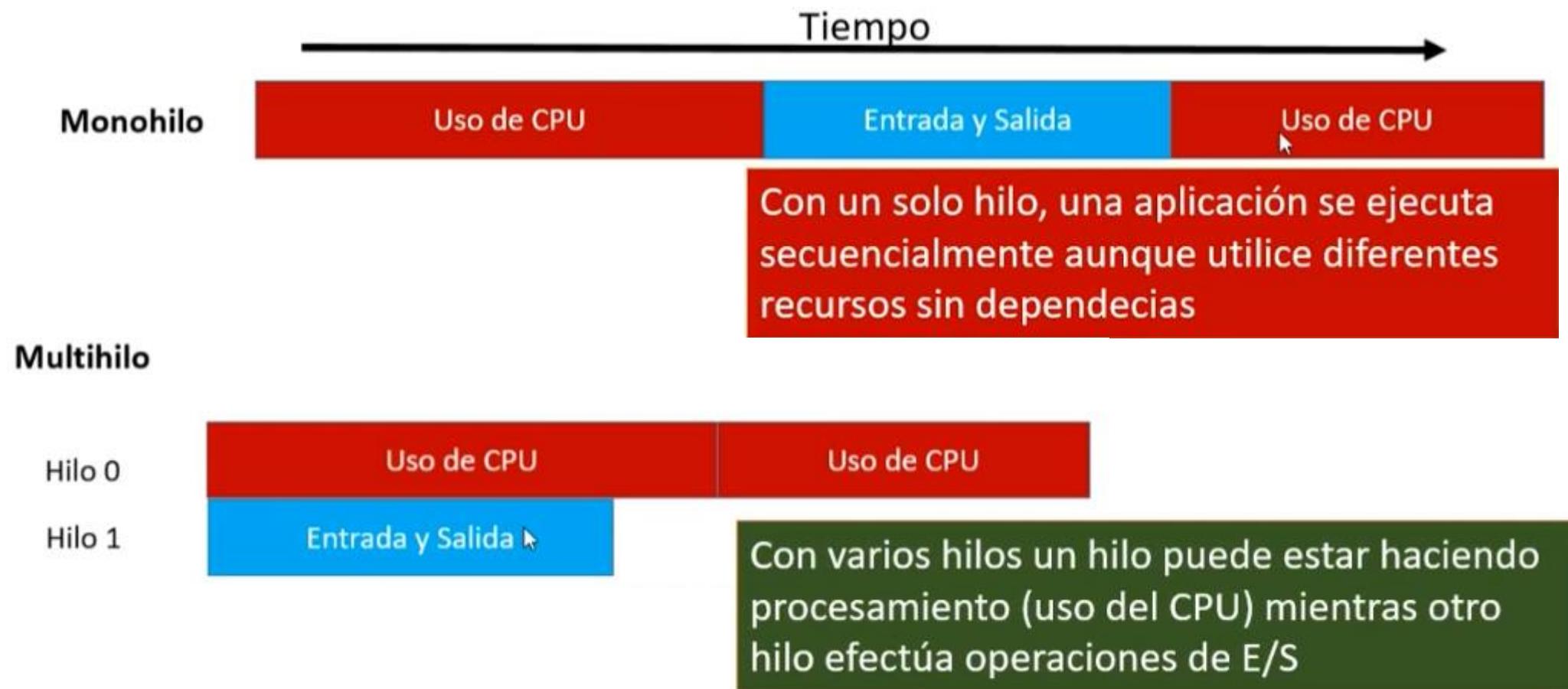
Sistemas operativos multihilo

- Sistema operativo que mantiene varios hilos de ejecución dentro de un mismo proceso.
- MS-DOS soporta un solo proceso con un solo hilo.
- UNIX soporta múltiples procesos de usuario, pero solo un hilo por proceso.
- Windows, Solaris, Linux, OS X y OS/2 soportan múltiples hilos.

Beneficios

- Capacidad de respuesta.
 - Mayor interactividad al separar las interacciones con el usuario de las tareas de procesamiento en distintos hilos.
- Compartición de recursos.
 - Los hilos comparten la mayor parte de los recursos de forma automática.
- Economía de recursos.
 - Crear un proceso consume mucho más tiempo que crear un hilo (Ejemplo: en Solaris relación 30 a 1).
- Utilización sobre arquitecturas multiprocesador.
 - Mayor nivel de concurrencia asignando distintos hilos a distintos procesadores.
 - La mayoría de los sistemas operativos modernos usan el hilo como unidad de planificación.

Beneficios



Ejemplo 1

Un navegador web puede tener un hilo para leer las imágenes de la red, mientras que otro hilo las está mostrando en pantalla.

Monohilo

| | | | | | |
|------------------|------------------|------------------|------------------|------------------|------------------|
| Obtener imagen 1 | Mostrar imagen 1 | Obtener imagen 2 | Mostrar imagen 2 | Obtener imagen 3 | Mostrar imagen 3 |
|------------------|------------------|------------------|------------------|------------------|------------------|

Multihilo

Hilo 0

| | | | |
|------------------|------------------|------------------|------------------|
| Obtener imagen 1 | Obtener imagen 2 | Obtener imagen 3 | |
| | Mostrar imagen 1 | Mostrar imagen 2 | Mostrar imagen 3 |

Hilo 1

Ejemplo 2

Un procesador de texto puede tener un hilo para mostrar gráficos, otro hilo para responder al tecleo del usuario y un tercer hilo para realizar la revisión ortográfica y gramatical en segundo plano.

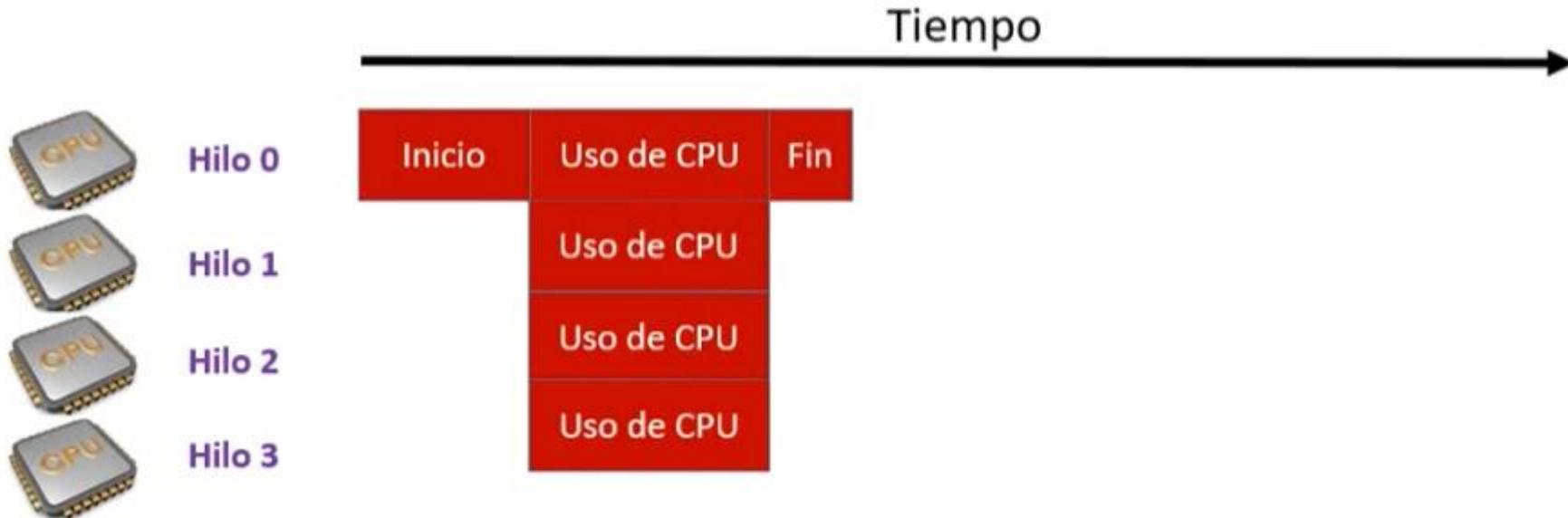
Multihilo



Trabajando con un solo hilo



Implementando hilos



Soporte de hilos

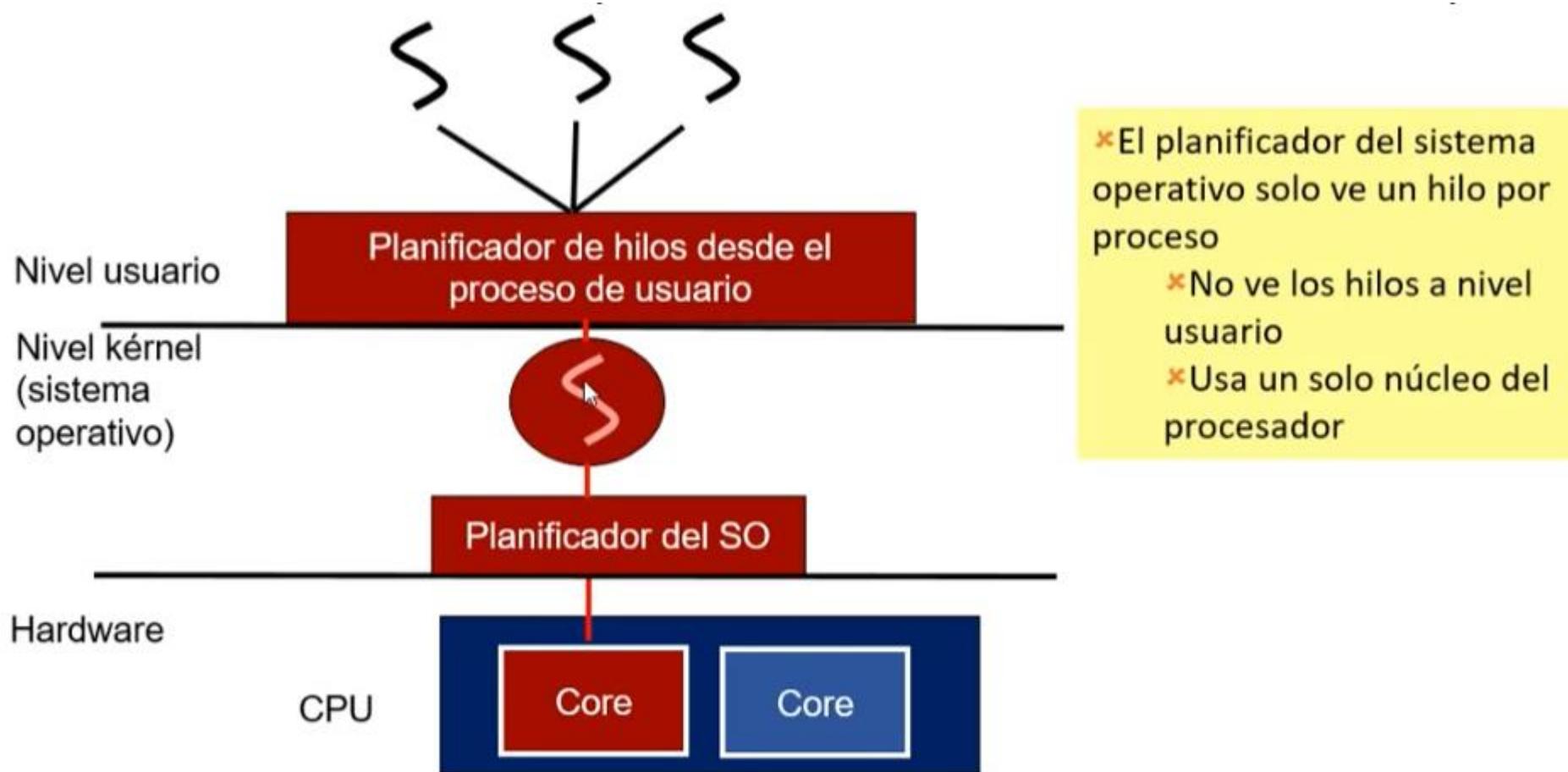
Espacio de usuario ULT – User Level Threads

- Implementados en forma de biblioteca de funciones en espacio de usuario.
- El kernel no tiene conocimiento sobre ellos, no ofrece soporte de ningún tipo.
- Es mucho más rápido pero presentan algunos problemas → Llamadas al sistema bloqueantes.

Hilos a nivel usuario



Hilos a nivel usuario (un CPU con dos núcleos)

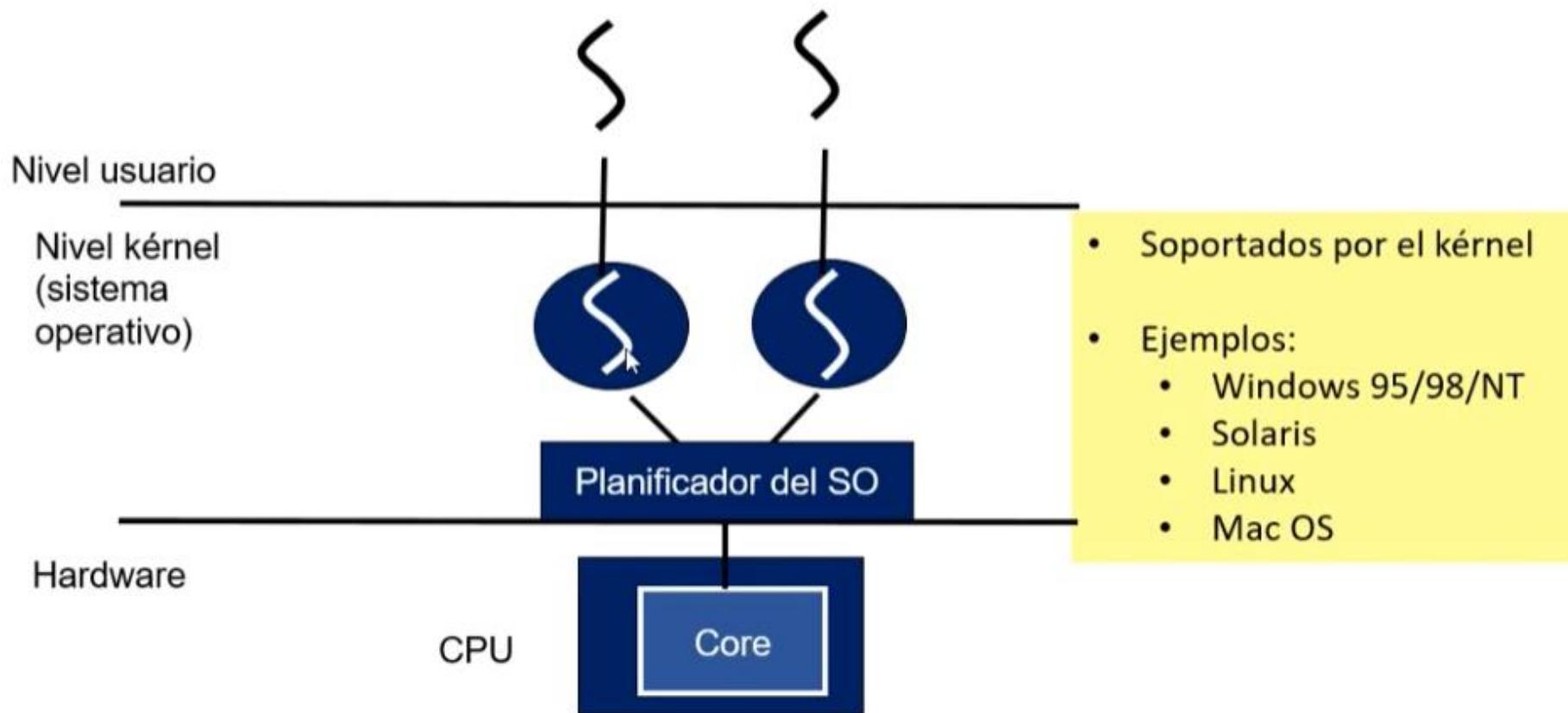


Soporte de hilos

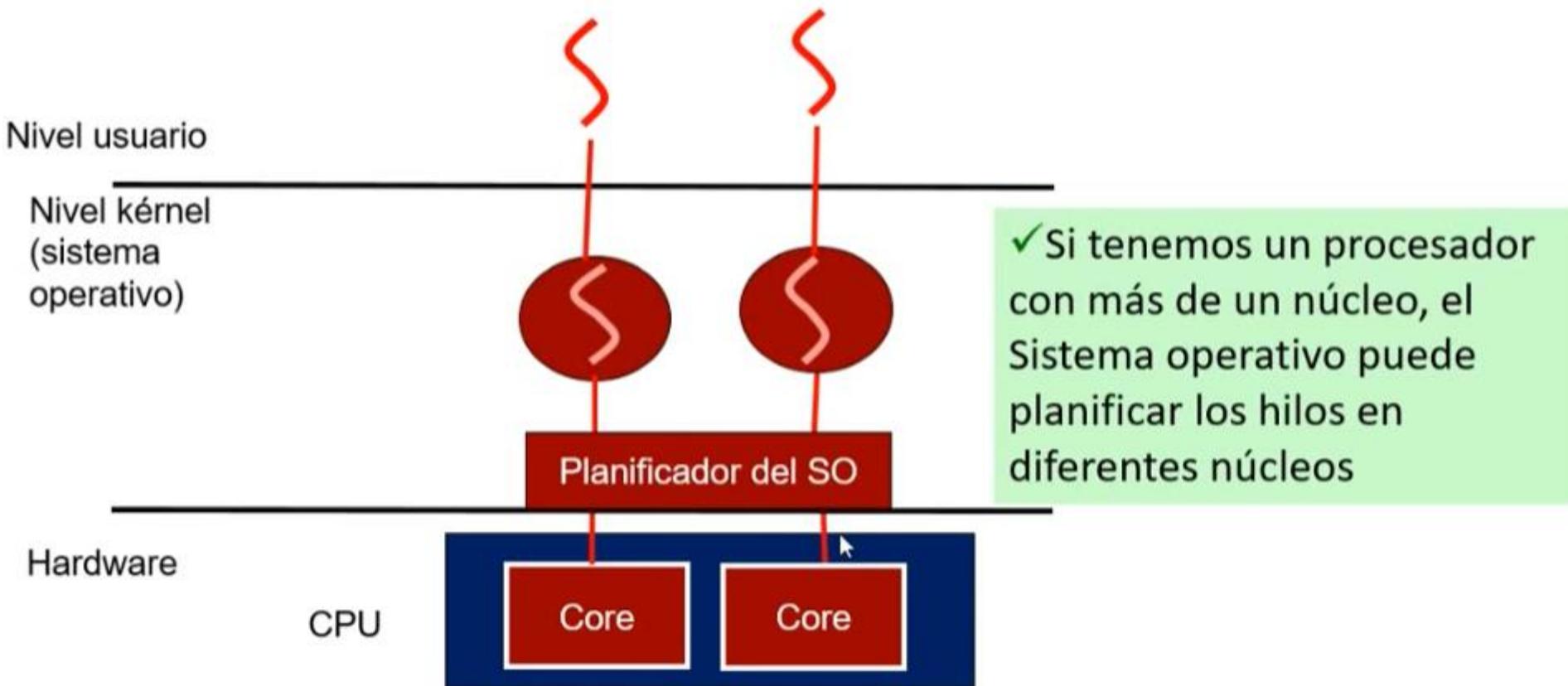
Espacio de núcleo KLT – Kernel Level Threads

- El kernel se ocupa de crearlos, planificarlos y destruirlos.
- Es un poco más lento ya que hacemos participar al kernel y esto supone un cambio de modo de ejecución.
- En llamadas al sistema bloqueantes sólo se bloquea el thread implicado.
- En sistemas SMP, varios threads pueden ejecutarse a la vez.
- No hay código de soporte para thread en las aplicaciones.
- El kernel también puede usar threads para llevar a cabo sus funciones.

Hilos a nivel Kernel



Hilos a nivel de kernel(CPU multicore)



Modelos multihilos

- Muchos a uno
- Uno a uno
- Muchos a muchos

Modelos de múltiples hilos: Muchos a uno

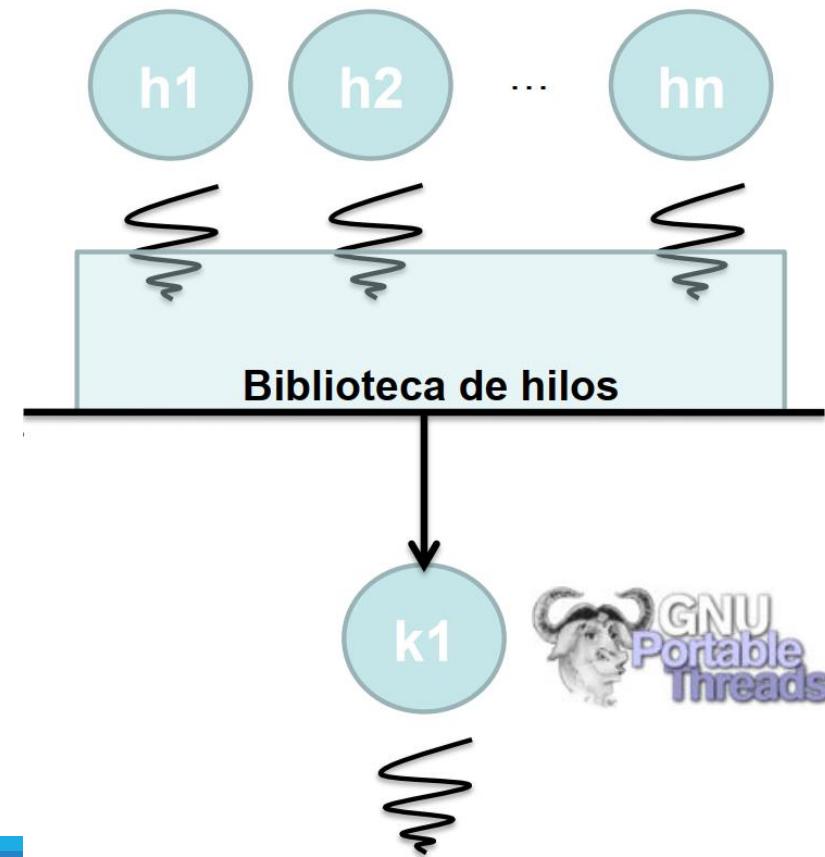
Hace corresponder múltiples hilos de usuario a un único hilo del núcleo.

Biblioteca de hilos en espacio de usuario.

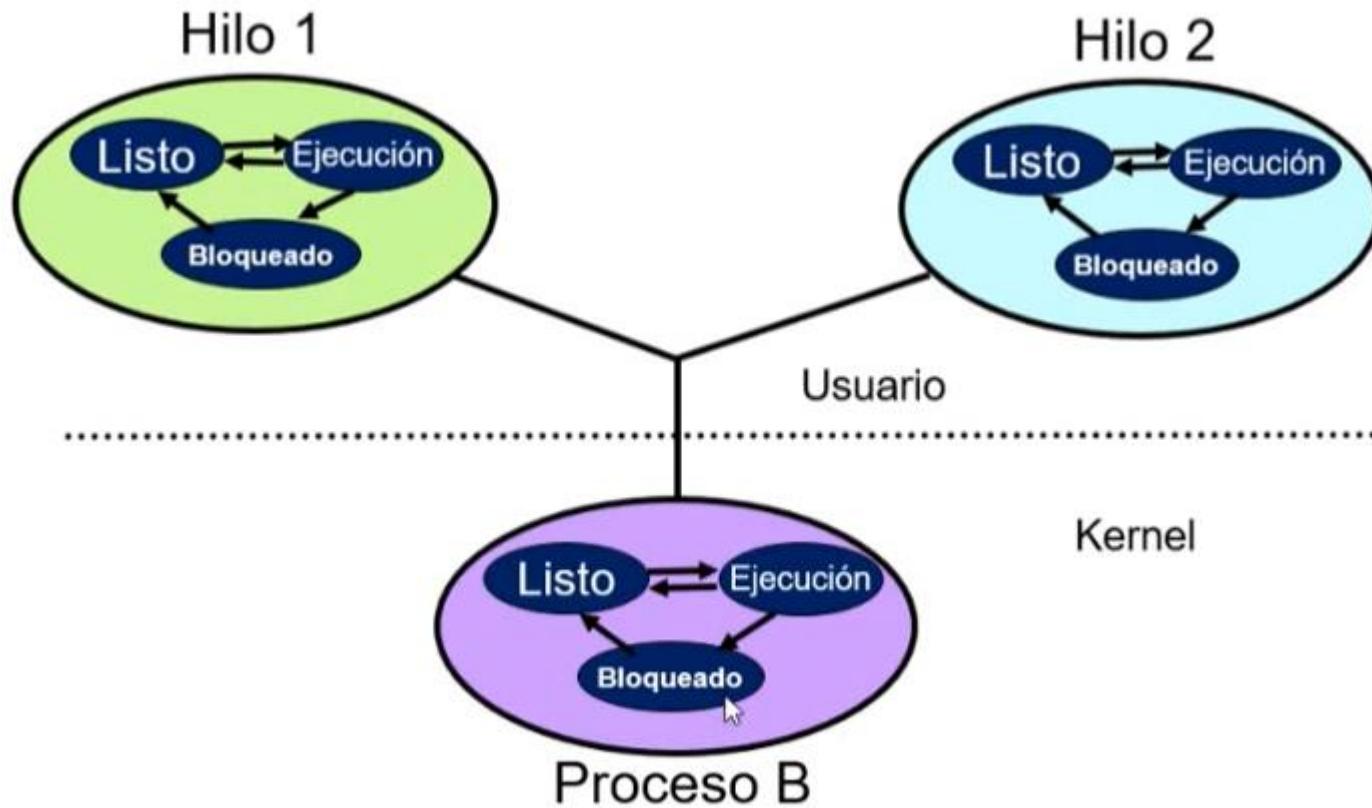
Llamada bloqueante:

- Se bloquean todos los hilos.

En multiprocesadores no se pueden ejecutar varios hilos a la vez.



Relaciones entre estados de UTLs y estados de procesos



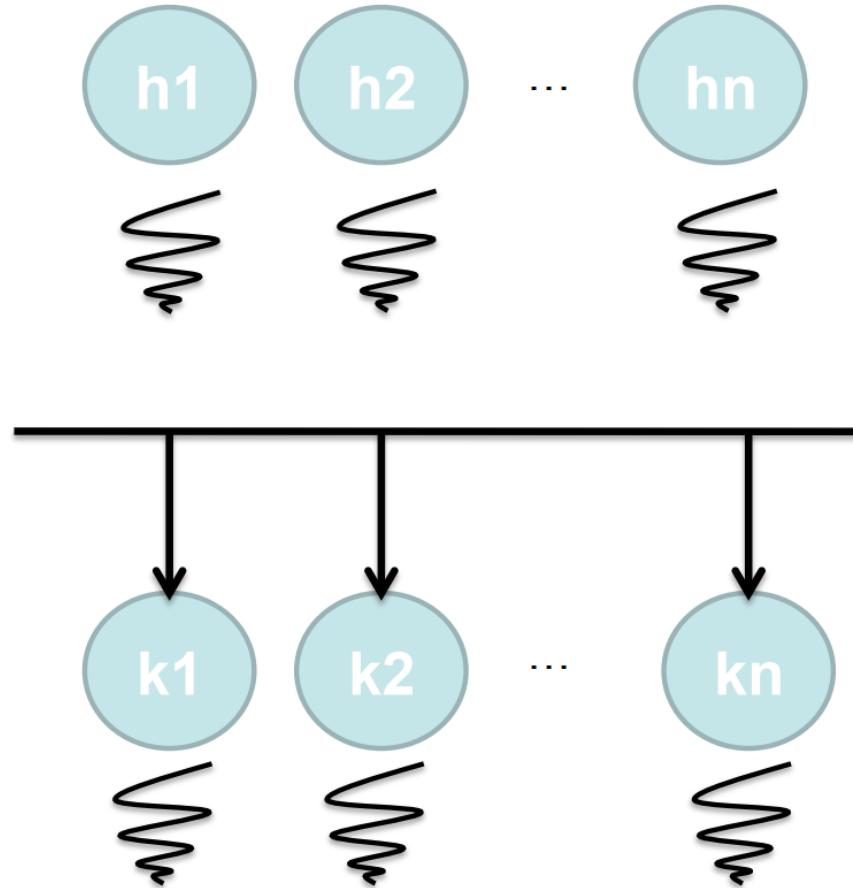
Modelo de múltiples hilos: Uno a uno

Hace corresponder un hilo el kernel a cada hilo de usuario.

La mayoría de las implementaciones restringen el número de hilos que se pueden crear.

Ejemplos:

- Linux 2.6.
- Windows.
- Solaris 9.



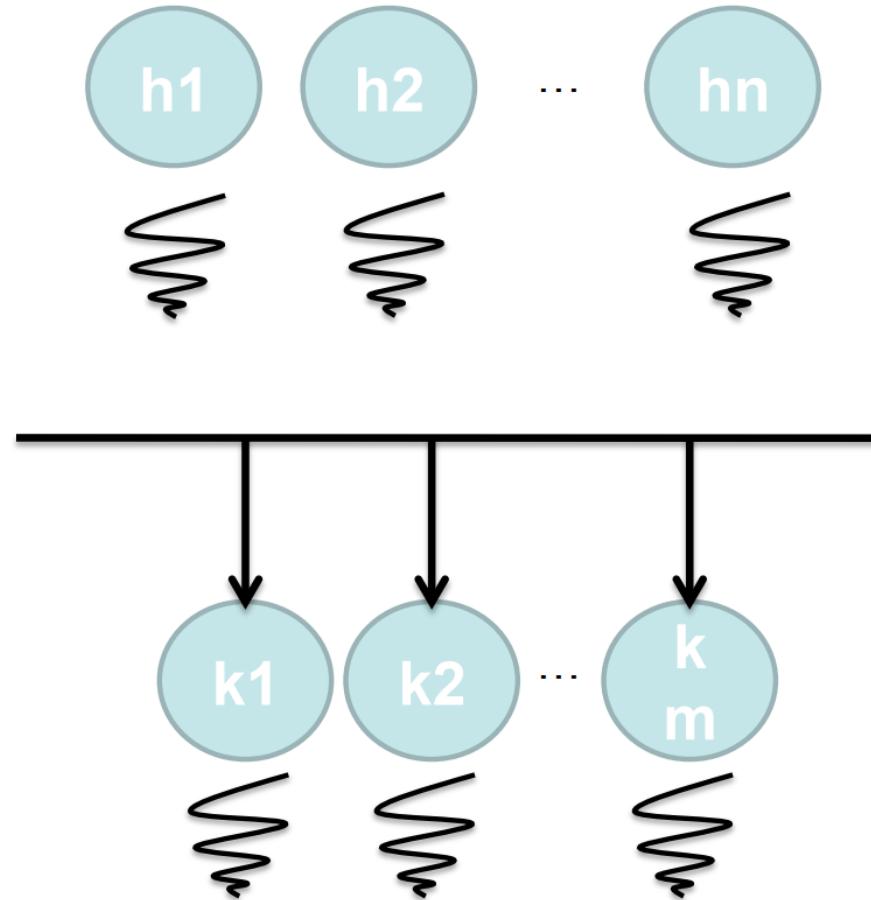
Modelos de múltiples hilos: Muchos a muchos

Este modelo multiplexa los threads de usuario en un número determinado de threads en el kernel.

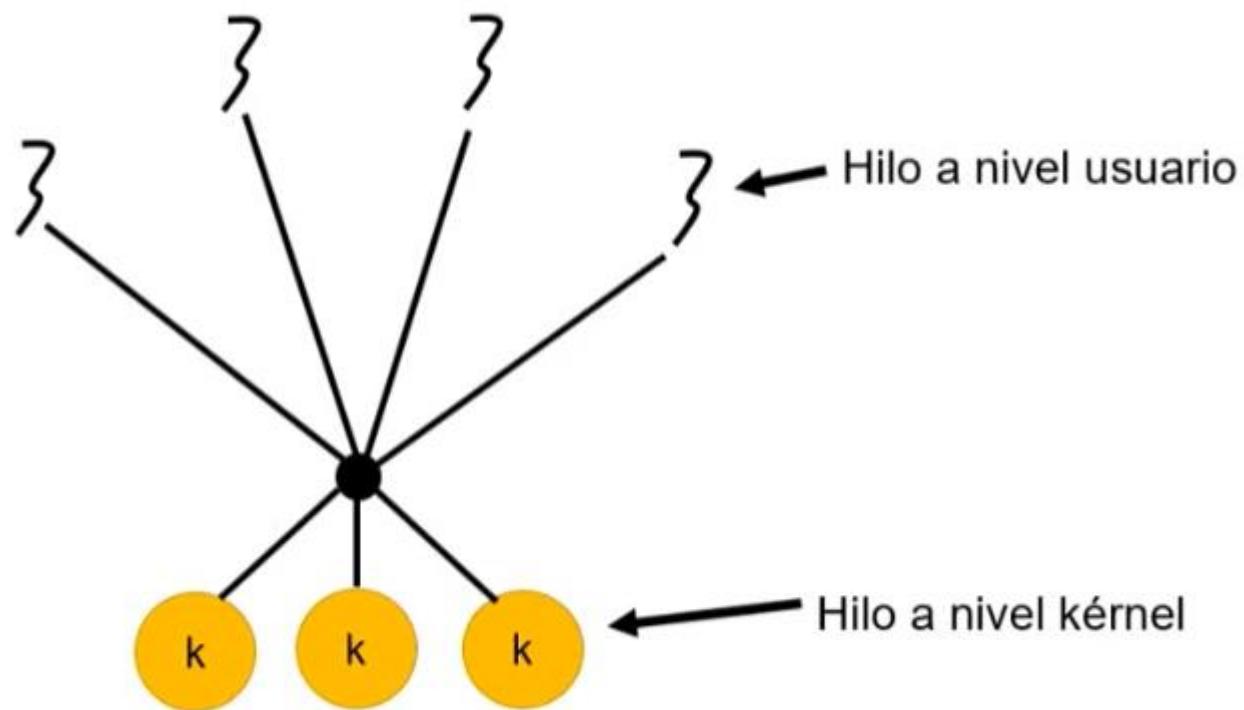
El núcleo del sistema operativo se complica mucho.

Ejemplos:

- solaris (versiones anteriores a 9).
- HP-UX.
- IRIX.



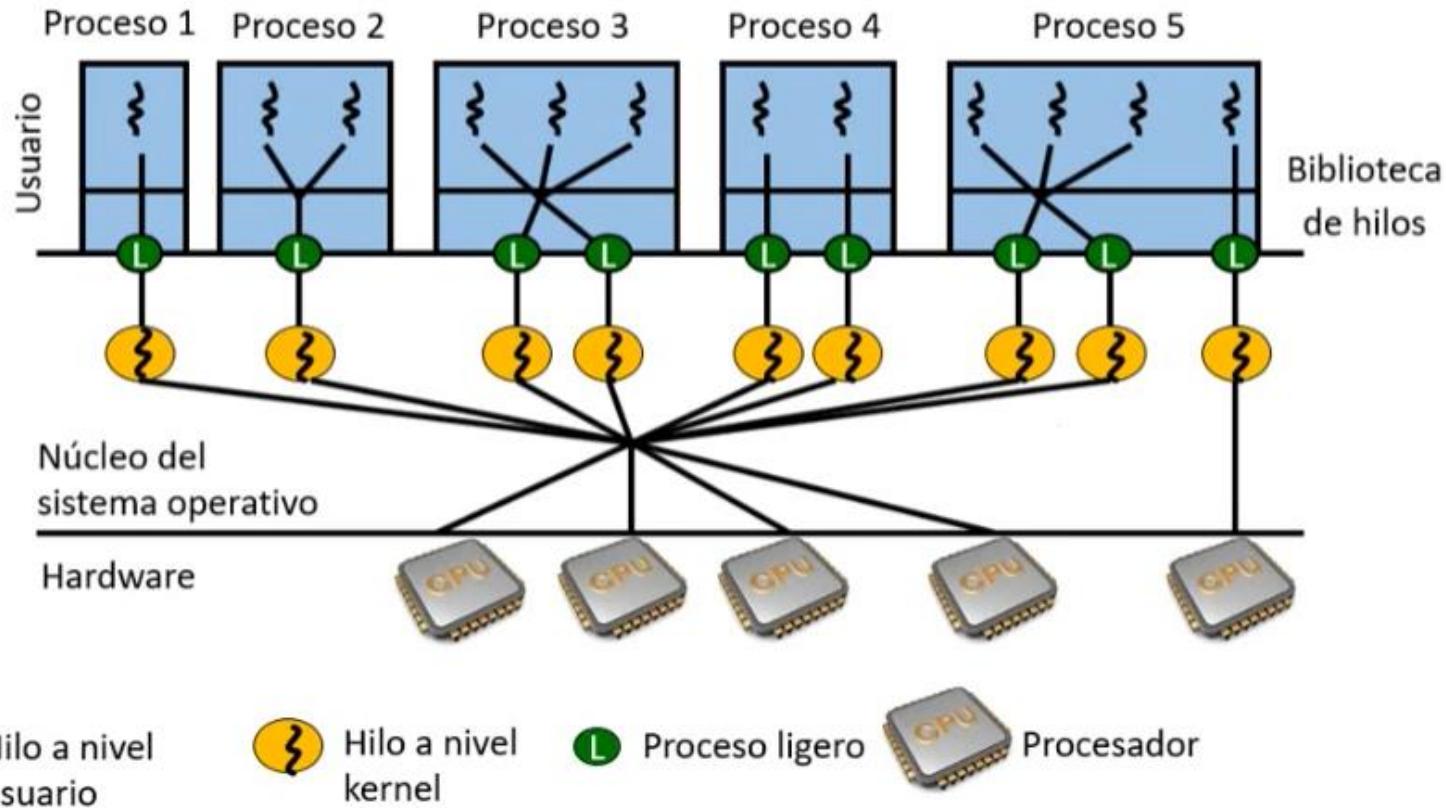
Modelos de múltiples hilos: Muchos a muchos



Ejemplo de arquitectura multihilo de Solaris

Los procesos tienen un hilo principal más hilos a nivel usuario

Proceso ligero es un hilo de ejecución en un proceso. Un proceso ligero corresponde a un hilo a nivel kernel

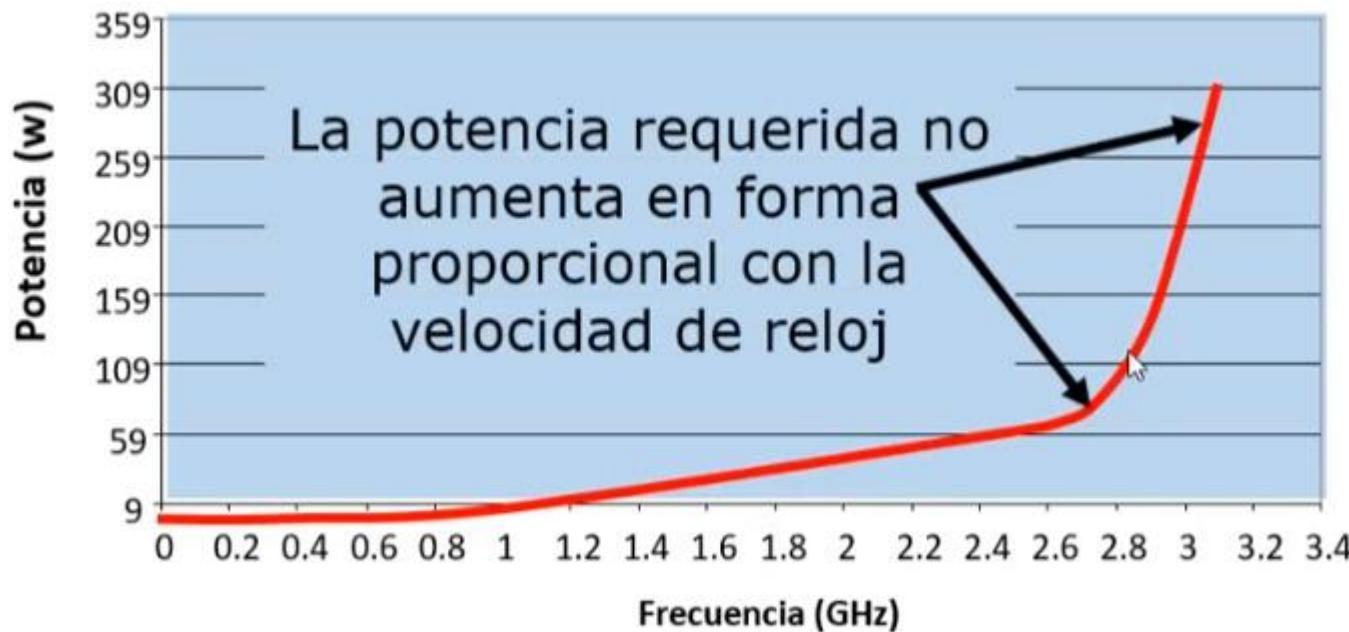


La velocidad del CPU tendía a duplicarse cada 18 meses

| Año | Procesador | Velocidad |
|------|-------------|-----------|
| 1991 | Intel 386 | 25 Mhz |
| 1993 | Intel 486 | 50 Mhz |
| 1995 | Pentium | 100 Mhz |
| 1996 | Pentium MMX | 200 Mhz |
| 1998 | Pentium II | 400 MHz |
| 2000 | Pentium III | 800 Mhz |
| 2001 | Pentium IV | 1.6 Ghz |
| 2003 | Pentium IV | 3.0 Ghz |

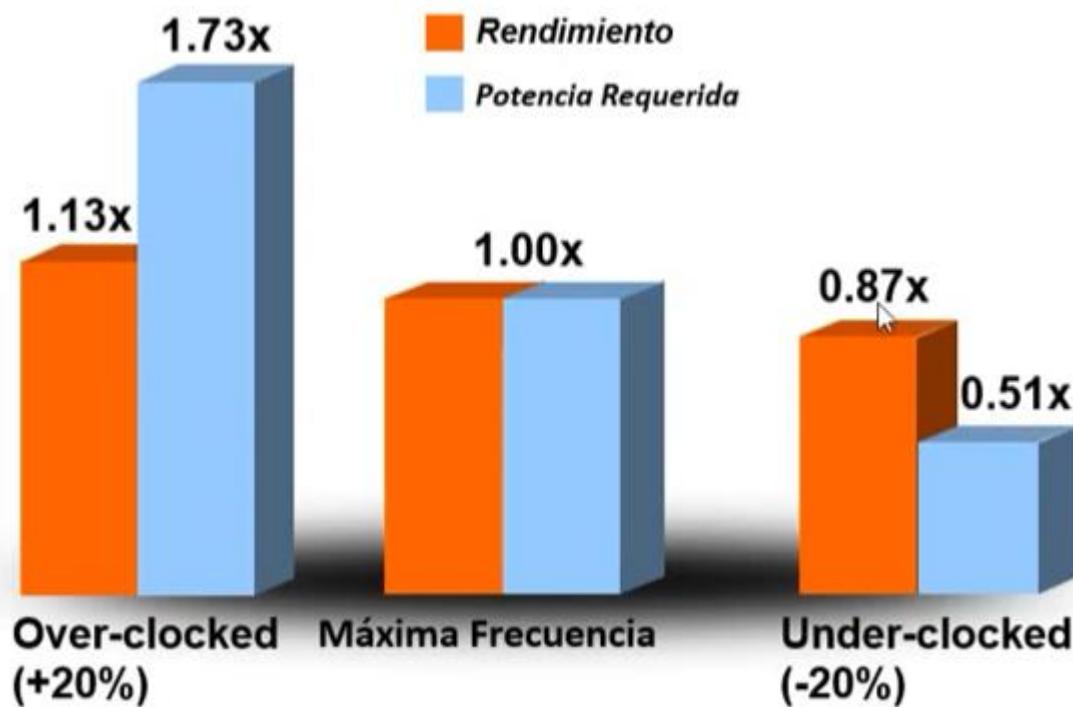
Potencia y frecuencia

Curva de Potencia vs. Frecuencia para arquitecturas con un núcleo

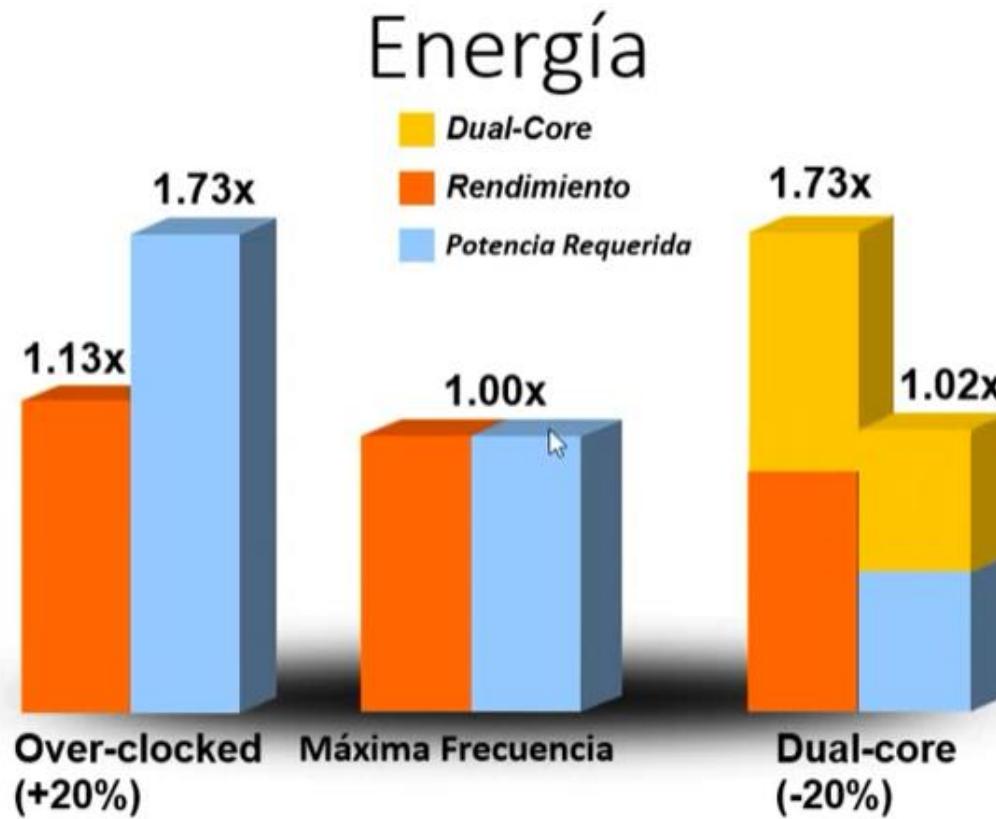


Menor velocidad de reloj nos permite ahorrar potencia para aumentar los núcleos

Over-clocking y under-clocking

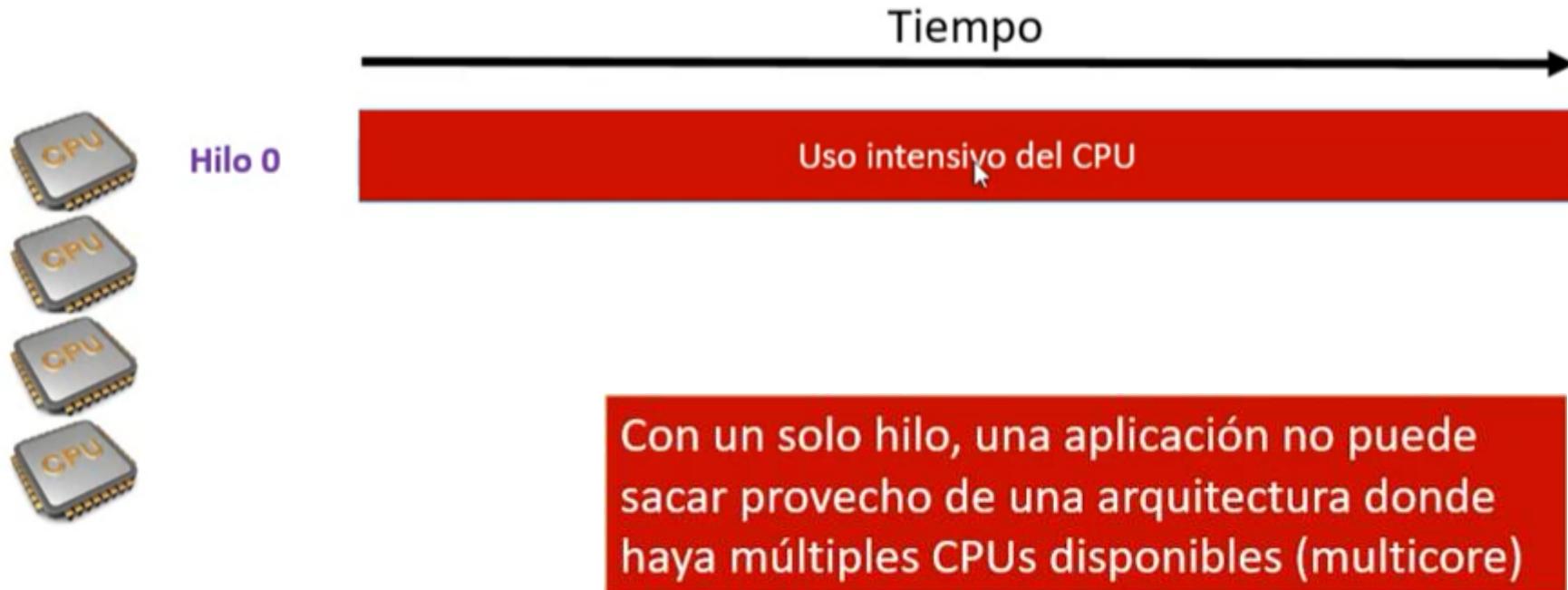


Over-clocking y under-clocking

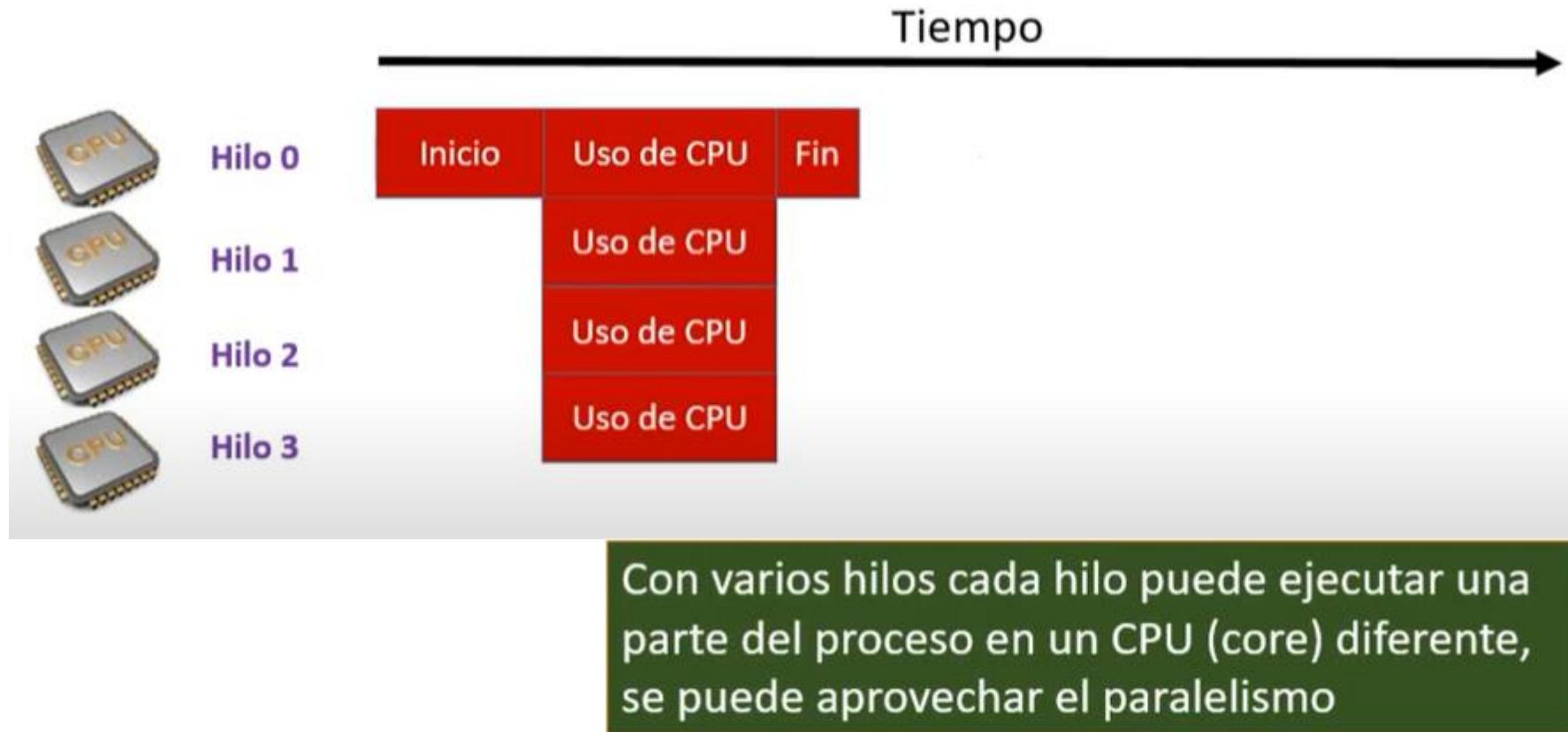


Aplicación multihilos en una arquitectura con múltiples CPU's o Multicore

¿Para que sirven los hilos en una arquitectura multicore o una arquitectura donde haya múltiples CPU's?



Aplicación multihilos en una arquitectura con múltiples CPU's o Multicore



Gestión de hilos básica

- Un hilo tiene:
 - Un identificador(TID)
 - Un stack
 - Una prioridad de ejecución
 - Una dirección de ejecución

Cancelación de hilos

Situación en la que un hilo notifica a otros que deben terminar.

Opciones:

- Cancelación asíncrona: Se fuerza la terminación inmediata del hilo.
 - Problemas con los recursos asignados al hilo.
- Cancelación diferida: El hilo comprueba periódicamente si debe terminar.
 - Preferible.

Hilos y procesamiento de solicitudes

- Las aplicaciones que reciben peticiones y las procesan pueden usar hilos para el tratamiento.
- Pero:
 - El tiempo de creación/destrucción del hilo supone un retraso (aunque sea menor que el de creación/destrucción de un proceso).
 - No se establece un límite en el número de hilos concurrentes.
 - Si llega una avalancha de peticiones se pueden agotar los recursos del sistema.

Thread Pools o Conjuntos de Hilos

- Se crea un conjunto de hilos que quedan en espera a que lleguen peticiones.
- Ventajas:
 - Se minimiza el retardo: El hilo ya existe.
 - Se mantiene un límite sobre el número de hilos concurrentes.

Gestión de hilos básica (Posix)

- Se dice que un hilo es dinámico si se puede crear en cualquier instante durante la ejecución.
- En POSIX:
 - Los hilos se crean dinámicamente con la función **pthread_create**.
 - Un hilo termina al invocar la función **pthread_exit**.
 - El hilo principal debe esperar a que todos los hilos terminen con **pthread_join**.

Ejemplo hilos Posix

```
void *tfunc(void *args)
{
    printf("Hola mundo\n");
}

int main()
{
    int i;
    pthread_t id_hilo[NTHREADS];

    for(i=0;i<3;i++)
        pthread_create(&id_hilo[i],NULL,tfunc,NULL);

    for(i=0;i<3;i++)
        pthread_join(id_hilo[i],NULL);
}
```

El hilo principal debe esperar a que terminen todos los hilos creados

- Si el hilo principal no espera a que terminen los hilos creados, al terminar termina el proceso con todos los hilos.

Creación de hilos

- `int pthread_create(pthread_t *thread,
const pthread_attr_t *attr,
void *(*func)(void *),
void *arg)`
 - Crea un hilo e inicia su ejecución.
 - **thread**: Se debe pasar la dirección de una variable del tipo `pthread_t` que se usa como manejador del hilo.
 - **attr**: Se debe pasar la dirección de una estructura con los atributos del hilo. Se puede pasar NULL para usar atributos por defecto.
 - **func**: Función con el código de ejecución del hilo.
 - **arg**: Puntero al parámetro del hilo. Solamente se puede pasar un parámetro.
- `pthread_t pthread_self(void)`
 - Devuelve el identificador del thread que ejecuta la llamada.

Espera y terminación

- `int pthread_join(pthread_t thread,
void **value)`
 - El hilo que invoca la función se espera hasta que el hilo cuyo manejador se especifique haya terminado.
 - **thread**: Manejador de del hilo al que hay que esperar.
 - **value**: Valor de terminación del hilo.

- `int pthread_exit(void *value)`
 - Permite a un proceso ligero finalizar su ejecución, indicando el estado de terminación del mismo.
 - El estado de terminación no puede ser un puntero a una variable local.

Ejemplo

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

void * thread_function(void *arg)
{
    int i;
    for ( i=0 ; i < 2 ; i++ ) {
        printf("Hola soy un hilo!\n");
        sleep(1);
    }
    printf("Hilo terminado\n");
    return NULL;
}
int main(int argc, char ** argv) {
    pthread_t mythread;

    pthread_create(&mythread,NULL,thread_function, NULL);

    pthread_join(mythread,NULL);

    exit(0);
}
```

Atributos de un hilo

- Cada hilo tiene asociados un conjunto de atributos.
- Atributos representados por una variable de tipo **pthread_attr_t**.
- Los atributos controlan:
 - Un hilo es independiente o dependiente.
 - El tamaño de la pila privada del hilo.
 - La localización de la pila del hilo.
 - La política de planificación del hilo.

Atributos

- `int pthread_attr_init(pthread_attr_t * attr);`
 - Inicia una estructura de atributos de hilo.
- `int pthread_attr_destroy(pthread_attr_t * attr);`
 - Destruye una estructura de atributos de hilo.
- `int pthread_attr_setstacksize(pthread_attr_t * attr, int stacksize);`
 - Define el tamaño de la pila para un hilo
- `int pthread_attr_getstacksize(pthread_attr_t * attr, int *stacksize);`
 - Permite obtener el tamaño de la pila de un hilo.

Hilos dependientes e hilos independientes

- `int pthread_attr_setdetachstate(pthread_attr_t *attr,
int detachstate)`
 - Establece el estado de terminación de un proceso ligero.
 - Si "detachstate" = PTHREAD_CREATE_DETACHED el proceso ligero liberara sus recursos cuando finalice su ejecución.
 - Si "detachstate" = PTHREAD_CREATE_JOINABLE no se liberan los recursos, es necesario utilizar `pthread_join()`.
- `int pthread_attr_getdetachstate(pthread_attr_t *attr,
int *detachstate)`
 - Permite conocer el estado de terminación

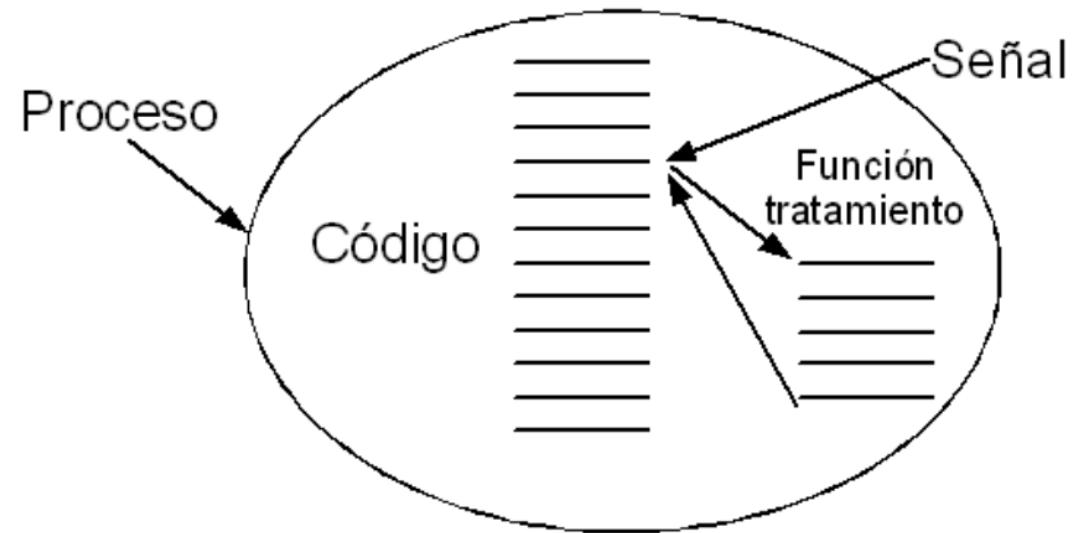
Señales

- Son un mecanismo que permite avisar a un proceso de la ocurrencia de un evento.
- Ejemplos:
 - Un proceso padre recibe la señal SIGCHLD cuando termina un proceso hijo.
 - Un proceso recibe una señal SIGILL cuando intenta ejecutar una instrucción máquina ilegal.

Son un mecanismo propio de los sistemas UNIX

Señales

- Las señales son interrupciones al proceso
- Envío o generación
 - Proceso -- Proceso (dentro del grupo) con el kill
 - SO -- Proceso



Señales

- Otras señales
 - SIGILL instrucción ilegal
 - SIGALRM vence el temporizador
 - SIGKILL mata al proceso
- Cuando un proceso recibe una señal:
 - Si está en ejecución: Detiene su ejecución en la instrucción máquina actual.
 - Si existe una rutina de tratamiento de la señal: Bifurcación para ejecutar la rutina de tratamiento.
 - Si la rutina de tratamiento no termina el proceso: Retorno al punto en que se recibió la señal.

Señales

- Otras señales
 - SIGALRM vence el temporizador
 - SIGKILL mata al proceso
- El SO las transmite al proceso
 - El proceso debe estar preparado para recibirla
 - Especificando un procedimiento de señal con sigaction.
 - Enmascarando la señal con sigprogmask.
 - Si no está preparado → → acción por defecto
 - El proceso, en general, muere.
 - Hay algunas señales que se ignoran o tienen otro efecto.
- El servicio pause para el proceso hasta que recibe una señal

Servicios POSIX para la gestión de señales

- `int kill(pid_t pid, int sig)`
 - Envía al proceso "pid" la señal "sig".
 - Casos especiales:
 - $\text{pid}=0 \rightarrow$ Señal a todos los procesos con gid igual al gid del proceso.
 - $\text{pid} < -1 \rightarrow$ Señal a todos los proceso con gid igual al valor absolute de pid.
- `int sigaction(int sig,
 struct sigaction *act,
 struct sigaction *oact)`
 - Permite especificar la acción a realizar como tratamiento de la señal "sig"
 - La configuración anterior se puede guardar en "oact".

La estructura sigaction

```
struct sigaction {  
    void (*sa_handler)(); /* Manejador */  
    sigset_t sa_mask; /* Señales bloqueadas */  
    int sa_flags;      /* Opciones */  
};
```

- Manejador:

- SIG_DFL: Acción por defecto (normalmente termina el proceso).
- SIG_IGN: Ignora la señal.
- Dirección de una función de tratamiento.

- Máscara de señales a bloquear durante el manejador.

- Opciones normalmente a cero.

Conjuntos de señales

- `int sigemptyset(sigset_t * set);`
 - Crea un conjunto vacío de señales.
- `int sigfillset(sigset_t * set);`
 - Crea un conjunto lleno con todas las señales posibles.
- `int sigaddset(sigset_t * set, int signo);`
 - Añade una señal a un conjunto de señales.
- `int sigdelset(sigset_t * set, int signo);`
 - Borra una señal de un conjunto de señales.
- `int sigismember(sigset_t * set, int signo);`
 - Comprueba si una señal pertenece a un conjunto.

Servicios POSIX para la gestión de señales

- **int pause(void)**
 - Bloquea al proceso hasta la recepción de una señal.
 - No se puede especificar un plazo para desbloqueo.
 - No permite indicar el tipo de señal que se espera.
 - No desbloquea el proceso ante señales ignoradas.
- **int sleep(unsigned int sec)**
 - Suspende un proceso hasta que vence un plazo o se recibe una señal.

Temporizadores

- El sistema operativo mantiene un temporizador por proceso (caso UNIX).
 - Se mantiene en el bloque de control de procesos (BCP) del proceso un contador del tiempo que falta para que venza el temporizador.
 - La rutina del sistema operativo actualiza todos los temporizadores.
 - Si un temporizador llega a cero se ejecuta la función de tratamiento.
- En UNIX el sistema operativo envía una señal SIGALRM al proceso cuando vence su temporizador.

Servicios POSIX para temporización

- **int alarm(unsigned int sec)**
- Establece un temporizador.
- Si el parámetro es cero, desactiva el temporizador.

Excepciones

- Win32 ofrece una gestión estructurada de excepciones.
- Extensión al lenguaje C para gestión estructurada de excepciones.
 - No es parte de ANSI/ISO C.

```
_try {
    /* Código principal */
}
_except (expr) {
    /* Tratamiento de
     excepción */
}
```

- La expresión de `_except` debe evaluarse a:
 - `EXCEPTION_EXECUTE_HANDLER`
 - Entra en el bloque.
 - `EXCEPTION_CONTINUE_SEARCH`
 - Propaga la excepción.
 - `EXCEPTION_CONTINUE_EXECUTION`
 - Ignora la excepción.

Código de excepción

- **DWORD GetExceptionCode()**
 - No es una llamada al sistema: Macro.
 - Solamente se puede usar dentro de tratamiento de excepciones.
 - Existe una lista larga de valores que puede devolver:
 - **EXCEPTION_ACCESS_VIOLATION**
 - **EXCEPTION_ILLEGAL_INSTRUCTION**
 - **EXCEPTION_PRIV_INSTRUCTION**
 - ...

Copia de cadenas segura

```
LPTSTR CopiaSegura(LPTSTR s1, LPTSTR s2) {  
    _try {  
        return strcpy(s1,s2);  
    }  
    _except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION?  
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {  
        return NULL;  
    }  
}
```

Entorno de un proceso en Windows

- **DWORD GetEnvironmentVariable (LPCTSTR lpszName , LPTSTR lpszValue , DWORD valueLength) ;**
 - Devuelve el valor de una variable de entorno.
- **BOOL SetEnvironmentVariable (LPCTSTR lpszName , LPTSTR lpszValue) ;**
 - Modifica o crea una variable de entorno.
- **LPVOID GetEnvironmentStrings () ;**
 - Obtiene un puntero a la tabla de variables de entorno.

Temporizadores en Windows

- **UINT SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc);**
 - Activa un temporizador y ejecuta la función lpTimerFunc cuando venza el tiempo.
 - La función debe cumplir con:
 - VOID TimerFunc(HWND hWnd, UINT uMsg, UINT idEvent, DWORD dwTime);
- **BOOL KillTimer(HWND hWnd, UINT uIdEvent);**
 - Desactiva un temporizador.
- **VOID Sleep(DWORD dwMilliseconds);**
 - Hace que el hilo actual se suspenda durante un cierto tiempo.

Puntos a recordar

- Un proceso puede tener varios hilos de ejecución.
- Una aplicación multihilo consume menos recursos que una aplicación multiproceso.
- Cada sistema operativo tiene un modo de soporte de hilos entre ULT y KLT.
- PTHREADS es una biblioteca de hilos de usuario.
- Win32 ofrece hilos en el núcleo con soporte para conjuntos de hilos (*Thread Pools*).

Puntos a recordar

- Las variables de entorno permiten pasar información a los procesos.
- Las señales POSIX se pueden ignorar o tratar.
- Los temporizadores tienen distinta resolución de POSIX en Win32.
- El tratamiento estructurado de excepciones permiten tratar situaciones anómalas mediante una extensión del lenguaje C.

Concurrencia

Concurrencia

Concurrencia.

Condiciones de carrera.

Exclusión mutua y sección crítica.

Semáforos.

El problema del productor consumidor.

El problema de los lectores escritores.

Concurrencia

- Los temas centrales del diseño de sistemas operativos están todos relacionados con la gestión de procesos e hilos:
 - Multiprogramación. Gestión de múltiples procesos dentro de un sistema monoprocesador.
 - Multiprocesamiento. Gestión de múltiples procesos dentro de un multiprocesador.
 - Procesamiento distribuido. Gestión de múltiples procesos que ejecutan sobre múltiples sistemas de cómputo distribuidos.

Concurrencia

La concurrencia aparece en tres contextos diferentes:

Múltiples aplicaciones. La multiprogramación fue ideada para permitir compartir dinámicamente el tiempo de procesamiento entre varias aplicaciones activas.

Aplicaciones estructuradas. Como extensión de los principios del diseño modular y de la programación estructurada, algunas aplicaciones pueden ser programadas eficazmente como un conjunto de procesos concurrentes.

Estructura del sistema operativo. Las mismas ventajas constructivas son aplicables a la programación de sistemas y, de hecho, los sistemas operativos son a menudo implementados en sí mismos como un conjunto de procesos o hilos.

PRINCIPIOS DE LA CONCURRENCIA

En un sistema multiprogramado de procesador único, los procesos se entrelazan en el tiempo para ofrecer la apariencia de ejecución simultánea. Aunque no se consigue procesamiento paralelo real, e ir cambiando de un proceso a otro supone cierta sobrecarga, la ejecución entrelazada proporciona importantes beneficios en la eficiencia del procesamiento y en la estructuración de los programas. En un sistema de múltiples procesadores no sólo es posible entrelazar la ejecución de múltiples procesos sino también solaparlas.

Dos procesos son concurrentes cuando se ejecutan de manera que sus intervalos de ejecución se solapan.



Si hay concurrencia



No hay concurrencia

Tipos de concurrencia

Concurrencia aparente: Hay más procesos que procesadores.

- Los procesos se multiplexan en el tiempo.
- Pseudoparalelismo

1 CPU



2 CPUs



Concurrencia real: Cada proceso se ejecuta en un procesador.

- Se produce una ejecución en paralelo.
- Paralelismo real.

4 CPUs

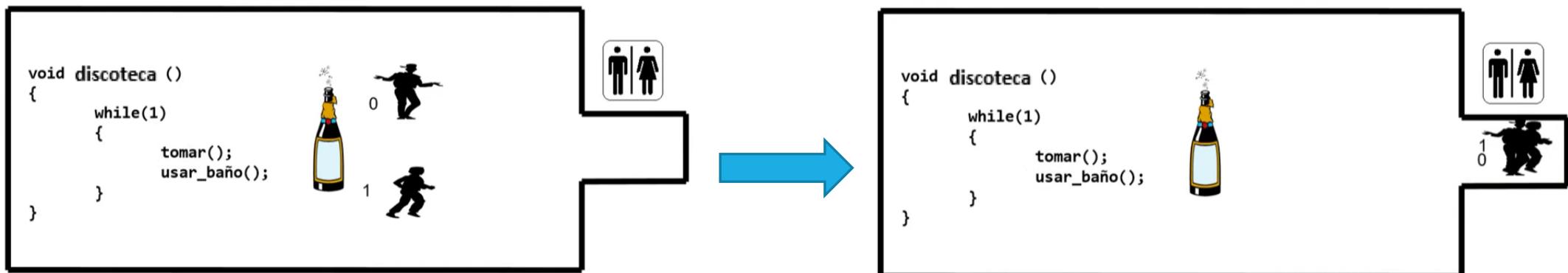


Problemas con la ejecución concurrente

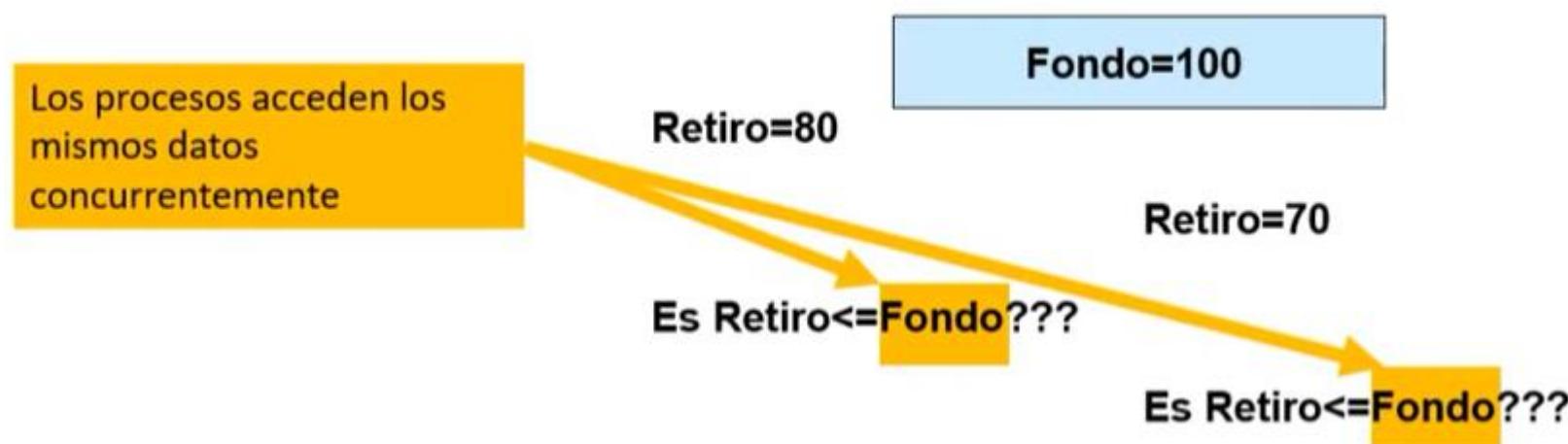
Procesos concurrentes (o hilos) necesitan compartir datos y recursos

Si no hay acceso controlado para acceder a esos datos compartidos, algunos procesos obtendrán una visión inconsistente de los datos.

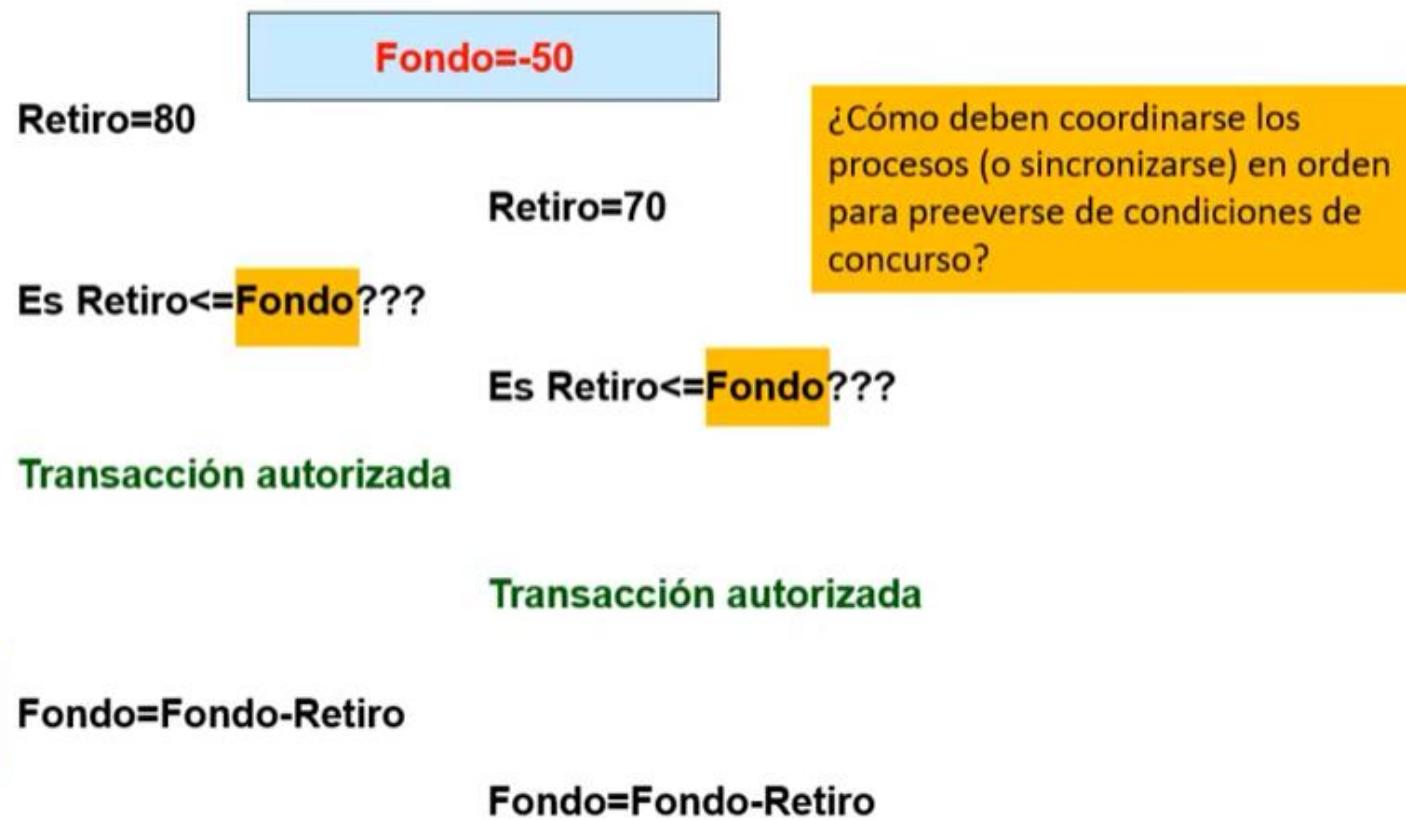
La acción ejecutada por procesos concurrentes dependerá del orden en como su ejecución se intercala.



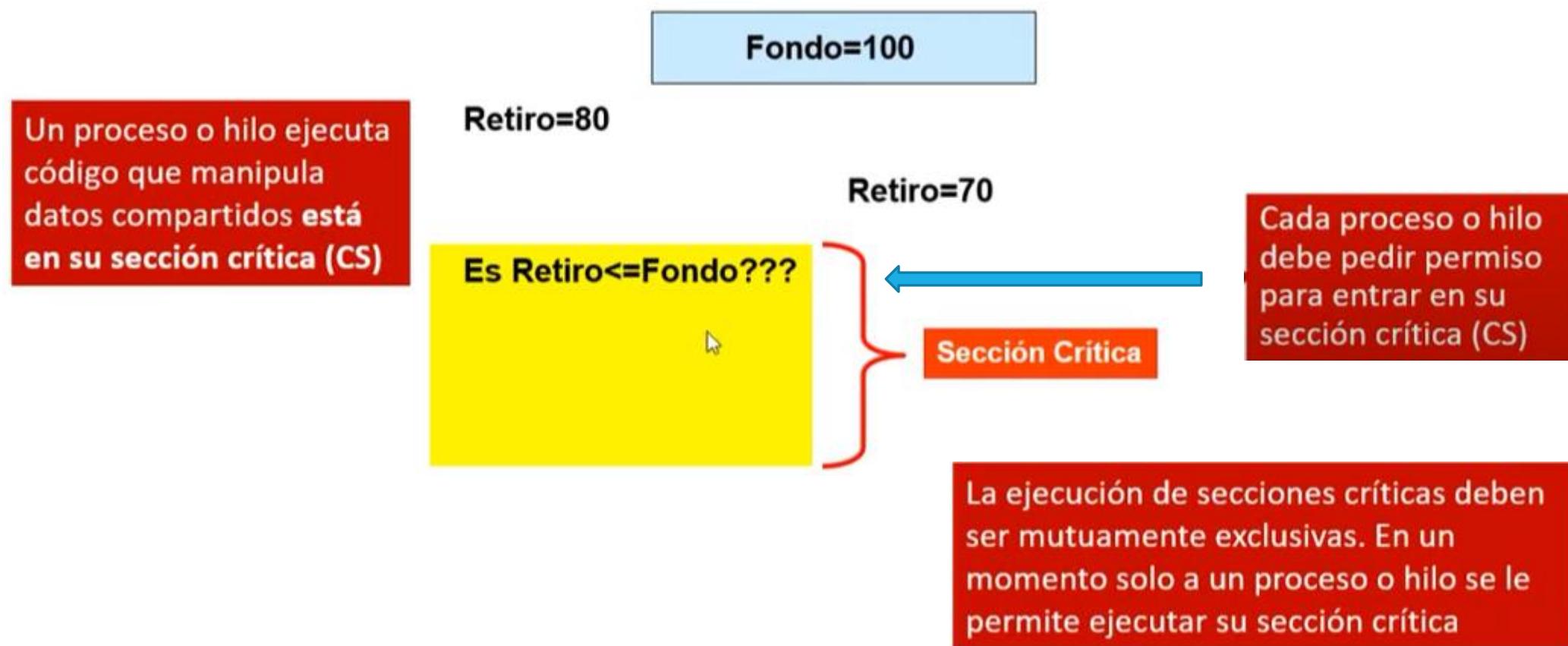
Ejemplo transacción bancaria



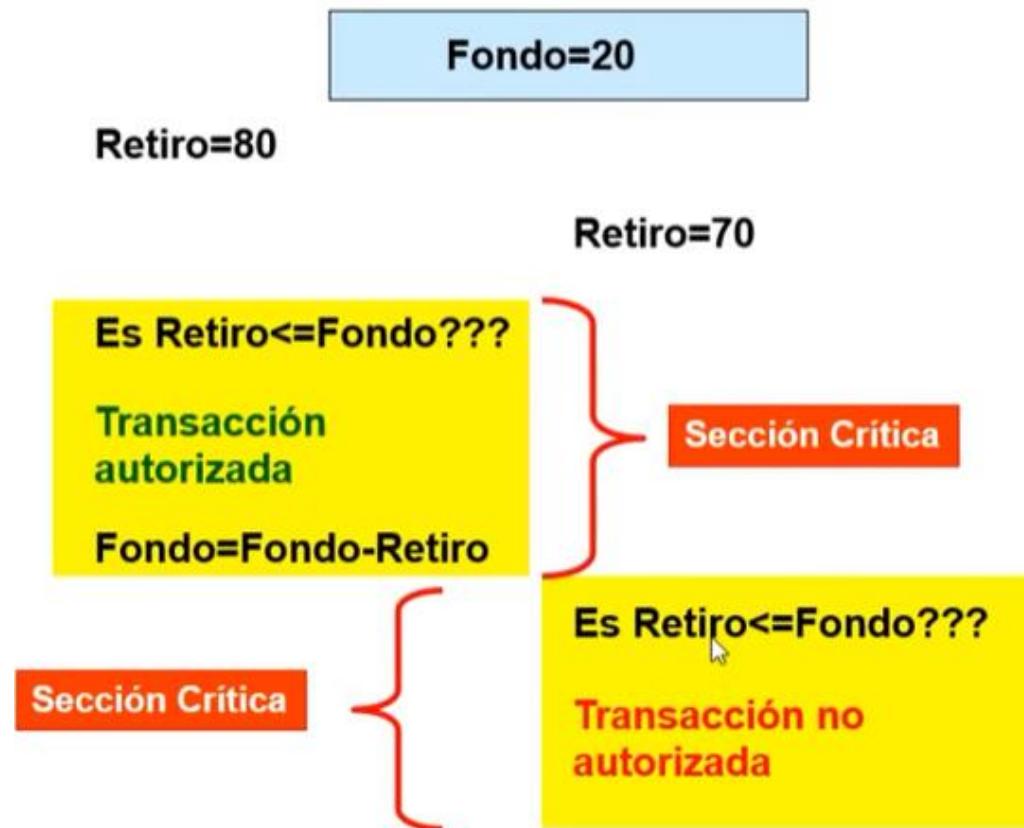
Ejemplo transacción bancaria



Ejemplo transacción bancaria



Ejemplo transacción bancaria



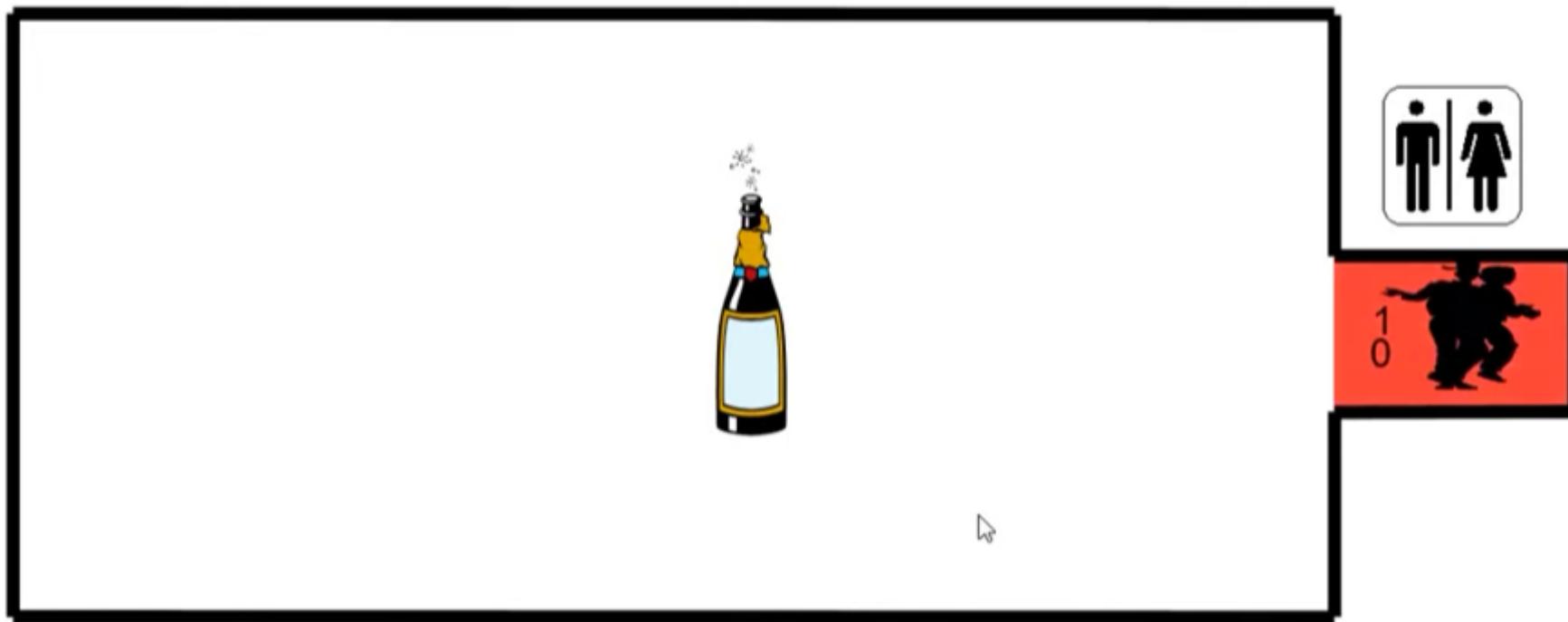
Cada proceso o hilo se ejecuta a una velocidad diferente de cero.

```
while(1)
{
    sección de entrada
    sección crítica
    sección de salida
    sección restante
}
```

No se asume nada en la velocidad relativa de n procesos o hilos

Requisitos para resolver el problema

1) Exclusión mutua



Requisitos para resolver el problema

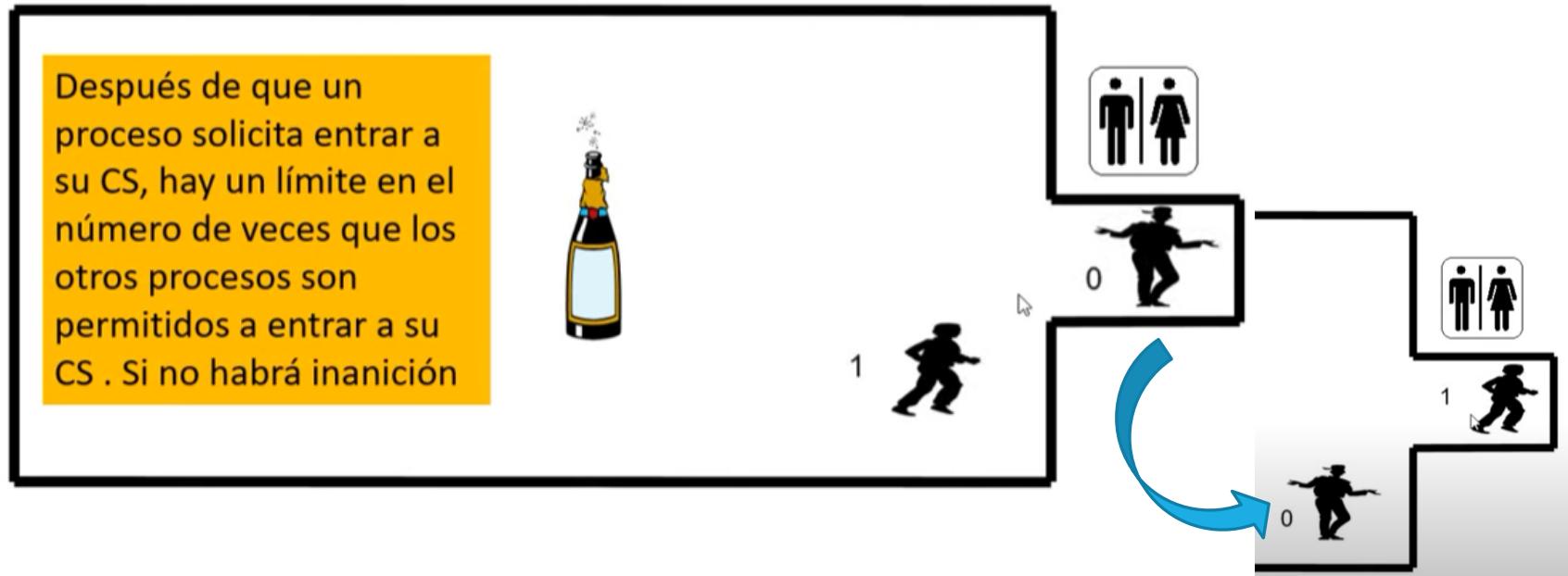
2) Progreso



Esta selección no puede posponerse
indefinidamente

Requisitos para resolver el problema

3) Espera limitada



Tipos de soluciones

- Software
 - Algoritmos cuya correctud no dependen de ninguna suposición.
- Hardware
 - Dependen de algunas instrucciones maquina especiales.
- Operación del sistema
 - Provee algunas funciones y estructuras de datos al programador para preservar esto. Es decir, son soluciones que provee el sistema operativo.

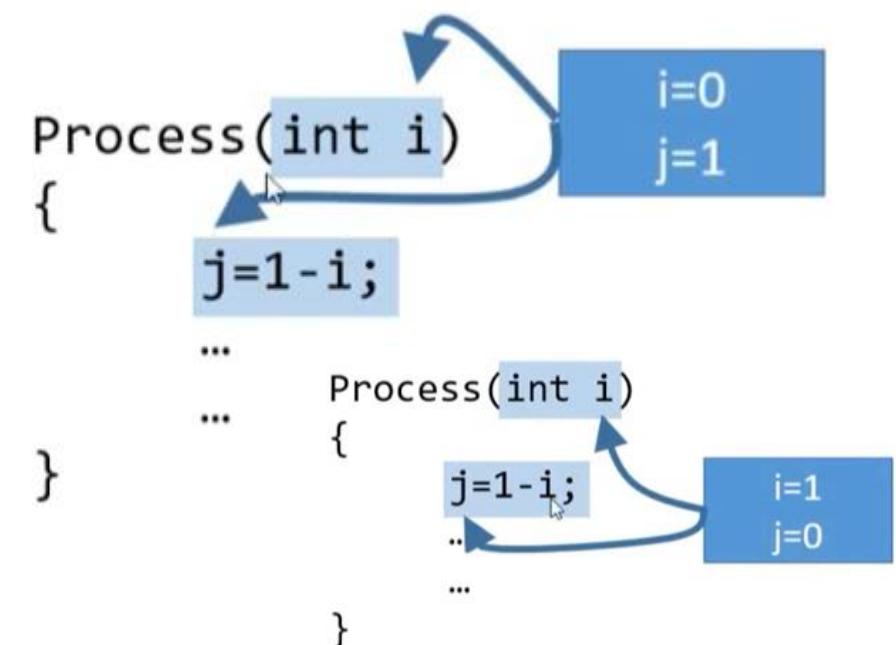
Soluciones por software

Iniciamos con 2 procesos o hilos: P0 y P1

- Los algoritmos 1, 2 y 3 son incorrectos
- El cuarto intento es correcto pero aun puede mejorar
- Los algoritmos de Decker y Peterson son los correctos

Notaciones

- Cuando tenemos los procesos o hilos p_i , p_j , i refiere al proceso actual, j refiere al otro proceso.



Problema baño disco

```
int turno=0;
```

El primer intento fué poner una pizarra llamada **turno** afuera del baño

```
Process Pi:
```

```
repeat
```

```
    while(turno!=i){};
```

CS

Cuando quiere entrar examina si la pizarra tiene su turno

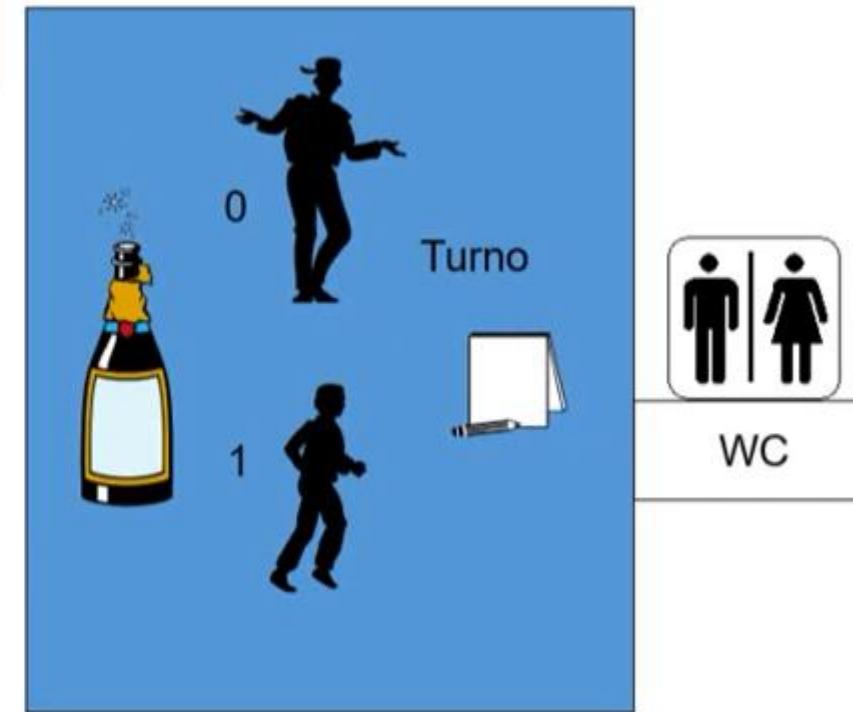
```
    turno=j;
```

Entra al baño

RS

Al salir, escribe en la pizarra el turno del otro

```
forever
```



Intento 1

Process P0:

Repeat

while (turno!=0) {};

CS

turno=1;

RS

forever

Process P1:

Repeat

while (turno!=1) {};

CS

turno=0;

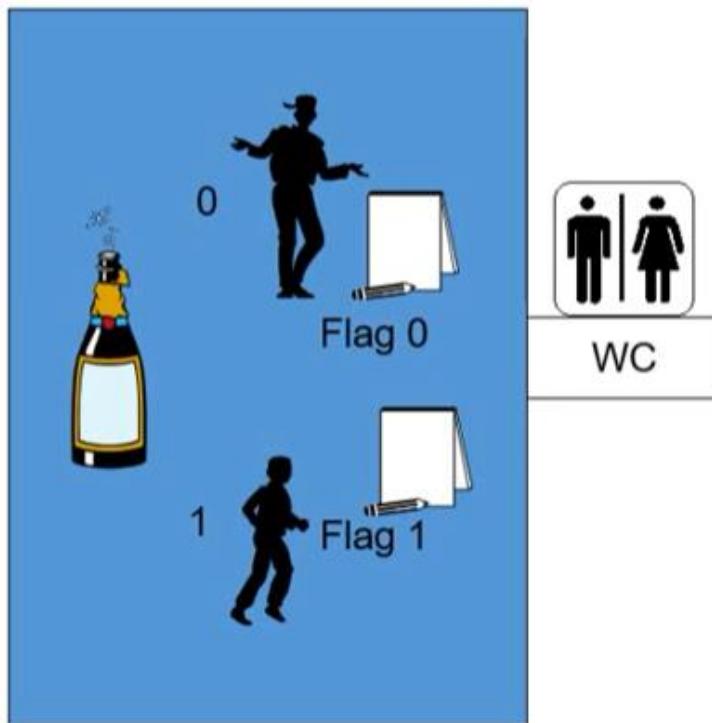
RS

forever

Intento 1

| Proceso 0 | Proceso 1 | turno |
|---------------------------------|---------------------------------|-------|
| <code>while(turno!=0){};</code> | | 0 |
| | <code>while(turno!=1){};</code> | 0 |
| CS | | 0 |
| <code>turno=1;</code> | | 1 |
| | <code>while(turno!=1){};</code> | 1 |
| | CS | 1 |

Intento 2



```
boolean flag[2];
```

Dos pizarras afuera del baño, una para cada uno

```
Process Pi:  
repeat
```

Cuando uno quiere entrar primero examina la pizarra del otro y mientras está en verdadero, espera

```
while(flag[j]) {};
```

Pone su pizarra en verdadero

```
CS
```

```
flag[i]:=true;
```

Entra al baño

```
RS
```

```
flag[i]:=false;
```

Al salir, pone su pizarra en falso

```
forever
```

| Proceso 0 | Proceso 1 | Flag[0] | Flag[1] |
|--------------------------------|--------------------------------|---------|---------|
| <code>while(flag[1]){};</code> | | False | False |
| | <code>while(flag[0]){};</code> | False | False |
| <code>flag[0]:=true;</code> | | True | False |
| | <code>flag[1]:=true;</code> | True | True |
| CS | | True | True |
| | CS | True | True |

Intento 3

```
boolean flag[2];
```

Process Pi:

repeat

```
    flag[i]=true;
```

```
    while(flag[j]) {};
```

CS

```
    flag[i]=false;
```

RS

forever

Ahora... al querer entrar al baño, primero
pone su pizarra en verdadero

Examina la pizarra del otro
y mientras está en
verdadero, espera

Entra al baño

Intento 3

Process P0:

repeat

flag[0]:=true;
 while(flag[1]){};

CS

flag[0]:=false;

RS

forever

Process P1:

repeat

flag[1]:=true;
 while(flag[0]){};

CS

flag[1]:=false;

RS

forever

Intento 3

| Proceso 0 | Proceso 1 | Flag[0] | Flag[1] |
|--------------------------------|--------------------------------|---------|---------|
| | | False | False |
| <code>flag[0]:=true;</code> | | True | False |
| | <code>flag[1]:=true;</code> | True | True |
| <code>while(flag[1]){};</code> | | True | True |
| | <code>while(flag[0]){};</code> | True | True |

Intento 4

Comprueba
nuevamente la
pizarra del otro

Process Pi:
repeat

flag[i]:=true;
while(flag[j])
{

Al querer entrar al baño, primero pone su pizarra en verdadero

Examina la pizarra del otro y mientras está en
verdadero

Quita el verdadero de su pizarra

espera un tiempo aleatorio

Espera un tiempo

flag[i]:=true;

Vuelve a poner su pizarra en
verdadero

}

CS

Entra al baño

flag[i]:=false;

pone su pizarra en falso

RS

forever

Intento 4

Process P0:

```
repeat
    flag[0]:=true;
    while(flag[1])
    {
        flag[0]:=false;
        espera
        flag[0]:=true;
    };
    CS
    flag[0]:=false;
    RS
forever
```

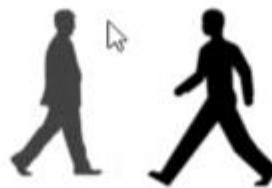
Process P1:

```
repeat
    flag[1]:=true;
    while(flag[0])
    {
        flag[1]:=false;
        espera
        flag[1]:=true;
    };
    CS
    flag[1]:=false;
    RS
forever
```

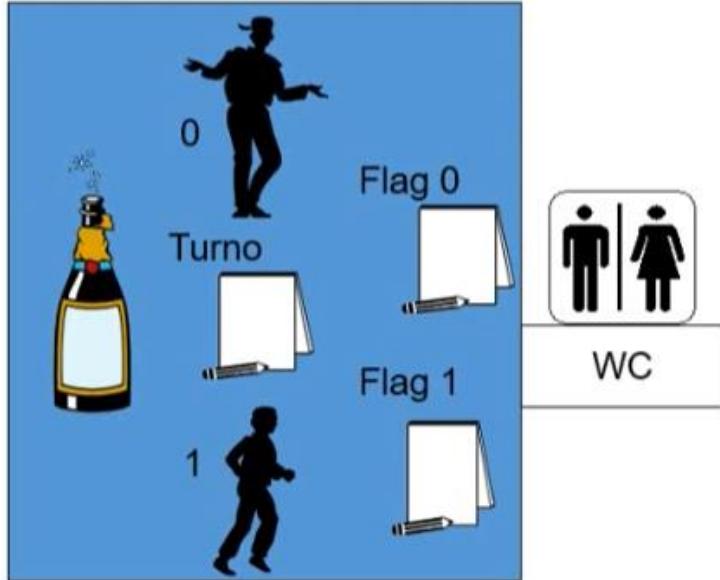
| Proceso 0 | Proceso 1 | Flag[0] | Flag[1] |
|-----------------|--------------------|---------|---------|
| flag[0]:=true; | flag[1]:=true; | False | False |
| while(flag[1]) | while(flag[0]) | True | True |
| ----- | | | |
| flag[0]:=false; | flag[1]:=false; | False | False |
| espera | espera | | |
| flag[0]:=true; | flag[1]:=true; | True | True |
| while(flag[1]) | while(flag[0]) | | |
| flag[0]:=false; | flag[1]:=false; | False | False |
| espera | espera | | |
| flag[0]:=true; | sigue en la espera | True | |
| while(flag[1]) | flag[1]:=true; | | True |
| CS | while(flag[0]) | | |
| flag[0]:=false; | | False | |

Livelock

- Dos personas, al encontrarse en un pasillo angosto avanzando en sentidos opuestos
- Cada una trata de ser amable moviéndose a un lado para dejar a la otra persona pasar
- Terminan moviéndose de lado a lado sin tener ningún progreso
- Ambos se mueven hacia el mismo lado, al mismo tiempo.



El problema del baño con el algoritmo de Dekker



```
boolean flag[2]={false,false};  
int turno=0;  
  
Process Pi:  
Repeat  
    flag[i]=true;  
    while(flag[j])  
    {  
        flag[i]=false;  
        while(turno!=i);  
        flag[i]=true;  
    }  
    CS  
    turno=j;  
    RS  
forever
```

Tres pizarras afuera del baño, dos banderas y otra que indica a quien le toca el turno

Al querer entrar al baño pone su bandera en verdadero

Examina la bandera del otro y si está en verdadero entra al ciclo

Pone su bandera en falso

Espera a que sea su turno

Usa el baño

Pone su bandera en verdadero

Pone el turno para el otro

Pone su bandera en falso



Process P0:

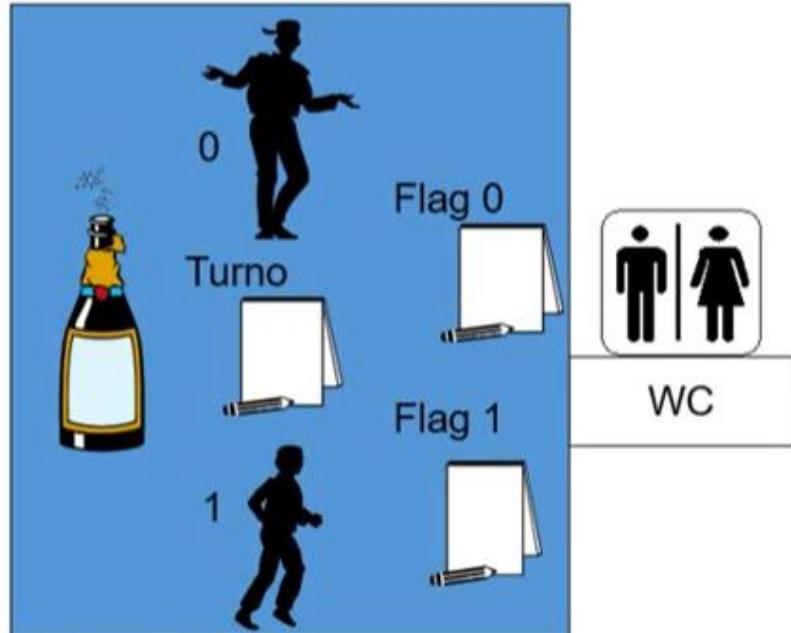
```
repeat
    flag[0]=true;
    while(flag[1])
    {
        flag[0]=false;
        while(turno!=0);
        flag[0]=true;
    }
    CS
    turno=1;
    flag[0]=false;
    RS
forever
```

Process P1:

```
repeat
    flag[1]:=true;
    while(flag[0])
    {
        flag[1]=false;
        while(turno!=1);
        flag[1]=true;
    }
    CS
    turno=0;
    flag[1]=false;
    RS
forever
```

| Proceso 0 | Proceso 1 | Flag[0] | Flag[1] | Turno |
|------------------|------------------|---------|---------|-------|
| flag[0]=true; | | False | False | 0 |
| - while(flag[1]) | flag[1]=true; | True | | True |
| flag[0]=false; | while(flag[0]) | | | |
| while(turno!=0); | flag[1]=false; | False | | False |
| flag[0]=true; | while(turno!=1); | | | |
| while(flag[1]) | while(turno!=1); | True | | |
| CS | while(turno!=1); | | | |
| turno=1; | while(turno!=1); | | | 1 |
| flag[0]=false; | while(turno!=1); | False | | |
| | flag[1]=true; | | True | |
| | while(flag[0]) | | | |
| | CS | | | |

Problema del baño con el algoritmo de Peterson



```
boolean flag[2]={false, false};  
int turno=0;  
  
Process Pi:  
repeat  
    flag[i]=true;  
    turno=j;  
    while(flag[j] && turno==j) {};  
  
    CS  
    flag[i]=false;  
    RS  
forever
```

Tres pizarras afuera del baño, dos banderas y una que indica a quien le toca el turno

Al querer entrar al baño pone su pizarra bandera en verdadero y la pizarra turno con número del otro borrachito

Si la bandera del otro está en verdadero y es turno del otro entonces tengo que esperar

Usa el baño

Pone su bandera en falso

Algoritmo de Peterson

| Proceso 0 | Proceso 1 | Flag[0] | Flag[1] | Turno |
|---------------------------------|---------------------------------|---------|---------|-------|
| | | False | False | 0 |
| Flag[0]=True; | | True | False | 0 |
| | Flag[1]=True; | True | True | 0 |
| Turno=1 | | True | True | 1 |
| | Turno=0 | True | True | 0 |
| while (flag[1]&&turno==1){}; | | True | True | 0 |
| | while (flag[0]&&turno==0){}; | True | True | 0 |
| CS | | True | True | 0 |
| Flag[0]=False | Puede pasar | False | True | 0 |
| | while (flag[0]&&turno==0){}; | False | True | 0 |
| | CS | | | |

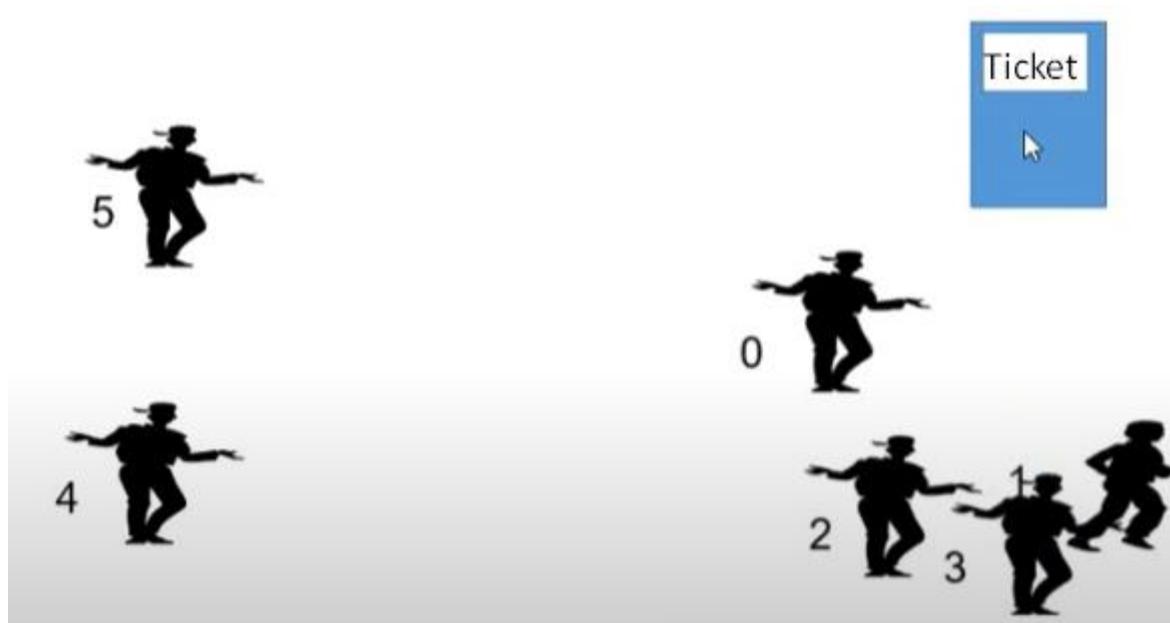
Soluciones por software

Los algoritmos de Dekker y Peterson son soluciones correctas para resolver el problema de concurrencia en la sección critica.

Funcionan muy bien para dos procesos P0 y P1.

¿Qué sucede si son 3 o mas procesos?

Algoritmo de la panadería



```
int numero[N];
boolean seleccionando[N];
```

Datos compartidos

```
Proceso Pi:  
repeat  
    seleccionando[i]:=true;  
    numero[i]:=max(numero[0]..numero[n-1])+1;  
    seleccionando[i]:=false;  
    for j:=0 to n-1 do {  
        while (seleccionando[j]) {};  
        while (numero[j]!=0 and (numero[j]<numero[i]  
        or numero[i]==numero[j] and j<i)){};
```

Pone su bandera de seleccionando

Busca el número más alto y toma ese número más 1

Quita su bandera de seleccionando

Recorre todos los procesos

Si hay uno que esté seleccionando su número, espera a que termine de seleccionar

CS Usa la sección critica

```
    numero[i]:=0;  
    RS      Pone su número en 0 indicando que ya no quiere entrar
```

Espera en el ciclo while si:

- Hay un proceso que tiene número diferente de 0 y menor
- O si hay otro proceso con el mismo número, que tenga id menor

```
    forever
```