

# TEMA-2-PROCESOS-Y-HEBRAS.pdf



mrg23



Sistemas Operativos



2º Grado en Ingeniería Informática



Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación  
Universidad de Granada



MÁSTER EN

## Inteligencia Artificial & Data Management

MADRID

Formamos  
**talento** para un futuro  
**Sostenible**

saber más



Esto no son apuntes pero **tiene un 10 asegurado** (y lo vas a disfrutar igual).

Abre la Cuenta NoCuenta con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Me interesa

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en [ing.es](#)



## TEMA 2: PROCESOS Y HEBRAS

### 1. DISEÑO E IMPLEMENTACIÓN DE PROCESOS Y HEBRAS

#### a. CONCEPTO FUNDAMENTAL

Los sistemas operativos (SO) están diseñados para gestionar los recursos de hardware y permitir la ejecución de múltiples aplicaciones de manera eficiente. Para lograrlo, simulan la ejecución concurrente de varios procesos, permitiendo que cada uno de ellos tenga la impresión de estar ejecutándose en su propia CPU.

SE PODRÍA CONSTRUIR EL SO COMO:

```
for (;;) {  
    if (aplicación())      Ejecutar();  
    if (MensajeRed())      ObtenMensaje();  
    if (TeclaPulsada())    ObtenTecla();  
    if (BloqueDiscoListo()) ObtenBloque();  
    ...  
}
```



#### b. PROCESO

Un proceso es una **instancia de un programa en ejecución**. Cada proceso contiene toda la información necesaria para su ejecución y se compone de varias partes (del programa eje.):

- **Conjunto de cabeceras**
- **Texto del programa**: Contiene las instrucciones que serán ejecutadas.
- **Datos**: Divididos en dos categorías:
  - o Datos inicializados: Variables que se inicializan antes de su uso.
  - o Datos no inicializados (bss): Variables que se declaran pero no se inicializan.
- **Pila**: Utilizada para la ejecución de funciones, donde se almacenan parámetros y variables.
- **Otras secciones**: Como tablas de símbolos que facilitan la ejecución del programa.

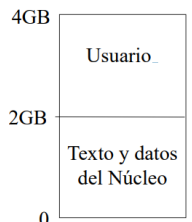
El núcleo divide el proceso en **regiones** (segmentos):

- Región de texto: sólo lectura
- Región de datos
- Región de pila

#### c. ESTRUCTURA DEL ESPACIO DE MEMORIA VIRTUAL

La **estructura** del espacio de memoria virtual de un proceso se divide en dos partes:

- **Código y datos del núcleo**: Ocupa la mitad inferior del espacio (2GB en un sistema de 32 bits)
- **Pila del usuario**: Se crea automáticamente y se ajusta dinámicamente a medida que se realizan llamadas a funciones (Con direcciones de 32 bits, el espacio virtual máximo para un proceso es de 4GB).

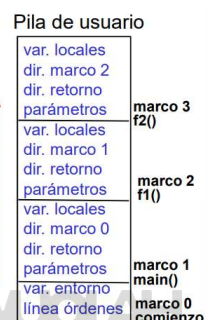


#### PARTES DE UN PROCESO

La pila de usuario se crea automáticamente y su tamaño se ajusta dinámicamente

La pila contiene **marcos de pila lógicos** que se añaden cuando se llama a una función y se eliminan cuando se finaliza la ejecución de esa función.

Como un proceso puede ejecutarse en modo supervisor y modo usuario, existe **una pila para cada modo**.



Consulta condiciones aquí



do your thing

#### d. CONTEXTO DEL PROCESO

Para representar un proceso, se necesita almacenar la siguiente información que nos de el **estado de ejecución** de un programa:

- Código y datos del programa.
- Pila de ejecución: Mantiene el estado actual de la ejecución.
- PC (Contador de Programa): Indica la siguiente instrucción a ejecutar.
- Registros de la CPU: Valores actuales que se están utilizando.
- Recursos del sistema y su auditoría: Información sobre memoria, archivos abiertos, y otros recursos.

## 2. ESTADOS DEL PROCESO

### a. ESTADOS Y TRANSICIONES

Un proceso puede estar en **diferentes estados de ejecución** que caracteriza lo que hace:

- **Nuevo**: El proceso ha sido creado pero aún no está listo para ejecutarse.
- **Preparado**: El proceso está en memoria y listo para ejecutarse, pero no está en ejecución.
- **Ejecutándose**: El proceso está actualmente en ejecución.
- **Bloqueado**: El proceso no puede continuar hasta que ocurra un evento específico (como completar una operación de entrada/salida).
- **Finalizado**: El proceso ha terminado su ejecución.

Conforme un programa se ejecuta, pasa de un estado a otro.

La transición entre estos estados puede ser desencadenada por eventos como interrupciones o finalización de operaciones de E/S.

### b. DIAGRAMAS DE ESTADOS

El diagrama de estados representa las transiciones de un proceso. Por ejemplo:

→ Un proceso en estado **Bloqueado** puede cambiar a **Preparado** cuando el evento por el que estaba esperando se completa.



Proceso en primer plano: `%gcc programa.c`

Proceso de fondo: `%gcc programa.c`  
`% cat file.txt`

## 3. PLANIFICACIÓN Y COLAS DE ESTADOS

### a. PCBs (PROCESS CONTROL BLOCKS)

El sistema operativo mantiene una estructura llamada **PCB** para cada proceso. Esta estructura contiene información crítica

a, como el estado del proceso, el contador de programa, y la lista de recursos asignados.

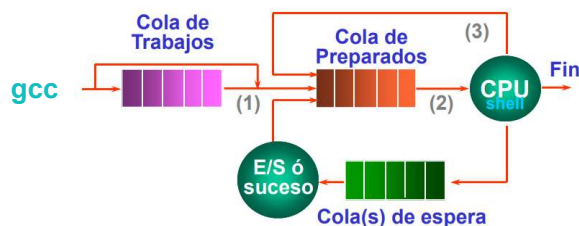
Los PCBs se organizan en colas según el estado del proceso:

- **Cola de trabajos**: Todos los procesos en el sistema
- **Cola de preparados**: Procesos listos para ejecutarse.
- **Cola de espera**: Procesos que están esperando recursos específicos o eventos de E/S.

Conforme un proceso cambia de estado, su PCB es retirado de una cola y encolado en otra.

## b. MODELO DE SISTEMA

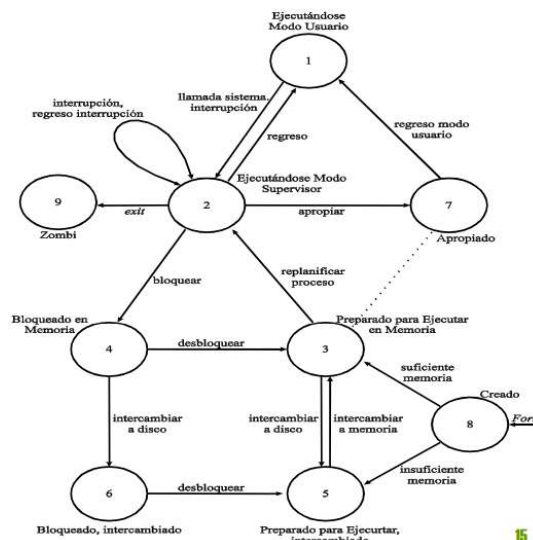
Un sistema operativo puede ser modelado como un conjunto de procesadores y colas, donde los procesos se gestionan según su estado y los recursos disponibles.



## ESTADOS/TRANSICIONES EN UNIX

Diagrama con las transiciones de estado típicas de un proceso en el sistema Unix. Los puntos clave incluyen:

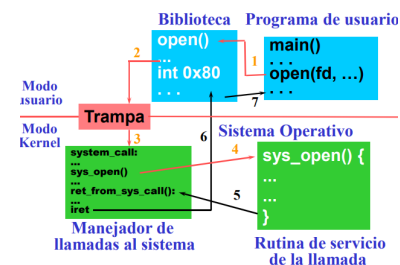
- Un proceso puede estar en **modo usuario** o **modo kernel**.
- Las transiciones entre estos estados son controladas por el sistema operativo, y el proceso no puede desplazarse a otro que esté ejecutándose en modo kernel.
- El proceso invoca funciones del sistema operativo a través de interrupciones, llamadas al sistema, o trampas (trampas). Ejemplos:
  - Llamadas como **open()** hacen una trampa al kernel, invocando el sistema operativo para realizar tareas específicas (como abrir archivos).
- El 3 y el 7 son realmente el mismo estado



## LLAMADAS AL SISTEMA

Concepto de **cambio de contexto** en Unix.

- En Unix, cuando se invoca una llamada al sistema (por ejemplo, para abrir un archivo o realizar una operación de E/S), el proceso actual se detiene temporalmente y el sistema operativo toma el control.
- Esto requiere que el estado del proceso sea guardado y restaurado. El proceso que está en ejecución pasa de **modo usuario** a **modo kernel**.



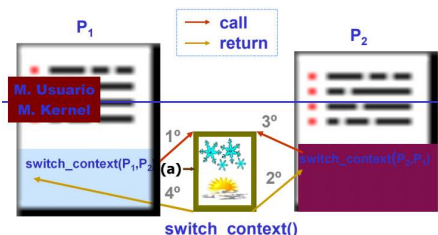
En esta parte se ilustra cómo se salva el contexto del proceso (registros, pila, PC, etc.) antes de pasar a ejecutar una tarea del kernel, y cómo se reanuda el proceso cuando la tarea del kernel ha finalizado.

## 4. CAMBIO DE CONTEXTO

### a. PROCESO DE CAMBIO DE CONTEXTO

El cambio de contexto es un mecanismo crucial que permite al SO suspender la ejecución de un proceso y guardar su estado para poder reanudarlo más tarde. Este proceso incluye varios pasos:

- Guardar el contexto del proceso actual: Incluye el PC, los registros de la CPU y la pila.
- Cargar el contexto del nuevo proceso: Se restaura el estado del proceso que se va a ejecutar. Este mecanismo asegura que los procesos pueden ser interrumpidos y reanudados sin perder su estado.





Esto no son apuntes pero **tiene un 10 asegurado** (y lo vas a disfrutar igual).

Abre la Cuenta NoCuenta con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Me interesa

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 5/5 de mayor riesgo.



ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en [ing.es](https://www.ing.es)






## ILUSTRACIÓN DEL CAMBIO DE CONTEXTO: SUPUESTOS

- Suponemos dos procesos:
  - P1 está ejecutando la instrucción n que es una llamada al sistema.
  - P2 se ejecutó anteriormente y ahora está en el estado preparado esperando su turno.

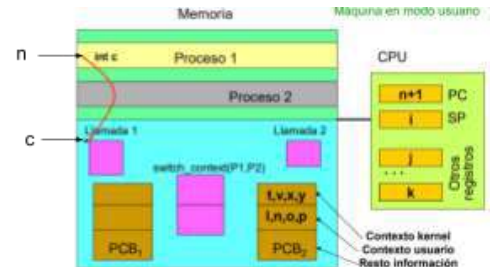
- Convenio:

-  Código del SO
-  Estructura de datos

-  Flujo de control
-  Salvar estructuras de datos
-  Instrucción i-ésima a ejecuta

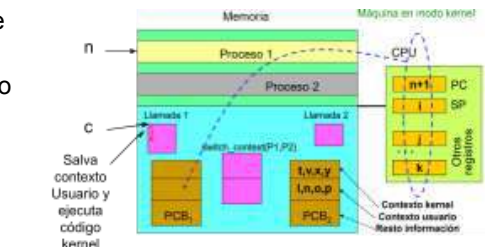
### 1º - P<sub>1</sub> EJECUTA UNA LLAMADA AL SISTEMA N

El proceso P1 se está ejecutando en modo usuario, y llega una instrucción que requiere una llamada al sistema. El contexto de P1 (sus registros, pila y contador de programa) se guarda, y se cambia al modo kernel para que el sistema operativo pueda manejar la llamada.



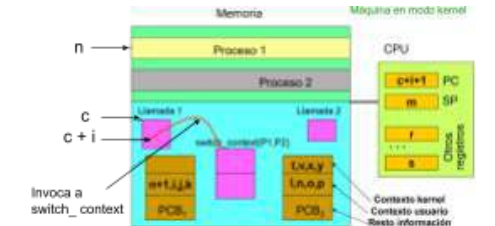
### 2º - SALVA CONTEXTO USUARIO Y EJECUTA F<sup>u</sup>on KERNEL

Se guarda todo el contexto de P1 (información de la pila de usuario y registros) para que pueda ser reanudado más tarde. Luego, el sistema operativo invoca la función `switch_context()`.



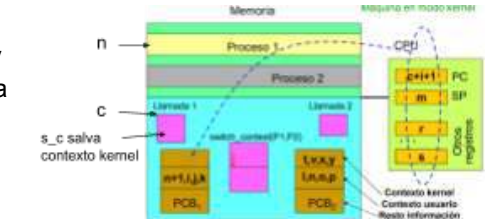
### 3º - PARAR PROCESO, INVOCA A CAMBIO\_CONTEXTO

El sistema operativo cambia el contexto del proceso, salvando el contexto del kernel para P1 y restaurando el de P2. El sistema carga el contexto de P2 para que pueda retomar su ejecución en el punto donde fue interrumpido previamente.



### 4º - CAMBIO\_CONTEXTO () SALVA CONTEXTO KERNEL

Finalmente, el contexto de P2 se carga completamente (incluidos los registros y el PC), y el sistema operativo devuelve el control a P2 para que continúe su ejecución en modo usuario.



Este proceso es crucial para la multitarea en los sistemas operativos, ya que permite que múltiples procesos se ejecuten "simultáneamente", conmutando entre ellos sin perder el estado de ninguno.

Consulta condiciones aquí



do your thing

Esta profundización en cómo el sistema salva el **contexto del kernel** durante el cambio de contexto. Después de que P1 se detiene, el sistema operativo guarda la información de su contexto en modo kernel (además del contexto en modo usuario). Esto incluye los registros de la CPU, la pila y otros datos esenciales para poder reanudar la ejecución más tarde.

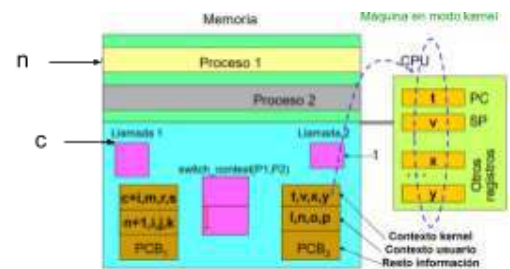
¿Cómo estamos? Llegados a este punto P1 está detenido, “congelado” y nos disponemos a reanudar, “descongelar”, a P2 (que previamente habíamos parado en algún instante anterior). Es decir, estamos en el punto marcado como (a) en la transparencia 18.

## 5º - REPONE CONTEXTO KERNEL DE P<sub>2</sub>

El quinto paso es cuando el sistema comienza a restaurar el contexto del proceso P2. Se destacan los siguientes puntos:

1. **Restauración del Contexto del Kernel de P<sub>2</sub>** : El sistema operativo carga el contexto kernel del proceso P2, lo que incluye registros, pila y otros datos de control necesarios para que el proceso pueda reanudar su ejecución en el núcleo.
2. **P2 en Modo Kernel** : Después de restaurar el contexto del kernel, el proceso P2 está listo para continuar su ejecución en modo kernel.

Este paso es clave para asegurar que P2 pueda retomar su ejecución sin perder información crítica.

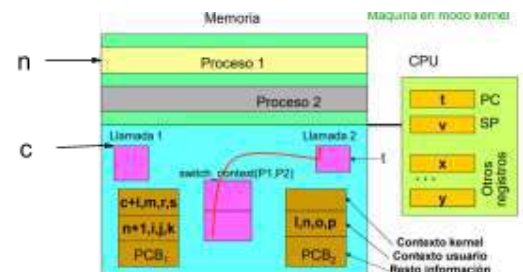


## 6º - EL KERNEL TERMINA LA F<sup>ON</sup> QUE INICIO DE P<sub>2</sub>

En este paso, el kernel termina de ejecutar cualquier función pendiente que P2 tenía cuando fue interrumpido. El kernel ha restaurado completamente el estado de P2 y ahora puede continuar ejecutando. Se destacan:

- El kernel completa la función o llamada al sistema que P2 había comenzado antes de ser interrumpido.
- El proceso P2 está listo para retornar al modo usuario.

Este paso finaliza la ejecución en modo kernel y prepara a P2 para volver al modo usuario.

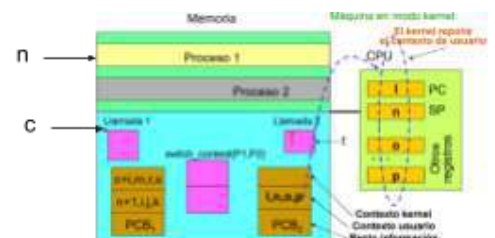


## 7º - FINALIZA F<sup>ON</sup>, RETORNA A MODO USUARIO

Qué compone el **contexto de un proceso**:

1. **Contexto de usuario**: Incluye las secciones de texto, datos, y la pila del proceso.
2. **Contexto de registros**: Involucra el PC (contador de programa), los registros generales de la CPU, y el puntero de pila (SP).
3. **Contexto de sistema**: Compuesto por una parte fija (como la entrada de tabla de procesos, TP) y una parte dinámica que varía (como las capas de contexto).

El contexto de proceso es toda la información que necesita ser almacenada y restaurada para que un proceso pueda ser suspendido y reanudado sin problemas.



1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en [ing.es](http://ing.es)

Que te den **10 € para gastar**  
es una fantasía.  
ING lo hace realidad.

Abre la **Cuenta NoCuenta** con el código  
WUOLAH10, haz tu primer pago y llévate 10 €.

**Quiero el cash**

[Consulta condiciones aquí](#)

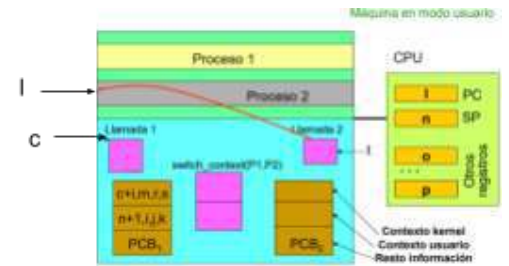


do your thing

## 8º - REANUDAMOS EJECUCIÓN DE P<sub>2</sub>

Ofrece observaciones sobre el proceso de cambio de contexto. Cuando conmutamos al proceso P<sub>2</sub>, este ya tiene una estructura de **PCB (Process Control Block)** preparada, y por lo tanto su estado ha sido previamente guardado. Si el proceso P<sub>2</sub> es recién lanzado, el sistema operativo ajusta su contexto de forma que se ejecute correctamente desde su primera instrucción.

El sistema operativo debe preparar el contexto de usuario y el contexto del kernel para que el nuevo proceso parezca que está retornando de una llamada al sistema.



## 5. CONTEXTO DEL PROCESO

### a. ESTRUCTURA DEL CONTEXTO DE UN PROCESO

El contexto es la información que permite al sistema operativo pausar y reanudar un proceso sin que este pierda su estado. El contexto del proceso es la unión de:

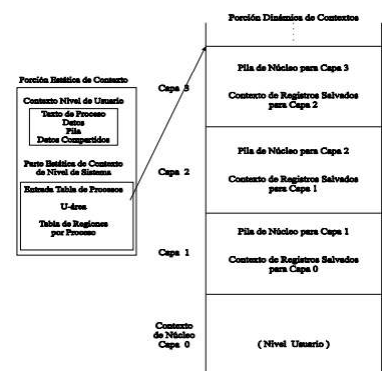
- Contexto a nivel de usuario:
  - Incluye las secciones de texto, datos y pila del proceso.
  - Este es el espacio de direcciones en el que opera el programa.
- Contexto a nivel de registros:
  - PC (Contador de Programa): Indica la próxima instrucción a ejecutar.
  - Registros de propósito general de la CPU, como los registros de datos y de direcciones.
  - SP (Puntero de pila): Apunta al tope de la pila, donde se almacenan las variables locales y las direcciones de retorno.
- Contexto a nivel de sistema
  - Parte estática: Información constante como las tablas de procesos o segmentos de memoria fijos.
  - Parte dinámica: Cambia a medida que el proceso avanza, como la pila de llamadas al sistema o los registros del kernel.

El contexto del proceso es crítico porque permite que un proceso sea suspendido y luego reanudado sin perder el progreso.

### b. CAPAS DE CONTEXTO

Un proceso se ejecuta dentro de su propia "capa de contexto", que se gestiona por el sistema operativo. Cada vez que se produce una interrupción o una llamada al sistema, el sistema añade una nueva capa de contexto:

- Añadir una capa:** Ocurre cuando hay una interrupción o una llamada al sistema, y el sistema guarda el estado actual del proceso.
- Eliminar una capa:** Ocurre cuando el proceso regresa de una interrupción o del modo kernel al modo usuario.



Este concepto es importante porque ilustra cómo el sistema operativo organiza el contexto de un proceso en múltiples niveles, dependiendo de si el proceso está ejecutando código de usuario o de kernel. Un proceso se ejecuta dentro de su capa de contexto actual.

### c. OBSERVACIONES SOBRE EL CONTEXTO DEL PROCESO





Esto no son apuntes pero **tiene un 10 asegurado** (y lo vas a disfrutar igual).

Abre la Cuenta NoCuenta con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Me interesa

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en [ing.es](https://www.ing.es)



Cuando se cambia de un proceso a otro, el nuevo proceso tiene una estructura de **PCB** (Process Control Block) que contiene toda la información necesaria para reanudar su ejecución.

Si el proceso es nuevo, el sistema operativo debe crear un nuevo PCB y ajustar los valores del contexto de usuario para que comience desde la primera instrucción.

El sistema operativo simula que el proceso está volviendo de una llamada al sistema, lo que facilita su integración en la secuencia de ejecución.

#### d. CREACIÓN DE UN PROCESO

La llamada al sistema **CrearProceso()** está diseñada para crear un proceso cuyo PCB tiene la estructura anterior. Este mecanismo se utiliza para generar el PCB del nuevo proceso y establecer el contexto adecuado para que el proceso recién creado comience a ejecutarse. Algunos detalles clave son:

- El SO ajusta los valores del contexto de proceso de usuario para que el proceso recién creado se ejecute desde su primera instrucción.
- Se crea un contexto kernel para que parezca que el proceso retorna de una llamada al sistema, que el proceso está en su ciclo normal de ejecución.

Esta técnica es una manera eficiente de integrar nuevos procesos en el sistema sin necesidad de alterar la lógica del kernel.

### 6. CONSISTENCIAS DE ESTRUCTURAS Y EXCLUSIÓN MUTUA

La consistencia de estructuras es un problema crucial en la gestión de procesos y recursos compartidos: la **exclusión mutua**. Es fundamental que el sistema operativo gestione correctamente los accesos simultáneos a las estructuras del núcleo (kernel) para evitar inconsistencias. Algunos puntos clave son:

- **Unix tradicional** usaba el **enmascaramiento de interrupciones** y no permitía que los procesos que se ejecutaban en modo supervisor fueran interrumpidos.
- **Bloqueo de procesos**: Cuando un proceso está utilizando una estructura del núcleo, otros procesos deben ser bloqueados hasta que el recurso esté disponible. (sleep/wakeup)
- Los sistemas modernos, como **Solaris**, utilizan **spin-locks** para controlar las secciones críticas de código, permitiendo una gestión más eficiente y segura de los recursos compartidos.

Este mecanismo es esencial para mantener la coherencia del sistema y evitar problemas de corrupción de datos.

### 7. TIPOS DE PROCESOS

#### a. TIPOS DE PROCESOS EN SO

→ **Procesos de usuario**: Son aquellos creados por los usuarios para ejecutar aplicaciones.

→ **Procesos demonios** (daemons):

- No están asociados a ningún terminal.
- Realizan funciones del sistema (impresión, administración y control de redes, ...).
- Pueden crearlos el proceso Init o los procesos de usuario.
- Se ejecutan en modo usuario.

→ **Procesos del sistema**:

- Se ejecutan en modo supervisor
- Proporcionan servicios generales del sistema.
- Los crea el proceso 0
- No son tan flexibles como los demonios (recompilación)

La diferenciación entre procesos de usuario y del sistema es importante para entender cómo se gestionan los recursos y cómo se priorizan las tareas dentro del sistema operativo.

#### b. TIPOS DE PROCESOS EN UNIX

Consulta condiciones aquí



do your thing

WUOLAH

En Unix, el núcleo identifica a los procesos por su PID. Los procesos se crean con la llamada al sistema fork (excepto el proceso 0).

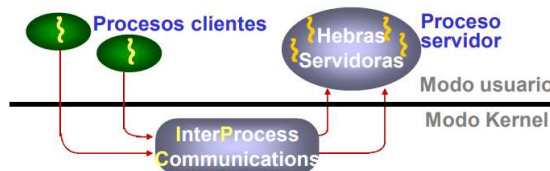
Además, algunos procesos especiales incluyen:

- **Proceso 0**: Es creado de forma manual cuando arranca el sistema. Este proceso es responsable de crear al proceso 1 y luego se convierte en el proceso intercambiador de memoria.
- **Proceso 1 (Init)**: Es el antecesor de todos los procesos en Unix. Todos los procesos del sistema derivan de este proceso.

El **Proceso Init** es fundamental para la estructura de Unix, ya que gestiona el inicio y el control de todos los procesos hijos.

## 8. LIMITACIONES DE PROCESOS

→ **Espacio común de direcciones**: Las aplicaciones que requieren compartir datos, como los servidores, necesitan un espacio de direcciones común para poder operar de manera eficiente. Los procesos tradicionales no permiten compartir espacio de direcciones de manera directa, lo que lleva a la necesidad de mecanismos como la **comunicación entre procesos (IPC)**.



→ **Uso limitado de un único procesador**: Un proceso solo puede ejecutarse en un procesador a la vez, lo que limita el rendimiento en sistemas con múltiples procesadores.

Estos problemas impulsan la necesidad de **hebras (threads)**, que permiten la concurrencia y el uso compartido de recursos dentro del mismo espacio de direcciones.

## 9. HERAS (THREADS)

### a. CONCEPTO DE HEBRA

Las hebras son unidades de ejecución dentro de un proceso. Permiten que un solo proceso contenga múltiples flujos de control, lo que mejora la eficiencia en el uso de recursos.

En un proceso confluyen dos ideas que podemos separar:

- **Flujo de control**: secuencia de instrucciones a ejecutar determinadas por PC, la pila y los registros.
- **Espacio de direcciones**: direcciones de memoria y recursos asignados (archivos, ...)

Permitir más de un flujo de control dentro del mismo espacio de direcciones.

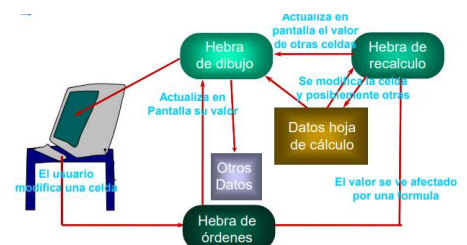
### Hoja de cálculo multihebra

Ejemplo práctico de cómo las hebras se pueden usar en una aplicación real, como una hoja de cálculo multihebrada.

El flujo de trabajo sería el siguiente:

1. **Hebra de órdenes**: Maneja las órdenes del usuario, como cuando este modifica una celda.
2. **Hebra de recálculo**: Actualiza los valores de otras celdas si la modificación afecta a fórmulas o dependencias en otras partes de la hoja.
3. **Hebra de dibujo**: Actualiza la interfaz gráfica de la hoja de cálculo, mostrando los cambios al usuario en la pantalla.

Ilustra cómo un programa puede gestionar varias tareas a la vez sin que una tarea interfiera con la otra.

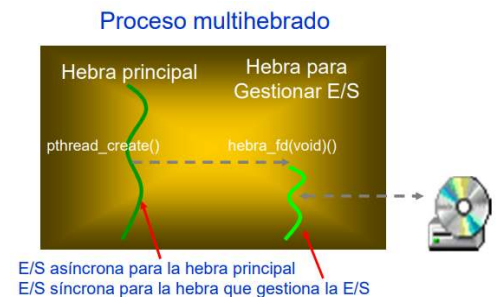


Cada hebra se ocupa de un aspecto diferente del funcionamiento del programa, mejorando la eficiencia y la capacidad de respuesta.

## b. E/S ASÍNCRONA CON HEBRAS

La **entrada/salida asíncrona (E/S asíncrona)** es otro uso de las hebras, como se explica en esta diapositiva. En lugar de bloquear un proceso completo mientras se espera a que se complete una operación de E/S (como leer o escribir en un disco), se pueden utilizar hebras para gestionar la E/S de forma más eficiente:

- **Hebra principal:** Realiza otras tareas mientras una operación de E/S está en curso.
- **Hebra de gestión de E/S:** Se encarga específicamente de las operaciones de E/S, permitiendo que la hebra principal continúe su trabajo sin tener que esperar a que se complete la operación de E/S.



Esto mejora el rendimiento y la eficiencia, especialmente en aplicaciones que dependen de operaciones frecuentes de E/S.

## c. PROGRAMAS MULTIHEBRAS

No todos los programas necesitan o se benefician del uso de hebras. Esta diapositiva identifica las aplicaciones que pueden aprovechar las ventajas de la **multihebración**:

- **Tareas independientes:** Un depurador que necesita una interfaz gráfica y que realiza operaciones asíncronas es un buen candidato para usar hebras, E/S asíncronas,...
- **Programas únicos con operaciones concurrentes:** Ejemplos incluyen servidores de archivos o servidores web, donde varias peticiones de usuarios se pueden manejar simultáneamente.
- **Uso del hardware multiprocesador:** En sistemas con múltiples núcleos o procesadores, las hebras permiten que varias partes de un programa se ejecuten al mismo tiempo en diferentes procesadores.

Sin embargo, también se señala que las hebras **no son útiles** si todas las operaciones son intensivas en el uso de la CPU o si las hebras tienen muy poco trabajo que hacer, debido a la sobrecarga adicional que conllevan.

## d. CÓDIGO REENTRANTE Y SEGURIDAD CON HEBRAS

Aquí se introduce el concepto de **código reentrante**, que es esencial para el uso correcto de hebras. El **código reentrante** es aquel que puede ser ejecutado simultáneamente por 2 o más hebras sin causar errores o corrupción de datos. Se dice también que es **thread-safe**. Para que el código sea reentrante:

- No debe utilizar **datos locales estáticos** dentro del módulo, ya que estos podrían ser modificados por varias hebras al mismo tiempo.
- Debe evitar el uso de **variables globales compartidas** sin mecanismos de sincronización adecuados.

Un sistema operativo debe ser **código reentrante** para manejar hebras de manera segura. Ejemplos:

- **Linux Kernel 2.4** no era 100% reentrante.
- **Linux Kernel 2.6** y versiones posteriores sí lo son.
- **MS-DOS y BIOS** no son reentrantes.

Esto no son apuntes pero **tiene un 10 asegurado** (y lo vas a disfrutar igual).

Abre la Cuenta NoCuenta con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Me interesa

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en [ing.es](#)

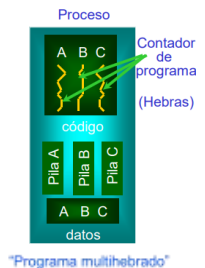


## 10. DISEÑO Y TIPOS DE HEBRAS

### a. DISEÑO DE HEBRAS

Una **hebra** es una unidad de planificación dentro del sistema operativo que tiene su propio **contexto hardware**, lo que incluye:

- **Contador de Programa (PC)**: Marca la siguiente instrucción que la hebra debe ejecutar.
- **Registros de la CPU**: Cada hebra tiene sus propios registros.
- **Pila de Ejecución**: Cada hebra tiene su propia pila para manejar las llamadas a funciones y las variables locales.



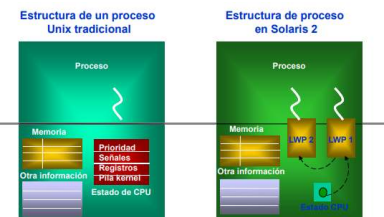
"Programa multihbrero"

Todas las hebras de un mismo proceso comparten el mismo espacio de direcciones, lo que significa que pueden acceder a los mismos datos y archivos, pero se ejecutan de manera independiente.

### b. IMPLEMENTACIÓN DE LWP (LIGHTWEIGHT PROCESSES)

Aquí se describe cómo se implementan los **procesos ligeros (LWP)** en sistemas operativos como Solaris:

- En los sistemas **Unix tradicionales**, un proceso tiene un solo flujo de ejecución (una sola hebra) que utiliza la CPU.
- En **Solaris 2**, cada proceso puede tener múltiples **LWPs**, que permiten ejecutar varias hebras dentro de un mismo proceso. Cada LWP tiene su propio contexto de CPU (como registros, PC, pila de kernel) y puede ser ejecutado de forma independiente por la CPU.



Se compara cómo se estructuran los procesos en un sistema tradicional frente a cómo se manejan en un sistema que admite LWPs.

### c. TIPOS DE HEBRAS 43

- **Hebras Kernel**: implementadas dentro del kernel por el SO. Conmutación entre hebras rápida, pero sigue implicando una llamada al sistema.
- **Hebras de usuario**: implementadas a través de una biblioteca de usuario que actúa como un kernel miniatura. La conmutación entre ellas es muy rápida ya que no involucra al kernel, pero el SO no las conoce, lo que puede ser una desventaja.
- **Enfoques híbridos**: combinan hebras kernel y de usuario, utilizando procesos ligeros (LWP) para equilibrar rendimiento y flexibilidad (p.ej. Solaris 2).

### d. HEBRAS DE USUARIO

**Ventajas:**

- Alto rendimiento porque no consumen recursos del kernel.
- La conmutación entre hebras de usuario es extremadamente rápida.

**Desventajas:**

- El sistema operativo **no conoce** la existencia de estas hebras, lo que conlleva algunos problemas:
  - o Si una hebra se bloquea (por ejemplo, esperando por E/S), todo el proceso se bloquea.
  - o No hay protección entre hebras de usuario, lo que puede llevar a problemas de sincronización.
  - o Problemas de coordinación entre el planificador de la biblioteca y el del SO.

Consulta condiciones aquí



do your thing

WUOLAH



## 11. ESTÁNDARES Y SOPORTE DE HEBRAS EN SO

### a. ESTÁNDARES DE HEBRAS

- **POSIX (Pthreads)**
  - o Es un estándar ISO/IEEE ampliamente soportado en sistemas Unix.
  - o Casi todos los UNIX tienen una biblioteca de hebras.
  - o Define una API común para la creación y gestión de hebras.
- **Win32**
  - o Microsoft ofrece un modelo de hebras propio, diferente a POSIX.
  - o Existen bibliotecas comerciales de POSIX
- **Solaris**
  - o Solaris tenía su propio modelo de hebras antes de que POSIX se convirtiera en estándar.
- **Hebras Java**
  - o Java incluye su propia implementación de hebras, independiente del sistema operativo subyacente.

### b. SOPORTE DE HEBRAS EN SOLARIS 2.x

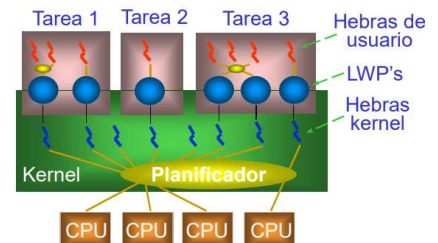
**Solaris 2.x** es una versión multihebrada de Unix que soporta **multiprocesamiento simétrico (SMP)** y **planificación en tiempo real**. Las características clave incluyen:



- **Procesos ligeros (LWP)**: Hebras de usuario soportadas por el kernel. La conmutación entre LWPs es más lenta que la conmutación entre hebras de usuario, ya que involucra llamadas al sistema.
- El kernel solo conoce los **LWPs** que existen en cada proceso de usuario, lo que facilita la planificación y ejecución de hebras.

### c. MODELO DE TAREAS, HEBRAS Y LWPs EN SOLARIS

- **Hebras de usuario**: Ejecutan tareas de usuario sin involucrar al kernel.
- **LWPs (Lightweight Processes)**: Soportados por el kernel para gestionar múltiples hebras de usuario.
- **Hebras de kernel**: Ejecutan tareas del sistema operativo y gestionan las hebras de usuario a través de LWPs.



Este modelo permite que un proceso utilice múltiples CPUs de forma eficiente.

### d. HEBRAS EN LINUX

Linux soporta tanto hebras de kernel como hebras de usuario. Algunos puntos clave son:

- **Hebras de kernel**: Usadas para ejecutar funciones del sistema en modo kernel, como la limpieza de caché de disco o la gestión de conexiones de red.
- **Bibliotecas de hebras de usuario**: Existen varias, aunque no todas las bibliotecas del sistema son reentrantes. Por ejemplo, **glibc v.2** es reentrante, lo que permite un uso seguro de las hebras.

## 12. APIs DE PROCESOS Y HEBRAS

### a. APIs DE UNIX Y WIN32

Compara las APIs que ofrecen **Unix** y **Win32** para la gestión de procesos:

- Unix:

- `fork()`, `exec()`: Para crear procesos.
- `_exit()`: Para terminar procesos.
- `wait()`, `waitpid()`: Para esperar la terminación de un proceso hijo.

- Win32

- `CreateProcess()`: Para crear un nuevo proceso.
- `ExitProcess()`: Para finalizar un proceso.
- `GetExitCodeProcess()`: Para obtener el código de salida de un proceso

Operación	Unix	Win32
Crear	<code>fork()</code> <code>exec()</code>	<code>CreateProcess()</code>
Terminar	<code>_exit()</code>	<code>ExitProcess()</code>
Obtener código finalización	<code>wait</code> <code>waitpid</code>	<code>GetExitCodeProcess</code>
Obtener tiempos	<code>times</code> <code>wait3</code> <code>wait4</code>	<code>GetProcessTimes</code>
Identificador	<code>getpid</code>	<code>GetCurrentProcessId</code>
Terminar otro proceso	<code>kill</code>	<code>TerminateProcess</code>

### b. APIs DE HEBRAS

Compara las APIs de **Pthreads (POSIX)** y **Win32** para la gestión de hebras:

- Pthreads (POSIX):

- `pthread_create()`: Para crear una hebra.
- `pthread_exit()`: Para terminar una hebra.
- `pthread_cancel()`: Para finalizar una hebra de manera controlada.

- Win32

- `CreateThread()`: Para crear una hebra en el sistema Win32.
- `ExitThread()`: Para finalizar una hebra.
- `TerminateThread()`: Para terminar una hebra de manera forzada.

Operación	Pthread	Win32
Crear	<code>Pthread_create</code>	<code>CreateThread</code>
Crear en otro proceso	-	<code>CreateRemoteThread</code>
Terminar	<code>Pthread_exit</code>	<code>ExitThread</code>
Código finalización	<code>Pthread_yield</code>	<code>GetExitCodeThread</code>
Terminar	<code>Pthread_cancel</code>	<code>TerminateThread</code>
Identificador	-	<code>GetCurrentThreadId</code>