Conceptos Impotantes

- **Shell**: **Interfaz visual** que interpreta comandos del **Sistema Operativo** realizando **Lamadas al Sistema**.
- **Proceso**: Es una **instancia de un programa** en ejecución.
- **Multiprogramación**: Ejecución de **varios procesos simultáneamente**.
- **Llamadas al Sistema**: Es un mecanismo usado por una **aplicación** para **solicitar un servicio al Sistema Operativo**. Para ello se usan algunas **librerías de C**.

Procesos

Todas las computadoras modernas ofrecen varias tareas al mismo tiempo, lo cual, aunque no nos paremos a pensarlo, existe gracias a los distintos procesos.

Por ejemplo, cuando se arranca un PC, se inician muchos procesos de forma secreta que a menudo el usuario desconoce. Uno de ellos puede ser el encargado de esperar el correo electónico entrante o para que el antivirus compruebe periódicamente la disponibilidad de nuevas definiciones de virus. Aunque también existen otros procesos explícitos como arrancar una terminal u otro programa.

La conmutación entre dos procesos es muy rápida. Tanto que a veces existe una sóla CPU que trabaja en 1 segundo con decenas de procesos dando la apariencia de un paralelismo (que en realidad es **pseudoparalelismo**). Para obtener paralelismo real debemos trabajar con más de una CPU al mismo tiempo sobre la memoria física (**Multiprocesadores**).

El Modelo del Proceso

Los programas se organizan en varios **procesos secuenciales**. Un proceso no es más que la **instancia de un programa en ejecución** (que incluye sus propios valores de contador, registros y variables). Cada proceso tiene su propia CPU virtual que, en realidad es la misma CPU que conmuta de un proceso a otro. A esta conmutación es a la que se llama **multiprogramación**.

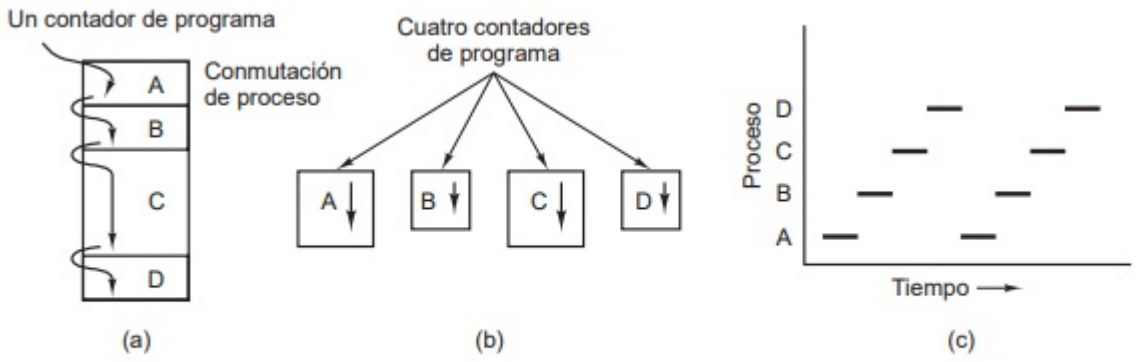

Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo hay un programa activo a la vez.

Como podemos comprobar en la imagen existe **un sólo contador de programa físico**, por lo que cuando

conmuta de un proceso a otro, se carga el **contador lógico** de cada programa que se guarda y carga cada vez que se conmuta de proceso.

Diferencia entre Proceso y Programa

Imaginemos que existe un cocinero (**CPU**) que esta preparando un pastel (**proceso** "preparar pastel"). Como no sabe hornearlo necesita leer los ingredientes den libro (**E/S**). De repente, su hijo se hace una herida y el cocinero va rápidamente a ponerle un vendaje (conmuta a un proceso de mayor prioridad, **proceso** "vendar a su hijo"). La clave es que el programa es en sí la receta pero este programa se puede ejecutar por distintos procesos.

Creación de un Proceso

Hay cuatro eventos principales que provocan la creación de procesos:

1. El **arranque del sistema**.
2. Que un **proceso de una llamada al sistema** para crear otro proceso.
3. Una **petición de usuario** para crear un proceso.
4. El inicio de un **trabajo por lotes**.

A menudo existen procesos que se suelen crear al arranque del sistema llamados **demonios** que permanecen en segundo plano para manejar ciertas **actividades en segundo plano**. Otras veces un proceso necesita la ayuda de otro distinto para realizar su trabajo. O el usuario ejecuta un comando en la terminal o inicia un programa haciendo doble click en el entorno de escritorio.

La última situación se aplica a los Sistemas de Procesamiento por Lotes, donde los usuarios pueden enviar trabajos de procesamiento por lotes. Cuando el sistema operativo decide que tiene los recursos para ejecutar otro trabajo, crea un proceso y ejecuta el siguiente trabajo de la cola de entrada.

La única forma que se tiene de **crear un proceso** es hacer una llamada al sistema y ejecutar un ***fork()***. Con esta llamada **el padre crea un clon exacto** del proceso que hizo la llamada. De hecho, al arrancar un sistema UNIX se crea el primer proceso llamado init que es el encargado de crear el primer proceso hijo, así van haciendo continuamente hijos y ampliando todos los procesos del sistema.

Tanto en UNIX como en Windows, una vez que se crea un proceso, el **padre y el hijo tienen sus propios espacios de direcciones distintos**. Sin embargo, el espacio de direcciones inicial del hijo es una **copia** del padre (pero en definitiva son dos espacios distintos). Aún así, el proceso **hijo también comparte otros recursos de su padre como los archivos abiertos**.

Por lo tanto podemos deducir que hay 3 tipos de procesos:

1. **De usuario**.
2. **De kernel**.
3. **Demonios**.

Terminación de Procesos

Hay cuatro eventos principales que provocan la terminación de procesos:

1. **Salida normal (voluntaria)**: normalmente es cuando un proceso **termina su trabajo**, la llamada para terminarse a sí mismo es ***exit()***. Además los *exit()* están implicitos al finalizar el main
2. **Salida por error (voluntaria)**: si el proceso descubre que **existe un error** (como por ejemplo que a un comando se le pasen argumentos inválidos) puede **terminarse a sí mismo** (o puede decidir que no, es cosa de cómo esté programado).
3. **Error fatal (involuntaria)**: puede ser provocado al dividir entre 0, hacer referencia a una instrucción ilegal o a una parte de memoria inexistente... el **proceso no sabe lo que hacer y se interrumpe**.
4. **Eliminado por otro proceso (involuntaria)**: si un proceso ejecuta un ***kill()* a otro proceso**.

Cuando un proceso termina, **debe notificar al padre para la terminación completa** del mismo. Esta notificación se realiza con ***wait(ret)* que comprueba si tiene algún hijo en estado 'zombie'** (proceso que ya ha terminado pero no ha sido eliminado del sistema). Si no tiene ningún hijo sale devolviendo un error, y **si tiene un hijo o más pero no son 'zombies' invoca *sleep()*** hasta que algún proceso hijo termine. Cuando un proceso hijo termina y se queda en estado 'zombie' el padre es el encargado de, mediante este wait, buscar al hijo y borrar sus contenidos de la tabla de procesos y devolver el **pid** y el **código de retorno** en *ret* para A.

Por lo tanto, podemos concluir que cuando un proceso hijo ejecute el *exit()* se queda en estado 'zombie' y espera a que el padre lo elimine completamente del sistema con el *wait()*

Jerarquías de Procesos

En UNIX, un proceso y todos sus hijos, junto con sus posteriores descendientes, forman un **grupo de procesos**. Cuando un usuario envía una **señal** del teclado, **ésta se envía a todos los miembros del grupo de procesos** actualmente asociado con el teclado, pero de manera individual, cada proceso puede atrapar la señal, ignorarla, etc.

Por otro lado, en Windows, cuando un proceso crea un hijo recibe un **manejador** que puede usar para controlar al hijo. Este manejador se puede pasar a otros procesos para que lo controlen ellos. Es la principal diferencia de por qué en Windows se invalida la jerarquía de procesos.

Señales

Es una **notificación asíncrona** que se manda a un proceso. Cuando se envía una señal, el Sistema Operativo se encarga de entregar la señal.

Los procesos pueden tener un **"Manejador de Señal"**, que realmente **es una rutina** que se ejecuta cuando el proceso recibe la señal que este Manejador tiene asociada. **Si el proceso no tiene ningún Manejador asociado**, se ejecuta el **"Manejador por Defecto"** que tiene cada señal asociado.

```
Por ejemplo, 'CTRL+C' tiene asociado un Manejador por Defecto que termina la
ejecución del proceso. Sin embargo, podemos asociarle un Manejador propio para
que se ejecute esa rutina y, si queremos, no terminal el proceso y realizar
otra operación.
```

Estados de un Proceso

Cuando un proceso se bloquea, lo hace debido a que por lógica no puede continuar, comúnmente porque está esperando a E/S o a la salida de otro proceso que aún no ha terminado. También es posible que un proceso esté listo para ejecutarse pero el sistema haya decidido asignarle a CPU a otro proceso.

- **En ejecución** (está usando la CPU en ese instante)
- **Listo** (ejecutable, el sistema ha decidido asignar la CPU a otro proceso)
- **Bloqueado** (No puede ejecutarse hasta que ocurra cierto evento externo a él)

1. El proceso **se bloquea** debido a que por lógica **no puede continuar**, comúnmente está esperando E/S o la salida de otro proceso que aún no ha terminado.
2. Se produce una **interrupción de reloj** y el **Sistema Operativo selecciona el orden de ejecución de los procesos** mediante algún algoritmo y decide ejecutar otro.
3. De la misma manera, se produce una **interrupción de reloj** y el **Sistema Operativo inicia el proceso**.
4. La entrada que estaba esperando (E/S o la salida de otro proceso) ya la ha obtenido y **vuelve a la cola de espera en "Listo"** para que el planificador de procesos lo selecciona y lo ejecute.

Implementación de los Procesos

Para realizar esta ardua tarea el Sistema Operativo mantiene una **tabla de procesos**. En ella hay únicamente **una entrada por cada proceso**. En cada tupla hay dicho proceso y toda la información necesaria sobre él (**contador del programa, apuntador de pila, asignación de memoria, estado de archivos abiertos, información de contabilidad y planificación...**). Toda esta información es necesaria tenerla guardada para cuando el Sistema Operativo seleccione otro proceso para ejecutar, ya que, cuando tenga que volver a este mismo proceso necesitará recuperar toda la información.

Administración de procesos	Administración de memoria	Administración de archivos
Registros	Apuntador a la información del segmento de texto	Directorio raíz
Contador del programa	Apuntador a la información del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Apuntador a la información del segmento de pila	Descripciones de archivos
Apuntador de la pila		ID de usuario
Estado del proceso		ID de grupo
Prioridad		
Parámetros de planificación		
ID del proceso		
Proceso padre		
Grupo de procesos		
Señales		
Tiempo de inicio del proceso		
Tiempo utilizado de la CPU		
Tiempo de la CPU utilizado por el hijo		
Hora de la siguiente alarma		

Figura 2-4. Algunos de los campos de una entrada típica en la tabla de procesos.

En esta última imagen vemos que en la Tabla de Procesos hay información de la **administración de procesos** pero también de su **administración de memoria** y de su **administración de archivos**.

Los procesos pueden tener varias **causas de interrupciones**:

- **De Reloj**.
- **Llamadas al Sistema**:
- Otras...

A parte de estas dos, vamos a explicar las interrupciones que provocan algunos E/S y como se gestionan de tal forma que aún teniendo una sola CPU de la impresión de que se mantienen varios procesos secuenciales.

Habitualmente mientras algún proceso se ejecuta, suele recibir alguna **interrupcion por parte de E/S**, en cada interrupción se genera un **vector de interrupción** que se almacena generalmente al final de la memoria. Esta ubicación contiene la **dirección del procedimiento del servicio de interrupciones**.

Cómo ocurre alguna Interrupción:

1. El hardware mete el contador del programa actual y la **palabra de estado** (registro que contiene la info sobre el estado de un proceso, normalmente incluye la dirección de la siguiente dirección a ejecutar) en **la pila**.
2. El hardware **salta a la dirección** especificada en el **vector de interrupción** (lo que es lo mismo que cargar el contador de la interrupción).
3. El software de interrupción (en assembly) **guarda** todos los registros en la **tabla de procesos**.
4. El SO (en assembly) se **deshace de la información que la interrupción metió en la pila** y establece una nueva pila temporal.
5. Se realiza en un procedimiento de C el **resto del trabajo** para esta interrupción específica.
6. El planificador decide qué **proceso se debe ejecutar a continuación**.
7. El SO (en assembly) **carga los registros y el mapa de memoria** y empieza a **ejecutar** el proceso que ahora es el actual.

Hilos

La principal razón de tener hilos es que en muchas aplicaciones se desarrollan varias actividades a la vez. Al descomponer una aplicación en varios hilos secuenciales que se ejecutan en cuasi-paralelo, el modelo de programación se simplifica.

Principales ventajas o razones de por qué los hilos trabajan mejor en paralelo que los procesos por lo general:

1. Los hilos **comparten espacio de direcciones y todos sus datos**.
2. Los hilos son **más ligeros** que los procesos. Son más fáciles de crear y destuir y conmutar.
3. Cuando hay una gran cantidad de **cálculos y operaciones E/S**, los hilos pueden **solapar** estas actividades que colaboran entre sí y agilizar la aplicación.

Ahora supongamos un **procesador de texto de 3 hilos**. Si el programa tuviera sólo un hilo, entonces cada bez que se iniciara un respaldo en el disco se ignorarían los comandos del teclado y ratón hasta que terminara el respaldo. De la misma manera que los eventos de teclado podrían interrumpir el respaldo de disco.

Al tener 3 hilos operando en el mismo documento a la vez, compartiendo memoria, etc. se evitarían estos fallos.

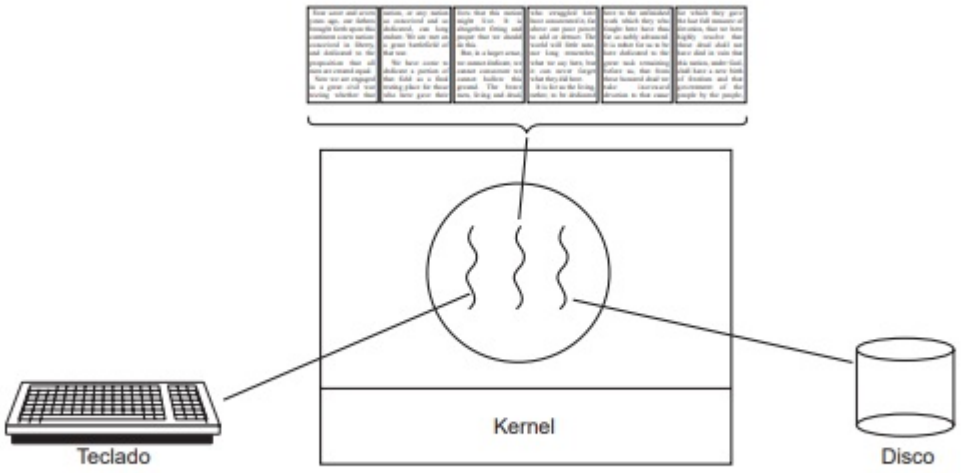

Figura 2-7. Un procesador de palabras con tres hilos.

Por otro lado, se pueden tomar otros ejemplos un poco más complejos pero quizás más educativos.

Consideremos la página de inicio de Apple, obviamente esta página tiene muchísimas más visitas que una página en varios niveles de profundidad explicando las características de un timo como unos cascos inhalámbricos. Los servidores mantienen una colección de páginas de uso frecuente en la memoria principal (a esta colección se le conoce como **caché**) como la página principal de Apple y la de los nuevos dispositivos lanzados el último año.

Primeramente existe un **hilo despachador** que lee las peticiones entrantes. Después de examinar cada solicitud, la envía a un **hilo trabajador** inactivo enviándole una solicitud para despertarlo. El trabajador comprueba si la página pedida se encuentra en caché a la que tienen acceso inmediato todos los hilos. De no ser así, inicia una operación *read* para obtener la página del disco (mientras espera la página del disco se bloquea para dar más trabajo a otros hilos despachadores o trabajadores).

```
while (TRUE)                    while(TRUE)
{                               {
    obtener_petición();             esperar_trabajo();
    pasar_trabajo();                buscar_pagina_cache();
}                                   if (pagina_no_esta_en_cache())
                                    {
                                        leer_pagina_disco
                                    }
                                    devolver_pagina();
                                }
```

Como podemos ver en este último ejemplo, sin hilos no se podrían atener a otros usuarios webs mientras accede a disco.

Modelo	Características
Hilos	Cuasi-paralelismo, llamadas al sistema con bloqueo

Procesos con 1 sólo hilo	Sin paralelismo, llamadas al sistema con bloqueo
Máquina estados finitos	Paralelismo, interrupciones

Modelo Clásico de Hilo

Una manera de ver a un proceso es como si fuera una forma de agrupar recursos relacionados. El otro concepto que tiene un proceso es un hilo. Aunque el hilo se debe ejecutar en cierto proceso, el hilo y proceso son conceptos distintos y pueden tratarse por separado.

Los hilos proporcionan al modelo de procesos el permitir que se lleven a cabo varias ejecuciones en el mismo entorno del proceso. Cuando un proceso comparte varios hilos se llama **multihilamiento** y permiten que las conmutaciones de hilos ocurran en una escala de nanosegundos.

Cuando se ejecuta un proceso con **multihilamiento** en un sistema con una CPU, los hilos toman turnos para ejecutarse, la CPU conmuta rápidamente entre un hilo y otro dando la ilusión de que los hilos se ejecutan en paralelo.

Todos los hilos **comparten el mismo espacio de direcciones**, por lo que los hilos **comparten las variables globales y pueden acceder a la pila de otro hilo** (no hay protección ya que es imposible e innecesario). Además todos los hilos comparten archivos abiertos, procesos hijos, alarmas señales, etc.

Elementos por proceso	Elementos por hilo
Espacio de direcciones	Contador de programa
Variables globales	Registros
Archivos abiertos	Pila
Procesos hijos	Estado
Alarmas pendientes	
Señales y manejadores de señales	
Información contable	

Figura 2-12. La primera columna lista algunos elementos compartidos por todos los hilos en un proceso; la segunda, algunos elementos que son privados para cada hilo.

Los elementos de la primera columna son propiedades de un proceso, no de un hilo, pero que lo comparten todos los hilos. Sin embargo, los de la segunda columna es importante que sean privados del hilo:

Un hilo puede estar en **varios estados**: *ejecución*, *bloqueado* o *listo* (como pudimos comprobar en el ejemplo de la Web). Es importante que **cada hilo tenga su propia pila** (y su propio registro que indica la posición de la pila actual), que contiene un conjunto de valores para cada procedimiento llamado. Este conjunto de valores contiene las variables locales del procedimiento y la dirección de retorno para volver al hilo (Cada hilo llama a distintos procedimientos y por ende, tiene un historial de ejecución diferente y neceita su propia pila).

Creación de Hilos

Por lo general, los procesos comienzan con un sólo hilo presenta, que tiene la habilidad de crear otros hilos mediante la llamada a un procedimiento de biblioteca <pthread.h> como *pthread_create()*.

```
pthread_create (pthread_t *thread, pthread_attr *attr, void *(start_routine)
(*void), void *arg)

thread: Es el lugar donde el ID del nuevo hilo creado va a ser guardado, NULL
si el ID no es requerido.
attr:   Es el atributo especificado, NULL si el hilo es creado sin atributos
start:  Es le función principal del hilo... el hilo empieza a ejecutarse en
esta dirección
arg:    Es el argumento que se le pasa a la función 'start' (si se quieren
pasar varios argumentos es necesario pasarle una estructura con todos ellos)
```

Cuando un **hilo termina su trabajo** puede salir inmediatamente con *pthread_exit()*. En algunos casos el hilo 'padre' (por llamarlo análogamente a los procesos) puede ejecutar *pthread_join()* para bloquear al hilo llamador hasta que un hilo específico haya terminado.

Otra opción interesante que proporcionan los hilos es *pthread_yield()* que **permite a un hilo entregar voluntariamente la CPU**. En este caso no hay una interrupción de reloj (de tal forma que **si los hilos son 'amables' y entregan periódicamente la CPU no necesitan esperar a una interrupción** y así ser mucho más óptimos que los procesos aún).

```
#define NUMBER_OF_THREADS   10

void *print (void *tid)
{
    printf ("Hola! te saludo, soy hilo %d", tid);
    pthread_exit(NULL);
}

int main ()
{
    pthread_t   thread[NUMBER_OF_THREADS];
    int         status;
    int         i;

    for (i = 0; i < NUMBER_OF_THREADS, i++ )
    {
        printf("Soy el hilo main, vamos a crear el hilo número %d", i);
        status = pthread_create (&thread[i], NULL, print, (void*) i);

        if (status != 0)
        {
            printf ("error code %d", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

```
NOTA, REPASAR TAMBIÉN POR LA PRÁCTICA DE HILOS DE CLASE PARA ENTENDER MEJOR SU
COMPATICIÓN
```

Hilos en Espacio Usuario

En este espacio de usuario **el kernel no sabe nada acerca de ellos**. En lo que al kernel concierne, **está administrando procesos ordinarios como si fuera con un sólo hilo**. Con esta categoría los hilos se implementan mediante una biblioteca.

Esta estructura se basa en que los hilos se ejecutan encima de un sitema en tiempo de ejecución, el cual es una colección de procedimientos que administran hilos (procedimientos como *pthread_create pthread_exit*...)

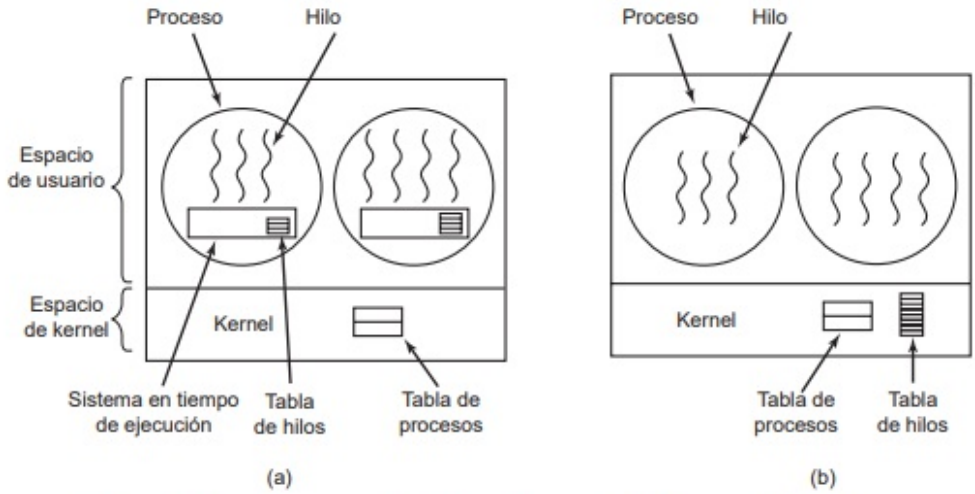

Figura 2-16. (a) Un paquete de hilos de nivel usuario. (b) Un paquete de hilos administrado por el kernel.

```
En informática, un Sistema en Tiempo de Ejecución es un software que provee
los servicios para un programa en ejecución pero no es considerado en sí mismo
como parte del SO.
```

Cuando los hilos se administran en espacio de usuario, **cada proceso necesita su propia tabla de hilos** para llevar cuenta de los hilos en este proceso ya que el kernel administra los procesos como si tuvieran un sólo hilo. Esta tabla es **similar a la tabla de procesos del kernel** salvo que la tabla de hilos únicamente lleva cuenta del contador del programa del hilo, el apuntador a la pila, registros, estado del hilo... en definitiva la información de cada hilo. Además, cuando se bloquea y se desbloquea el hilo, la información se guarda toda allí para desbloquearlo. Esta tabla es administrada por el **Sistema en Tiempo de Ejecución**.

Conmutar Hilos Nivel Usuario

Cuando un hilo hace algo que puede ponerlo en estado bloqueado en forma localL, (por ejemplo esperar a que otro hilo dentro de su proceso complete cierto trabajo) llama a un procedimiento del **Sistema en Tiempo de Ejecución**.

- Este procedimiento **comprueba** si el hilo realmente **necesita ponerse en bloqueado**.

- Si lo necesita bloquear, el propio hilo **almacena sus propios registros en la tabla de hilos**.
- **Busca** en la tabla **un hilo en estado 'listo'** y carga sus registros.

Una vez el procedimiento del sistema haga esas tareas, simplemente toca comnutar el apuntador de pila y el contador del programa.

Como se puede ver, realizar la **conmutación de hilos es mucho más veloz que haceer un trap al kernel** como en el caso de los **hilos nivel kernel**.

Otra **ventaja** que tienen los **hilos nivel usuario** es que permiten que **cada proceso tenga su propio algoritmo de planificación personalizado**. Y además, si queremos aumentar el número de hilos en ejecución, si lo hacemos a nivel kernel va a ser más costoso para el almacenamiento.

PROBLEMAS DE HILOS A NIVEL USUARIO

- Las **Llamadas al Sistema con bloqueo** pueden **parar a todos los hilos del proceso** (por eso antes hemos dicho que el procedimiento comprueba si el hilo necesariamente debe ponerse en bloqueado).
- Los fallos de página de un hilo también bloquean a todos los hilos.

Hilos en Espacio de Kernel

Como podemos observar en la última imagen, en este caso **no se necesita ningún Sistema en Tiempo de Ejecución**. Ya que el **núcleo administra los hilos**: no hay tabla de hilos en cada proceso, sino una **tabla de hilos global**. Cuando un hilo desea cerar un nuevo hilo o destruir uno existente, realiza una llamada al kernel.

En la tabla de hilos global **se almacena la misma información que con los hilos a nivel usuario**, sólo cambia que en vez de hacerlo dentro del espacio de usuario (en concreto dentro del sistema en tiempo de ejecución), **lo hace en espacio de kernel**. Además, no nos olvidemos que **a nivel kernel se está guardando también la tabla de procesos**.

En este caso, cuando un hilo se bloquea, el **kernel puede ejecutar cualquier otro hilo del sistema**, que es **más lento** ya que las llamadas al sistema son **más costosas** que el método que explicamos antes. Pero **la ventaja**, es que si un hilo en un proceso produce un **fallo de página**, el **kernel puede comprobar con facilidad** si el proceso tiene otros hilos que puedan ejecutarse y no perder tiempo a que traiga página de disco.

Implementaciones Hibridas

Una forma de combinar las dos ventajas es utilizar **hilos kernel y multiplexar uno o más hilos kernel en uno o más hilos usuario** (a gusto del programador).

El kernel está únicamente consciente sólo de los hilos kernel y los planifica. Algunos de esos hilos pueden tener varios hilos de nivel usuario multiplexados. Los de nivel usuario se crean, destruyen y planifican de igual forma que los de nivel usuario.

Hilos Emergentes

Los hilos se utilizan con frecuencia en los **sistemas distribuidos**:

La llegada de un mensaje al Sistema hace que éste cree un nuevo hilo para manejar el mensaje. Dicho hilo se conoce como **hilo emergente**. Una ventaja clave es que, como son nuevos, no tienen historial que sea necesario restaurar y no es muy costoso de crear.

Cada uno empieza desde cero y es idéntico a los demás.

La **ventaja de utilizar hilos emergentes** es que la **latencia** entre la llegada del mensaje y el inicio del procesamiento suele ser **muy baja**. Hacer que el **hilo emergente se ejecute en espacio de kernel es por lo general más rápido y sencillo** que colocarlo en espacio usuario en el caso de hilos emergentes. Además, los hilos en espacio kernel pueden acceder fácilmente a E/S que pueden ser necesarios para el procesamiento de interrupciones.

Utilización de CPU

- Estrategia: si un proceso limitado a E/S requiere ejecutarse, debe obtener rápidamente la CPU
- Objetivo: desperdiciar poco tiempo de la CPU (por ejemplo, menos del 10%)

```
Si los procesos usan el 80% de su tiempo esperando en operaciones E/S, debe
haber 10 procesos en memoria.
Si usan el 20%, es suficiente con 2 procesos en memoria.
```

Con esto deducimos que, obviamente, si tenemos muchos procesos esperando mucho tiempo sin estar trabajando, realmente no estamos haciendo nada útil (más que esperar). Por lo que para obtener mayor grado de multiprogramación (más rendimiento) necesitamos tener más procesos activos a medida que los procesos que tenemos necesitan esperar más tiempo sin hacer nada.