

Hilos (threads)



Realizado por M. Curiel

Definiciones

Un **proceso** es una entidad que posee 2 características importantes:

- **Recursos**: un espacio de direcciones (programas, datos, pila y un PCB), archivos, memoria, etc. El SOP realiza la función de protección para evitar interferencias no deseadas entre procesos en relación con los recursos.
- **Planificación/Ejecución**: El proceso sigue una ruta de ejecución. Tiene un PC, un estado de ejecución (Listo, bloqueado, ejecutándose, etc.) y una prioridad.

Definiciones

- Estas dos características son independientes y pueden ser tratadas como tales por los sistemas operativos.
- En algunos sistemas operativos se le denomina hilo (thread) a la unidad activa y a la unidad propietaria de recursos se le suele denominar proceso o tarea.

Definiciones

En un entorno multihilo **se le asocia a los procesos:**

- Un espacio de direcciones virtuales que soporta la imagen del proceso.
- Acceso protegido a procesadores, a otros procesos, archivos y recursos de E/S

Definiciones

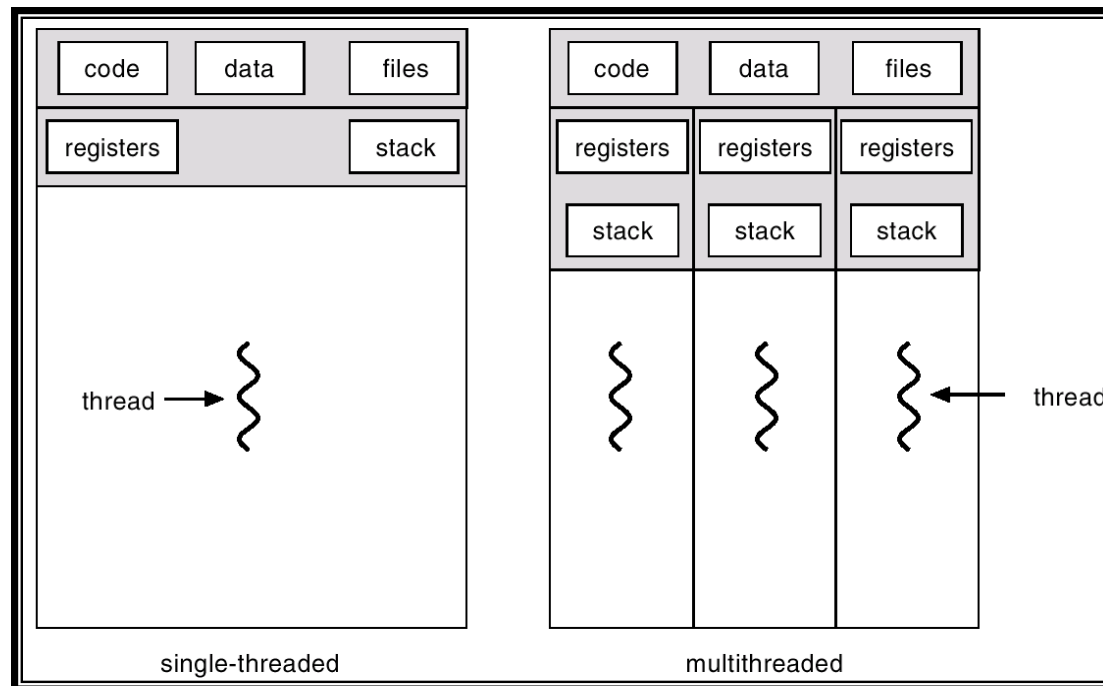
A cada hilo se le asocian :

- Un estado de ejecución.
- Un PC, un contexto (conjunto de registros) que se almacena cuando no está en ejecución.
- Una pila.
- Un espacio de almacenamiento para variables locales.
- Acceso a la memoria y recursos del proceso.

Características de los hilos

- No existe forma de predecir el orden de ejecución o el orden de culminación de varios hilos.
- Los hilos son invisibles fuera de los límites del programa o proceso que los contiene.
- Los hilos tienen su propia identidad, i.e., pila, PC, registros, prioridad.

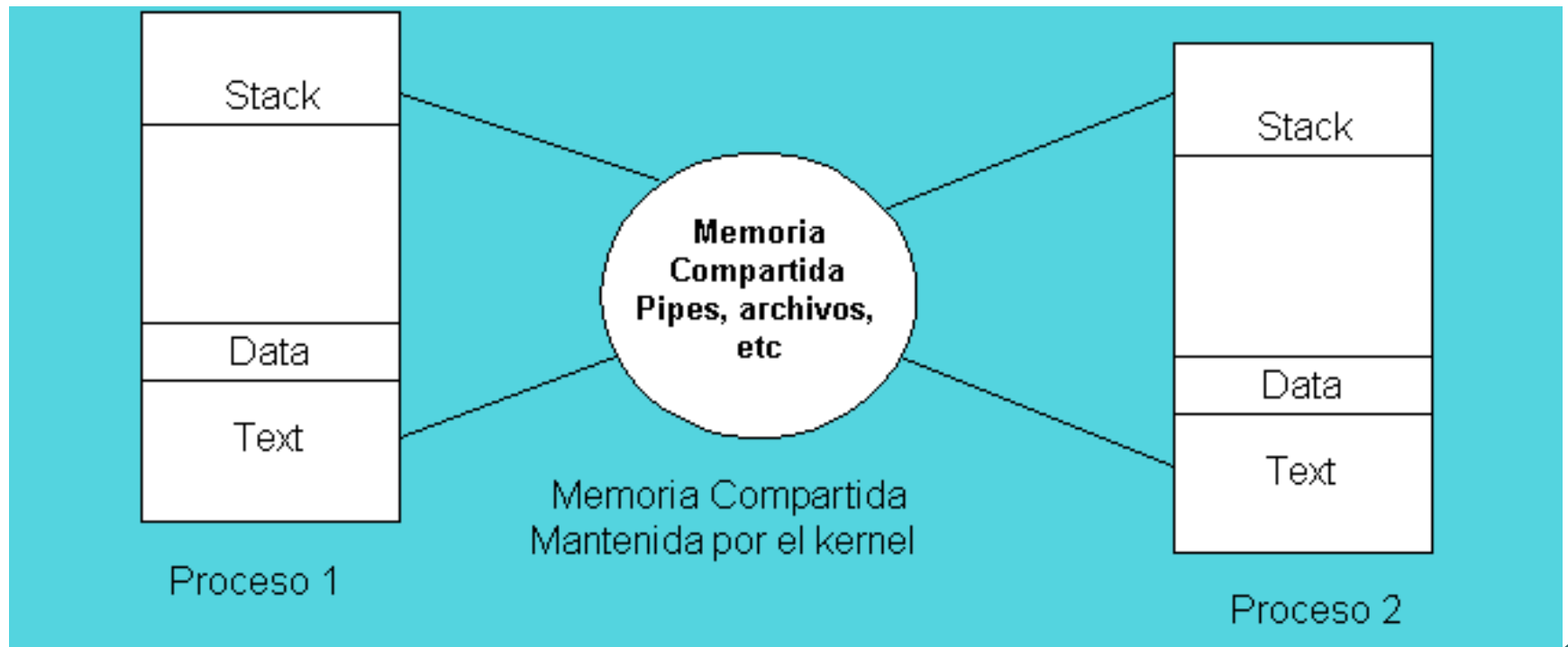
- Un programa **multihilos** es un proceso que contiene más de un hilo. Cada hilo dentro del programa es como un mini-programa con exactamente las mismas propiedades que se mencionaron previamente.



Comunicación

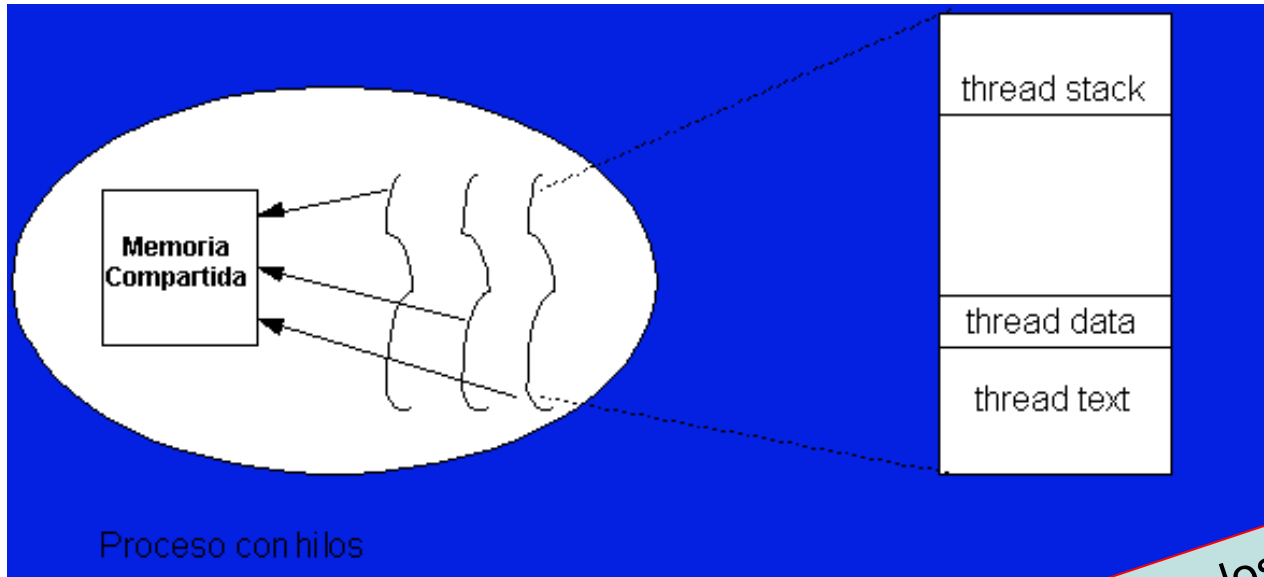
- Por pertenecer a diferentes espacios de direcciones, **la comunicación entre procesos** sólo es posible utilizando **llamadas al sistema.**

Mecanismos de Comunicación entre procesos



Comunicación entre Hilos

- Cada hilo dentro de un proceso es una entidad independiente, pero dado que los hilos forman parte de un mismo proceso o espacio de direcciones, **la comunicación** entre hilos **es mucho más rápida y simple.**



Múltiples hilos tienen acceso directo a los datos del proceso al que pertenecen.

Comunicación entre hilos

- La comunicación entre varios hilos **se lleva a cabo a través de recursos del proceso** usuario, no a través de los recursos del kernel.

Cambios de Contexto

- Los hilos comparten el mismo espacio de direcciones. Por lo tanto el “cambio de contexto” entre un hilo y otro, pertenecientes al mismo proceso, pudiera hacerse de forma totalmente independiente del sistema operativo (hilos a nivel de usuario)

Cambios de Contexto

- Los **cambios de contexto** de los **procesos** son más **costosos**. Implican salvar el contexto del proceso, cambio de modo (trap al sistema operativos, etc) y otro cambio de modo para restaurar contexto del nuevo proceso.

Otras características de los hilos

- Cada hilo es independiente de otro hilo.
- No obstante, ciertas acciones que un hilo ejecuta afectarán a otros hilos dentro del mismo proceso. Por ejemplo:
 - Si un hilo abre un archivo, el archivo estará abierto para el resto de los threads dentro del mismo proceso.
 - Si un hilo termina usando la llamada al sistema *exit()*, esto provocará que todos los threads dentro del mismo proceso terminen.

Por qué usar hilos?

Los mayores beneficios de los hilos provienen de las consecuencias del rendimiento:

- Lleva **mucho menos tiempo crear un hilo** en un proceso existente, que crear un proceso totalmente nuevo. La creación de un hilo puede ser hasta 10 veces más rápida que la creación de un proceso en Unix.
- Lleva **menos tiempo finalizar un hilo** que un proceso.

Por qué usar hilos?

- Lleva menos tiempo el cambio de hilo que el cambio de proceso.
- Debido a que los threads son parte del mismo proceso, el **compartir datos entre ellos es muy fácil**; todo los hilos tienen acceso a los datos globales. No se necesitan llamadas al sistema para su comunicación.

Por qué usar hilos?

- Mayor eficiencia si se tiene un **multiprocesador** (y se soportan **hilos a nivel del kernel**). **Paralelismo**.
- **Mayor concurrencia** aún si se desconocen los hilos a nivel del kernel (si se logra, a través de la librería, que al bloquearse un hilo, pueda ejecutarse otro dentro del mismo proceso).

Por qué usar hilos?

- Los programas que realizan diversas tareas o que tienen varias fuentes y destinos de E/S (servidor) se pueden diseñar e implementar fácilmente usando hilos.

Modelos

- Hilos a nivel de usuario.
- Hilos a nivel de kernel
- Procesos Livianos

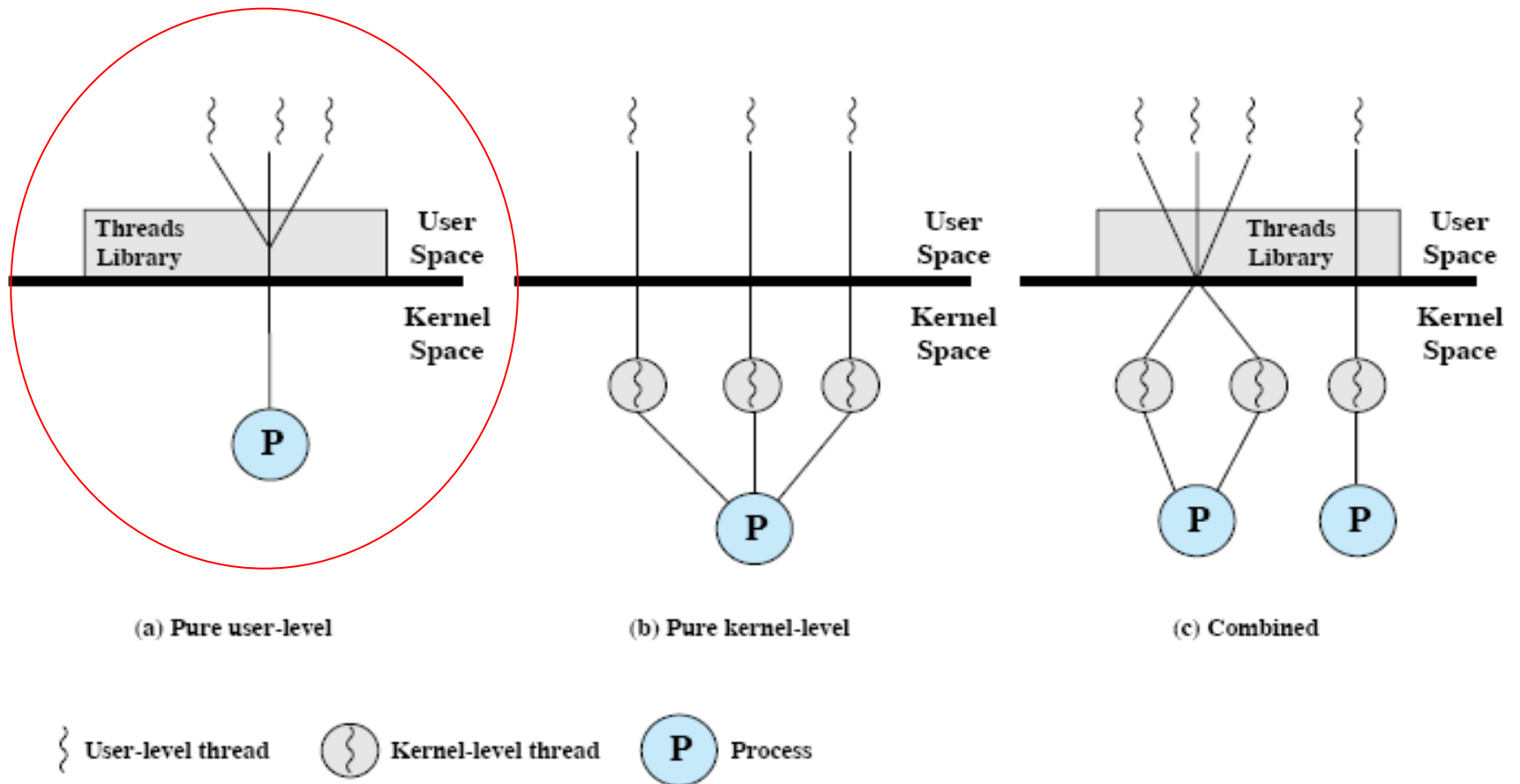
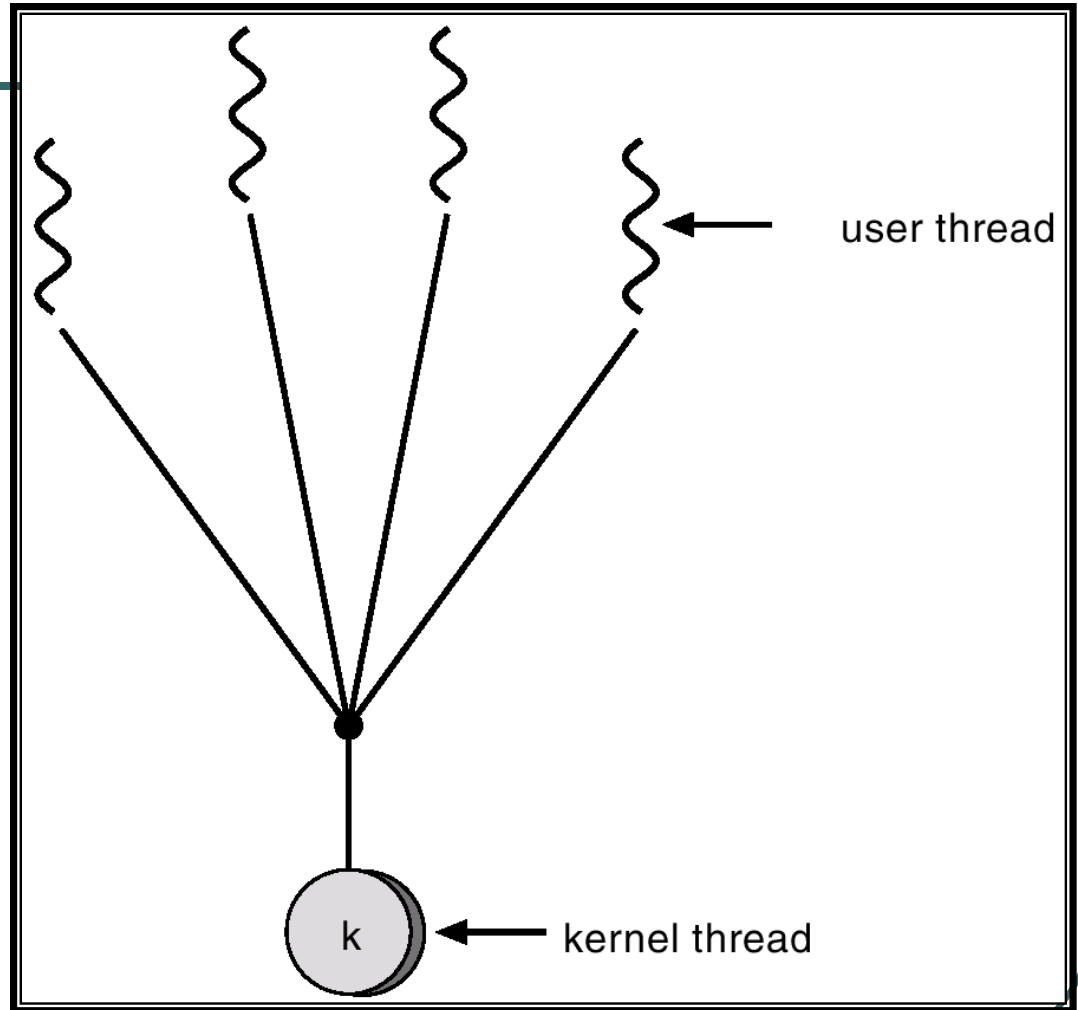


Figure 4.6 User-Level and Kernel-Level Threads

Implementaciones

Muchos-a-Uno

- Varios hilos a nivel de usuario son asignados a un mismo hilo a nivel del kernel
- Este modelo se utiliza en sistemas que no soportan hilos a nivel del kernel
- Ejemplos:
Solaris Green Threads
GNU Portable Threads



Hilos a Nivel de Usuario (ULT)

- El kernel probablemente desconozca la existencia de los hilos.
- Este tipo de hilos es soportado por librerías (ej. C-threads de Mach), las cuales ofrecen las funciones para la creación, sincronización y planificación (scheduling) de hilos.

Hilos a Nivel de Usuario (ULT)

- Como no se involucra al kernel, las operaciones sobre los hilos son más rápidas. La librería actúa como un mini-kernel que controla la ejecución de los hilos.
- Las operaciones de creación, destrucción, sincronización, etc., son **llamadas a rutinas de librería**. Cuando un hilo se suspende, la librería toma el control, salva su contexto y permite la ejecución de otro hilo dentro del mismo proceso.

Hilos a Nivel de Usuario

- Cualquier aplicación puede programarse para ser multihilos a través del uso de una librería de hilos

Ventajas de los Hilos a nivel de Usuario

- El cambio de hilo se hace a nivel de proceso usuario, no hay un cambio de modo. Esto ahorra el *overhead* que producen los dos cambios de modo.
- Se pueden diseñar (si la librería lo permite) estrategias de planificación adecuadas a la aplicación.
- Se pueden ejecutar en cualquier sistema operativo.

Desventajas

- Si un hilo hace una llamada al sistema bloqueante, se bloquearán todos los hilos del proceso a menos que se use la técnica del revestimiento.
- No se obtienen ventajas con el multiprocesamiento.

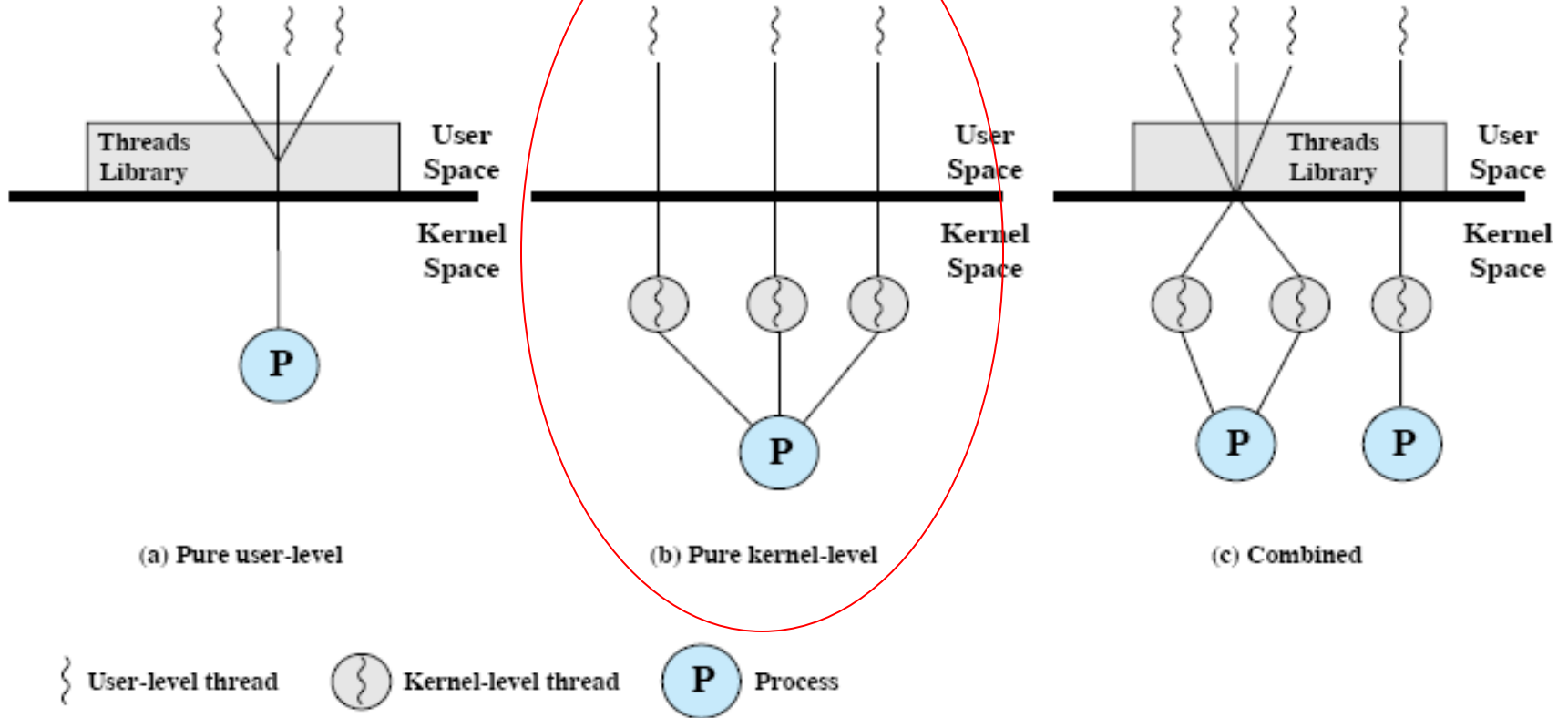


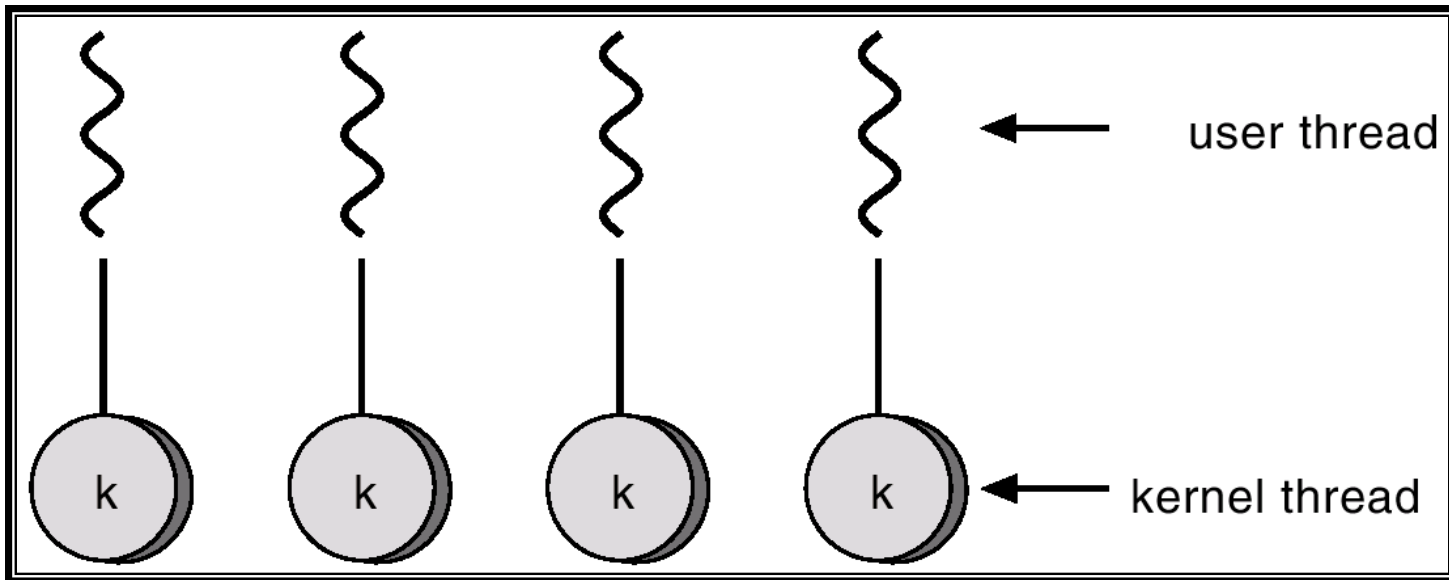
Figure 4.6 User-Level and Kernel-Level Threads

Hilos a Nivel de Kernel

Uno-a-uno

Hilos a Nivel del Kernel

- Cada hilo a nivel de usuario está asociado a un hilo del kernel
- Ejemplos
 - Windows 95/98/NT/2000
 - Linux



Hilos a Nivel del Kernel (KLT)

- El núcleo gestiona los hilos. No hay código de gestión de hilos en la aplicación, sólo la interfaz (API) para acceder a las utilidades de los KLT.
- Cualquier aplicación puede programarse para ser multihilo. Todos los hilos de una aplicación se mantienen en un solo proceso.
- El núcleo mantiene toda la información de cada hilo y realiza la planificación.

Ventajas

- Varios hilos de un mismo proceso se pueden planificar en distintos procesadores.
- Si se bloquea un hilo de un proceso, el SOP puede planificar otro hilo del mismo proceso.
- Las rutinas del núcleo pueden ser en si misma multihilos (demonios).

Desventajas

La siguiente tabla muestra el costo relativo de los hilos a nivel de usuario y a nivel de kernel, como se presenta en el Sun Solaris 2.3 Answer Book.

Operación	Microsegundos
Crear thread no vinculado	52
Crear thread vinculado	350
Fork(0	1700
Sincronizar thread no vinculado	66
Sincronizar thread vinculado	390
Sincronización entre procesos	200

Desventajas

- Si varios hilos acceden el mismo conjunto de datos, este acceso debe ser sincronizado (semáforos, variables de condición, etc.). Tanto las llamadas de sincronización como las de creación y destrucción de KLT requieren de llamadas al sistema.

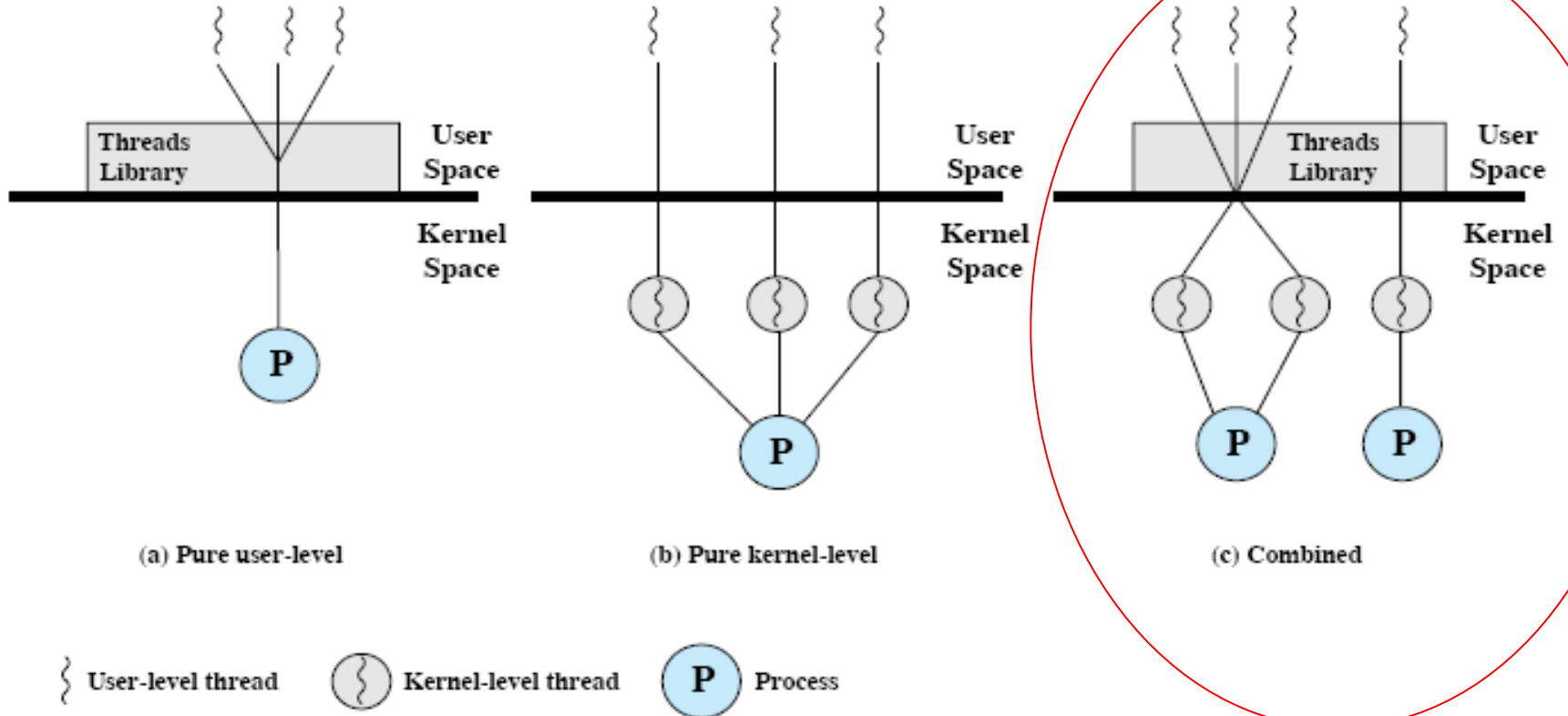


Figure 4.6 User-Level and Kernel-Level Threads

Enfoque combinado: Procesos Livianos

Enfoque combinado: Procesos Livianos

- **El kernel sólo reconoce y gestiona a los procesos livianos.**
- La librería a nivel de usuario multiplexa los hilos del usuario en un determinado número de procesos livianos y se encarga de la **planificación, el cambio de contexto y la sincronización** entre hilos sin involucrar al kernel.

Enfoque Combinado

- La librería seleccionará un hilo a nivel de usuario para que se ejecute en un proceso liviano.
- La librería también garantiza la ejecución de hilos que no estén bloqueados. Si un hilo a nivel de usuario que se está ejecutando llega a bloquearse, la librería elige a otro hilo para su ejecución.

Enfoque Combinado

- Un hilo de usuario usará exactamente un proceso liviano (a nivel del kernel).
- Cuando el hilo usuario concluye, el proceso liviano **se preserva** y puede ser utilizado por otro hilo usuario del mismo proceso o de otro proceso usuario.

Enfoque combinado

- Este concepto ha sido virtualmente abandonado, se habla básicamente de hilos a nivel del kernel o de hilos a nivel de usuario.

POSIX

- POSIX es el acrónimo de **Portable Operating System Interface**; la X viene de UNIX como signo de identidad del API. El término fue sugerido por Richard Stallman en respuesta a la demanda de la IEEE, que buscaba un nombre fácil de recordar.
- Una traducción aproximada del acrónimo podría ser "Interfaz para Sistemas Operativos migrables basados en UNIX".

POSIX

- Se trata de una familia de **estándares** de llamadas al sistema operativo definidos por el **IEEE** y especificados formalmente en el IEEE 1003.
- Persiguen generalizar las interfaces de los sistemas operativos para que una misma aplicación pueda ejecutarse en distintas plataformas.

POSIX

- Estos estándares surgieron de un proyecto de normalización de las API y describen un conjunto de interfaces de aplicación adaptables a una gran variedad de implementaciones de sistemas operativos.
 - POSIX.1, Core Services (implementa las llamadas del ANSI C estándar). I
 - POSIX.1b, extensiones para tiempo real
 - **POSIX.1c, extensiones para hilos (*threads*)**

Threads POSIX

- Las librerías de threads por lo regular incluyen llamadas para crear, destruir y sincronizar threads.
- La mayor parte de las funciones de librería de hilos devuelven 0 si culminaron con éxito y un código de error distinto de 0 si no fue así.

Threads POSIX

- Algunas de las funciones más importantes que ofrece la librería-a POSIX son:

Pthread_create: crea un thread para ejecutar una función específica.

Pthread_exit: hace que el thread invocador termine sin necesidad de que todo el proceso llegue a su fin.

Threads POSIX

- **Pthread_kill**: permite enviar una señal a un thread específico.
- **Pthread_join**: hace que el thread invocador espere hasta que termine un thread específico. Es similar a waitpid en el nivel de procesos.
- **Pthread_self**: devuelve la identidad del invocador.

Threads POSIX

- Un thread tiene un identificador, una pila, una prioridad de ejecución y una dirección de inicio de la ejecución.
- Los threads POSIX tienen un identificador del tipo **pthread_t**.
- Los threads de un proceso comparten su espacio de direcciones, pueden modificar variables globales, acceder a los descriptores de archivos abiertos o interferirse mutuamente de otras formas.

Pthread_create

Crea un hilo en forma dinámica y lo coloca en la lista de hilos listos para ejecución.

```
#include<pthread.h>
int pthread_create(pthread_t *tid, const
pthread_attr_t *attr, void *(*start_routine)(void
*), void *arg)
```

tid, apuntará al identificador del thread que se crea.

Los atributos del thread se almacenan en el objeto atributo al que apunta **attr**. Si **attr** es NULL, el nuevo thread tendrá los atributos por defecto. El tercer parámetro es el nombre de la rutina que el thread invoca cuando inicia su ejecución.

Start_routine requiere de un solo parámetro que se especifica con **arg**, un apuntador a void.

Pthread_exit

```
void pthread_exit(void *value_ptr)
```

Termina al thread que la invoca. El valor de **value_ptr** queda disponible para **pthread_join** si ésta tuvo éxito.

El valor de value_ptr en pthread_exit debe apuntar a datos que existan después que el thread ha terminado, así que no pueden ser variables automáticas del thread que está terminando.

Bibliografía

- **Graham Glass.** *UNIX for Programmers and Users. A complete guide.* Prentice Hall
- **K. Robbins and Steven Robbins .** *UNIX: Programación Práctica. Guía para la Concurrencia, la comunicación y los Multihilos.* Prentice Hall.
- **Stallings, William.** *Operating Systems, Internals and Design Principles 5/e.* Prentice Hall 2005. ISBN 0-13-147954-7