Sistemas Operativos I

Threads

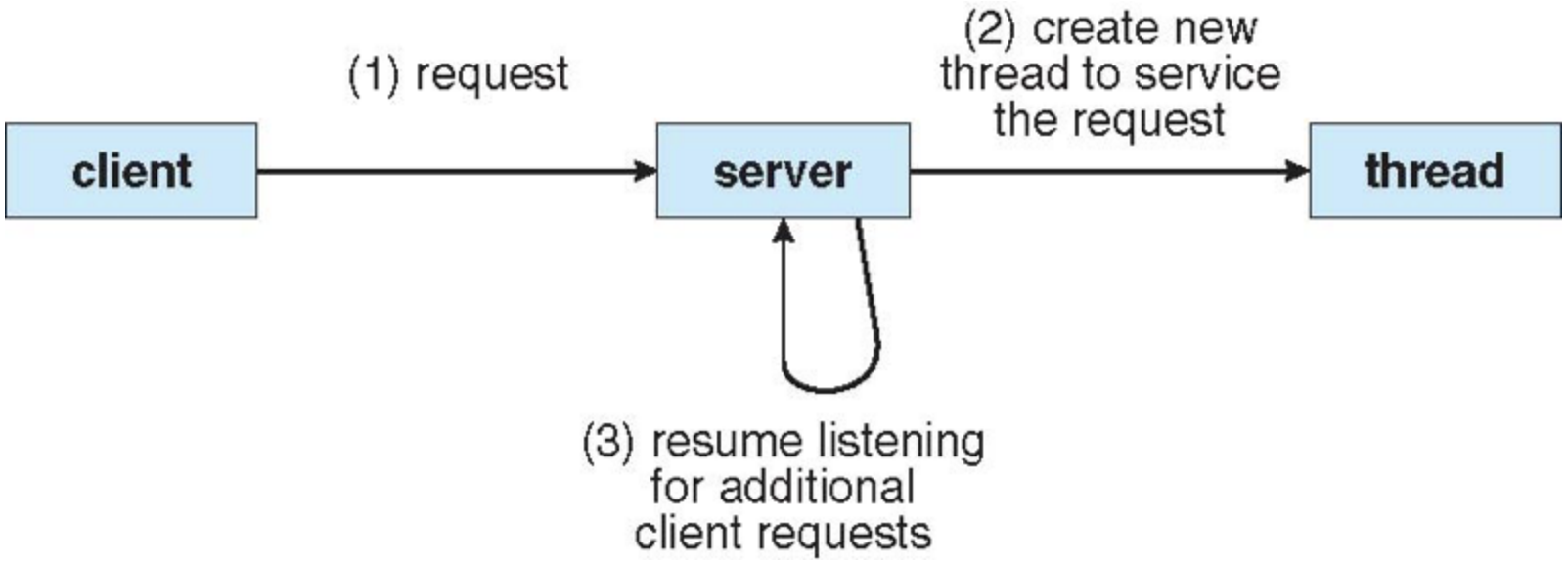
Conceptos

- Hasta ahora asumimos que un proceso era un programa en ejecución con un único thread de control
- Un thread es una unidad básica de utilización/asignación de CPU; contiene:
  - ID de thread
  - Contador de programa
  - Set de registros
  - Stack

Thread

- Comparte con otros threads que pertenecen al mismo proceso:
  - Sección de código
  - Sección de datos
  - Otros recursos del SO como archivos abiertos
- Si un proceso tiene múltiples threads de control, puede realizar más de una tarea a la ves.
- Ejemplo – Un servidor web acepta múltiples requests para páginas, imágenes, sonidos y otros recursos

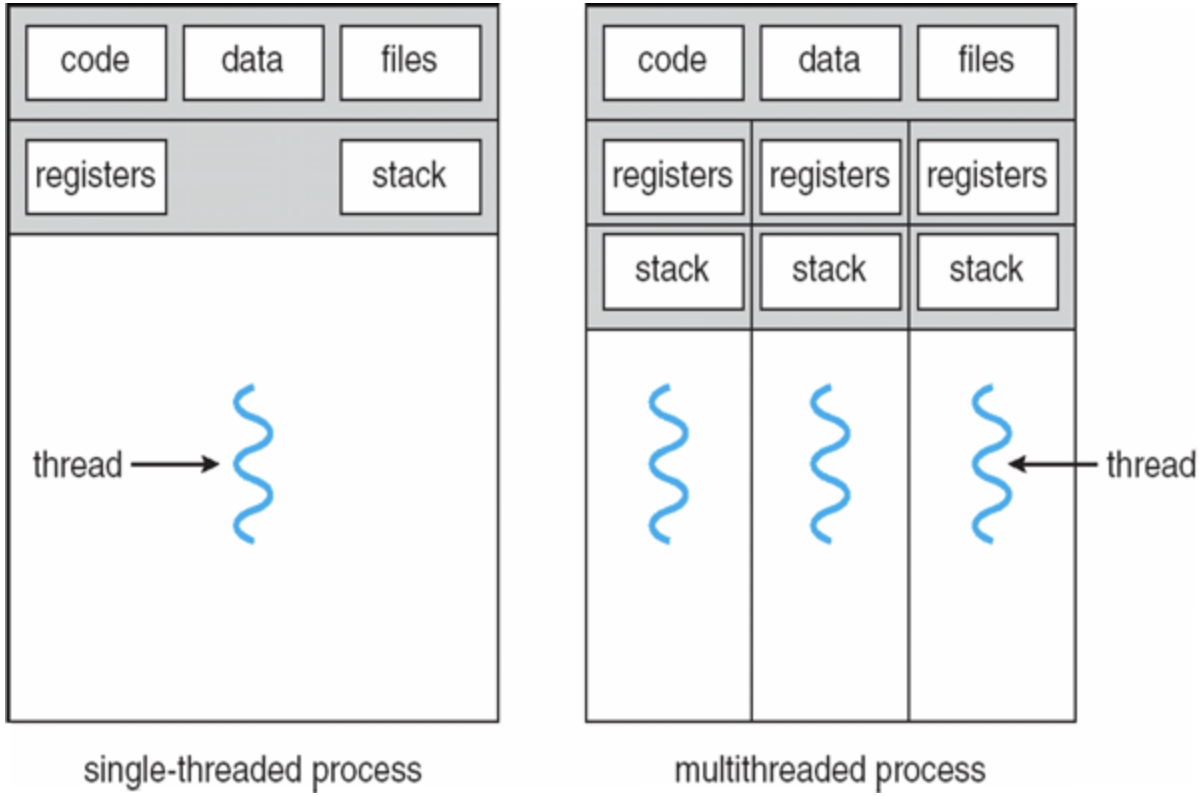
Servidor Multi-Thread



Procesador de Texto

> Procesador de Texto: formatea, muestra el texto, responde al input del usuario, chequea ortografía, ...

Proceso vs Thread



single-threaded process

multithreaded process

Beneficios

- **Respuesta** – permite a un programa continuar ejecutando aún cuando parte está bloqueada en una operación larga (p. ej. un web browser cargando una imagen)
- **Compartir Recursos** – los threads comparten memoria y otros recursos del proceso
- **Economía** – es más económico crear y cambiar de contexto threads (comparten los recursos del proceso al cual pertenecen)

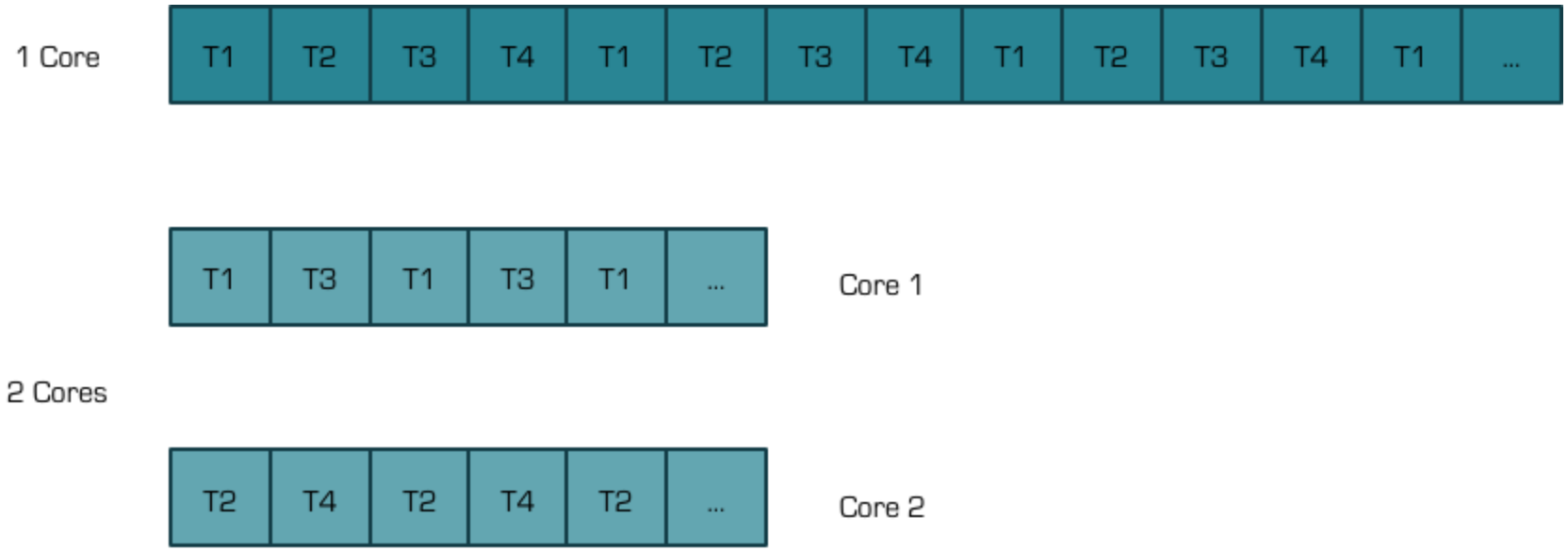
> En Solaris un crear un procesos es 30 veces más lento que crear un thread y 5 veces más lento en cambiar de contexto

- **Escalabilidad** (en arquitecturas multi-procesador) – el mismo proceso puede usar varias CPUs simultáneamente utilizando threads

Programación Multicore

- Los sistemas multicore imponen nuevos desafíos a los programadores:
  - **Dividir actividades** – encontrar partes del programa que pueden ser divididas en tareas concurrentes separadas y de esta forma ejecutar en paralelo en distintos cores
  - **Balanceo** – asegurar que cada tarea ejecuta trabajo del mismo valor
  - **División de datos** – asegurarnos que los datos se dividen para ser procesados en cores separados como las tareas.
  - **Dependencia de datos** – asegurar que la ejecución de tareas está sincronizada para obedecer a la - dependencia de datos
  - **Testing y debugging** – es mucho más difícil que en aplicaciones single-threaded dado que hay muchos caminos de ejecución diferentes cuando el programa ejecuta

1 Core vs 2 Cores



Threads a nivel Usuario y Kernel

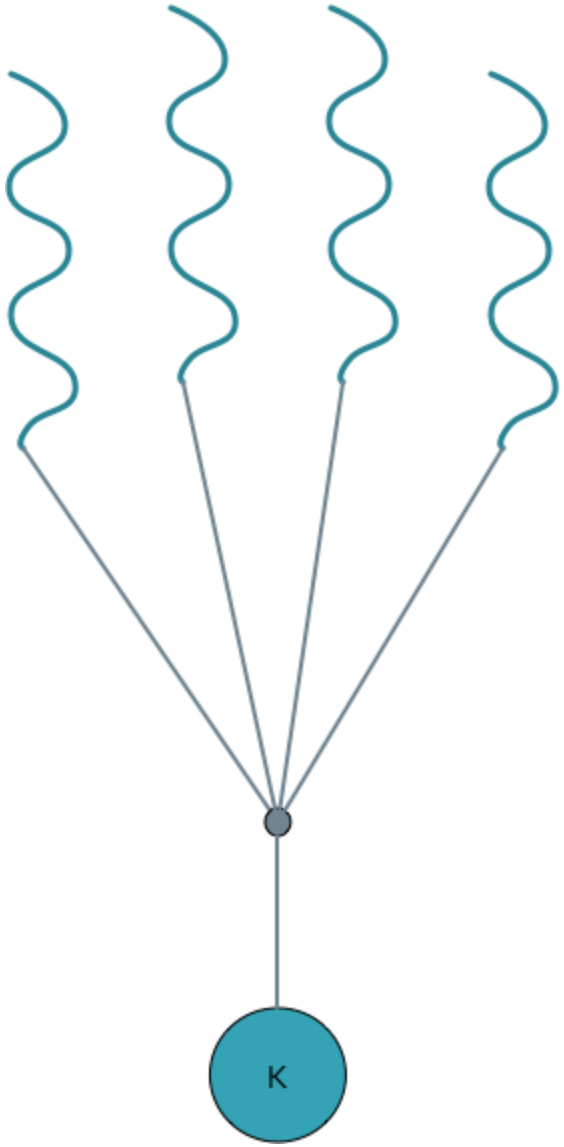
- Los threads del kernel (a veces llamados Lightweight Process - LWP) son creados y planificados por el kernel
    - Son más caros de crear que un thread del usuario
- Los threads del usuario son creados por una librería de threading y planificados por la librería (que ejecuta en modo usuario).
    - Todos los threads del usuario pertenecen al proceso que los creo
    - El kernel no sabe de la existencia de los threads del usuario
- La mayor diferencia se da cuando se usan sistemas multiprocesador:
- Los threads del usuario son completamente manejados por la librería, no pueden ejecutar en paralelo en diferentes CPUs
- Dado que los threads del kernel son planificados por el kernel, distintos threads pueden ejecutar en distintas CPUs

Models de Multithreading

- Muchos a Uno
- Uno a Uno
- Muchos a Muchos

Muchos a Uno

- Muchos threads a nivel usuario son mapeados a un único thread del kernel
- El manejo de los threads la hace la librería en espacio del usuario, es eficiente.
- Desventaja – el proceso entero se bloquea cuando el thread hace una llamada al sistema bloqueante
- Ejemplos: Solaris Green Threads | GNU Portable Threads

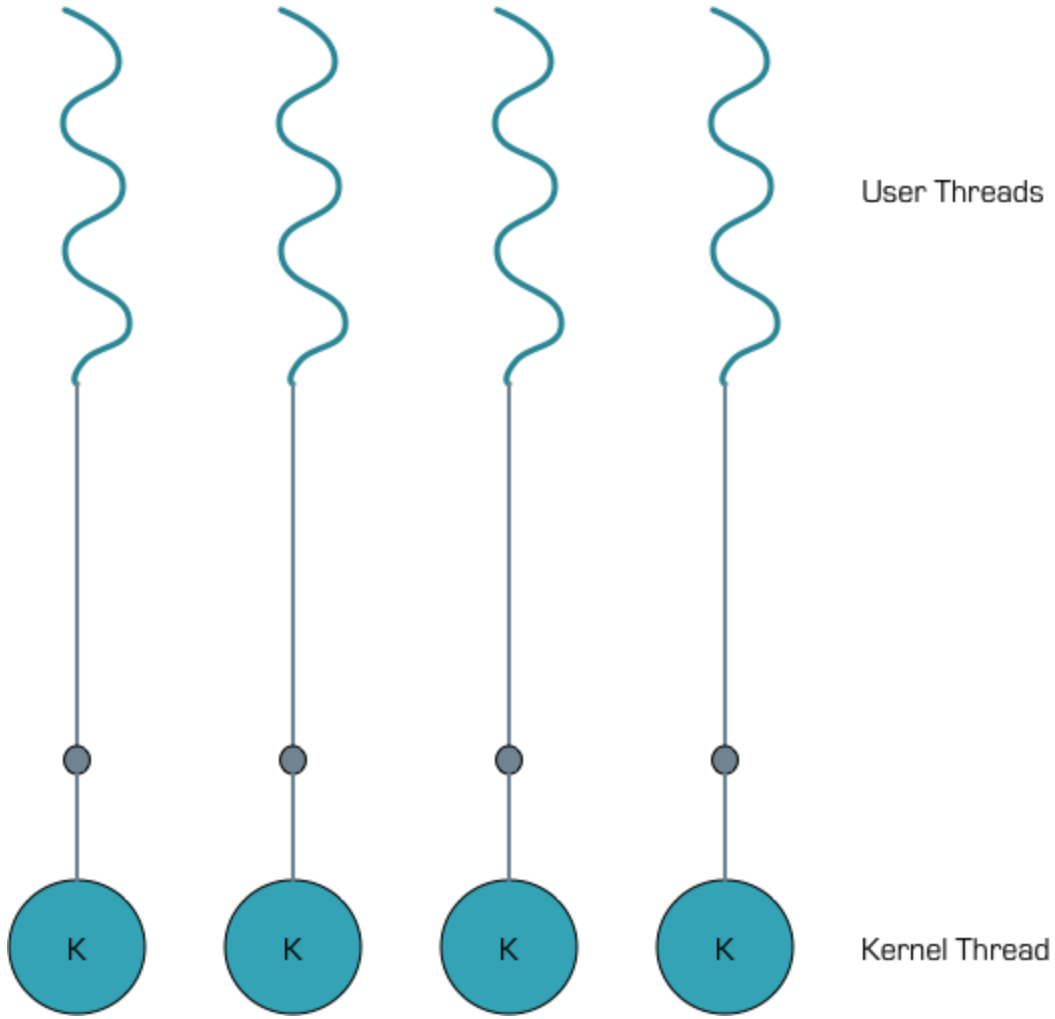


User Threads

Kernel Thread

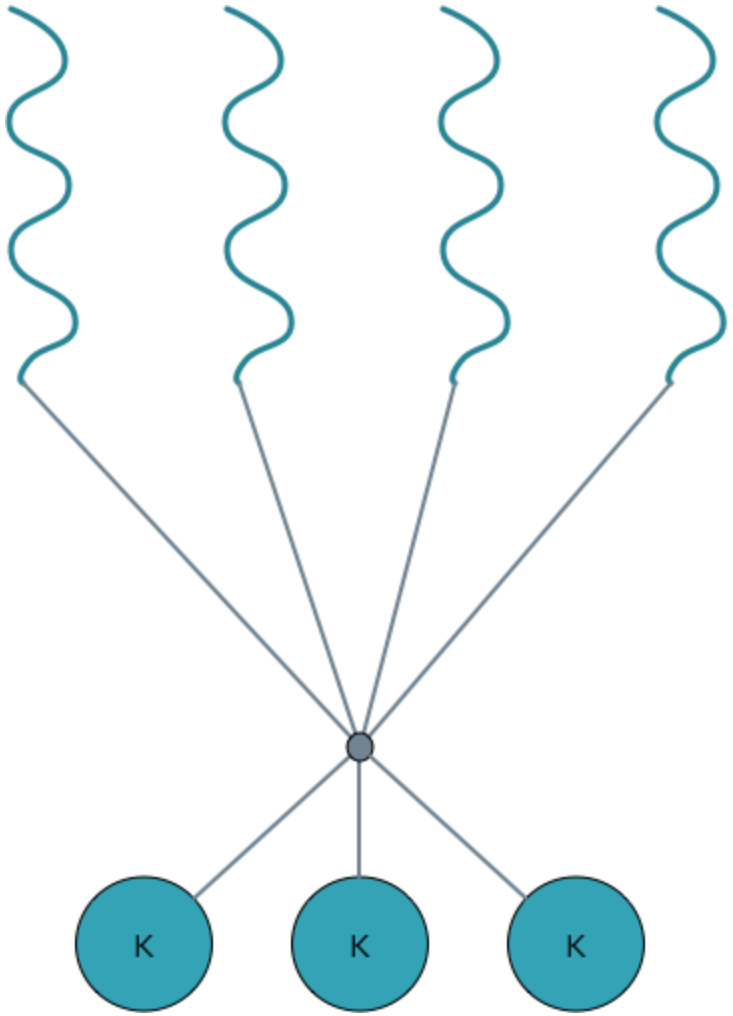
Uno a Uno

- Cada thread del usuario es mapeado a un thread del kernel
- Más concurrencia comparado al modelo muchos-a-uno dado que otro thread puede ejecutar cuando uno hace una llamada bloqueante
- Permite la ejecución en paralelo de múltiples threads en multiprocesadores
- Desventajas:
    - Crear un thread del usuario requiere la creación del correspondiente thread del kernel
    - La performance se va a degradar con lo cual la mayoría tiene una restricción en el número de threads permitidos
- Ejemplos: Windows NT/XP/2000 y posteriores, Linux, Solaris 9 y posteriores


K
K
K
K
User Threads
Kernel Thread

Muchos a Muchos

- Permite muchos threads del usuario ser mapeados a muchos threads del kernel
- Permite al SO crear la cantidad suficiente de threads del kernel
- Solaris previo a la versión 9, Windows NT/2000 con el paquete ThreadFiber

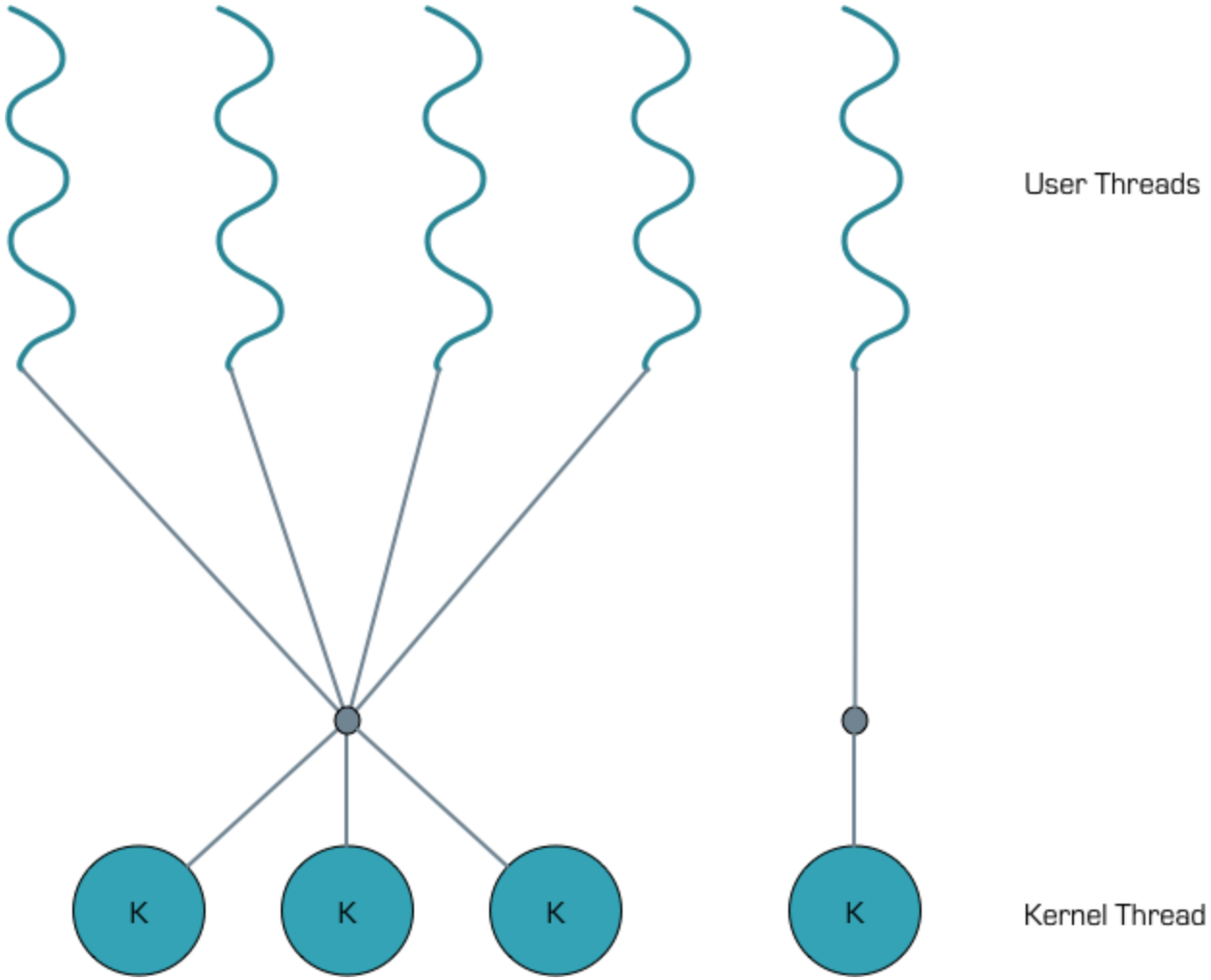


User Threads

Kernel Thread

Model de 2 Niveles

- Similar al modelo M:M, excepto que permite a un thread del usuario estar asociado a un thread del kernel fijo
- Ejemplos: IRIX, HP-UX, Tru64 UNIX, Solaris 8 y anteriores



User Threads

Kernel Thread

Librerías de Threads

- Las librerías de thread proveen una API para crear y manejar threads
- Dos formas principales de implementar
    - Librería completamente en espacio del usuario
    - Librería a nivel kernel soportada por el SO -> invocar una función de la librería termina en una llamada al sistema

PThreads

- Pueden ser provistos tanto a nivel usuario como a nivel kernel
- Es una API estándar POSIX (IEEE 1003.1c) para creación y sincronización de threads
- La API especifica el comportamiento de la librería de threads la implementación es tarea del desarrollador de la librería (p. ej. los diseñadores del SO)
- Común en sistemas UNIX (Solaris, Linux, Mac OS X)

Java/Python

- Los threads Java son manejados por la VM
- Generalmente implementados usando threads del SO (uno a uno)

Erlang/Go

- Los threads (procesos livianos) son manejados por la VM/Runtime
- Usan el modelo muchos a muchos
- Usan uno o más thread para manejar E/S

Consideraciones con Threads

- Semántica de las llamadas al sistema `fork()` y `exec()`
- Pool de Threads
- Información específica del Thread

Semántica del `fork()` y `exec()`

- La llamada `fork()` duplica sólo el thread que la invoca o todos?
- Si un thread invoca `fork()`, el nuevo proceso duplica todos los thtreads, o el nuevo proceso es single-threaded? Unix tiene ambas versiones
- Cuando se invoca la llamada `exec()`, el programa especificado reemplaza el proceso completo, incluyendo todos los threads. Por lo tanto, duplicar sólo el thread que invoca es lo apropiado.

Pool de Threads

- Crear un número de threads en un pool y esperar trabajo
- Ventajas:
  - Usulamente más rápido satisfacer un pedido que al crear un nuevo thread
  - Permite poner un límite al número de threads que utiliza una aplicación
- La creación de threads ilimitada puede sobrecargar el sistema, como tiempo de CPU y memoria

Información Específica

- Permite a cada thread tener su propia copia de los datos
- Es similar a una variable global, pero solo visible al thread
- Un uso típico es en los applications server para guardar datos de sesión del usuario