Threads

Semestre II-2013

Threads

- Visión General
- Programación en Multicore
- Modelos Multithreading
- Bibliotecas de Threads
- Manejo Implícito de Threads
- Consideraciones sobre los Threads
- Ejemplos a nivel del SO

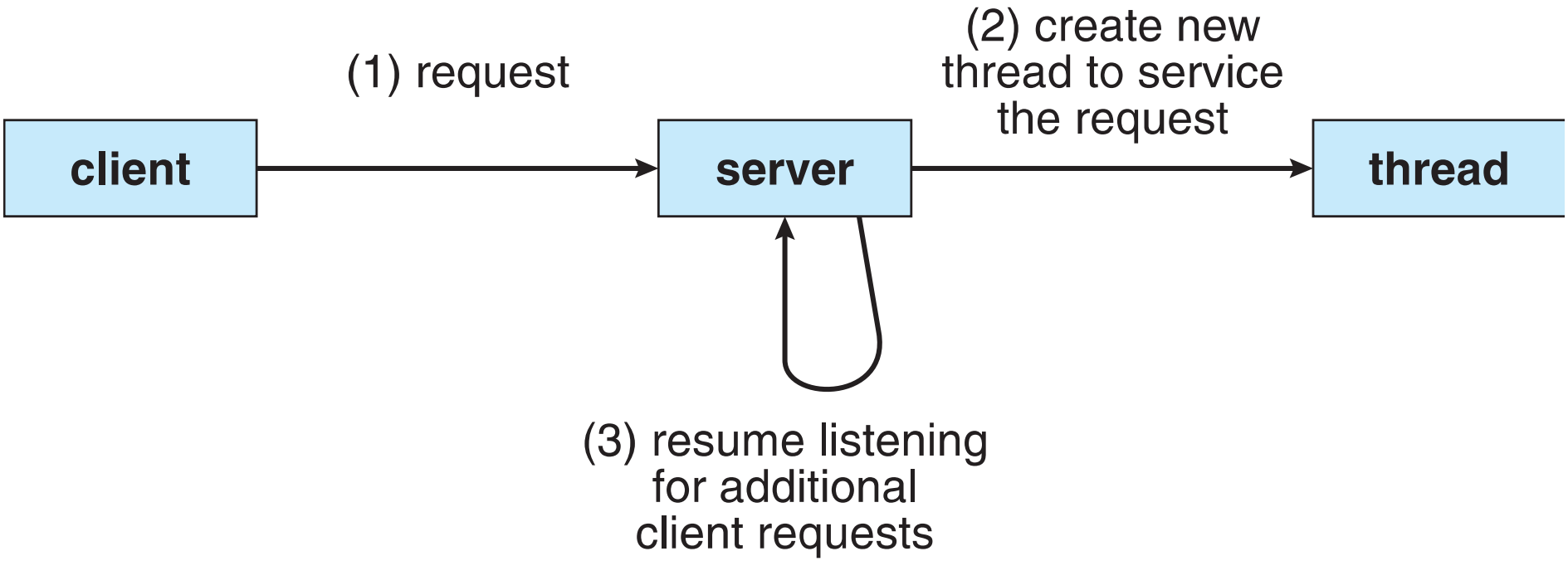
Objetivos

- Introducir la noción de Thread – Una unidad fundamental en la utilización del CPU, que establece las bases para sistemas de computación multithread
- Discutir los APIs y bibliotecas de Pthreads, Windows, y Java
- Explorar las diferentes estrategias para proveer el manejo implícito de threads
- Examinar las consideraciones relacionadas con la programación multithread
- Cubrir el soporte embebido en los SO Windows y Linux para el manejo de threads

Motivación

- Las aplicaciones modernas son multithreaded
- Los threads se ejecutan en la aplicación
- Múltiples tareas dentro de la aplicación pueden ser implementadas por threads separados
  - Actualización de la pantalla
  - Extraer o ubicar datos
  - Chequeo de ortografía
  - Preguntar por solicitudes de red
- La creación de un proceso es más pesada que la creación de un thread
  - ¿Por qué?
  - heavy-weight vs. light-weight
- Es posible simplificar la codificación e incrementar la eficiencia
- Los kernel generalmente son multithread

Arquitectura de un Servidor Multithread



Beneficios

- Capacidad de respuesta – Es posible continuar con la ejecución del proceso si parte del proceso se encuentra bloqueado, lo cual es realmente importante para interfaces de usuario
- Compartición de recursos – Los threads comparten los recursos del procesos, es más sencillo implementar los mecanismos de compartición de memoria y paso de mensajes
- Economía – Es más económico crear un proceso, incluso el intercambio de threads en ejecución produce menos sobrecarga que un cambio de contexto
- Escalabilidad – Los procesos pueden tomar ventaja de arquitecturas multiprocesador

Programación en Multicore

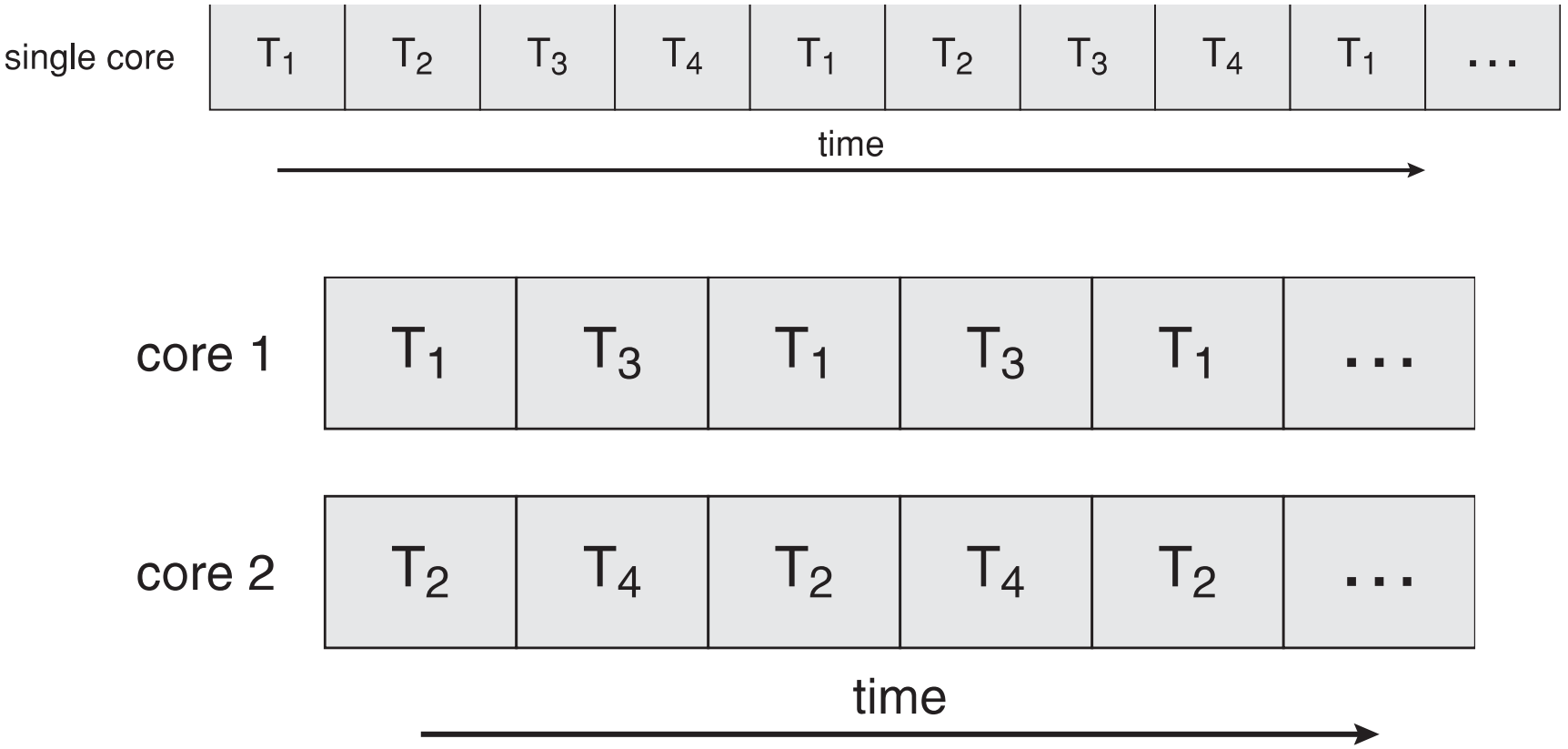
- Sistemas multicore o multiprocesador retan a los programadores, los retos incluyen:
  - División de actividades
  - Balanceo
  - Particionamiento de datos
  - Dependencia de datos
  - Mecanismos de prueba y depuración
- El paralelismo implica que el sistema debe ser capaz de realizar más de una tarea simultáneamente
- La concurrencia se refiere al soporte de realizar progresos en más de una tarea
  - En el caso de procesadores sencillos o single core, el planificador es el encargado de proveer concurrencia

Programación en Multicore

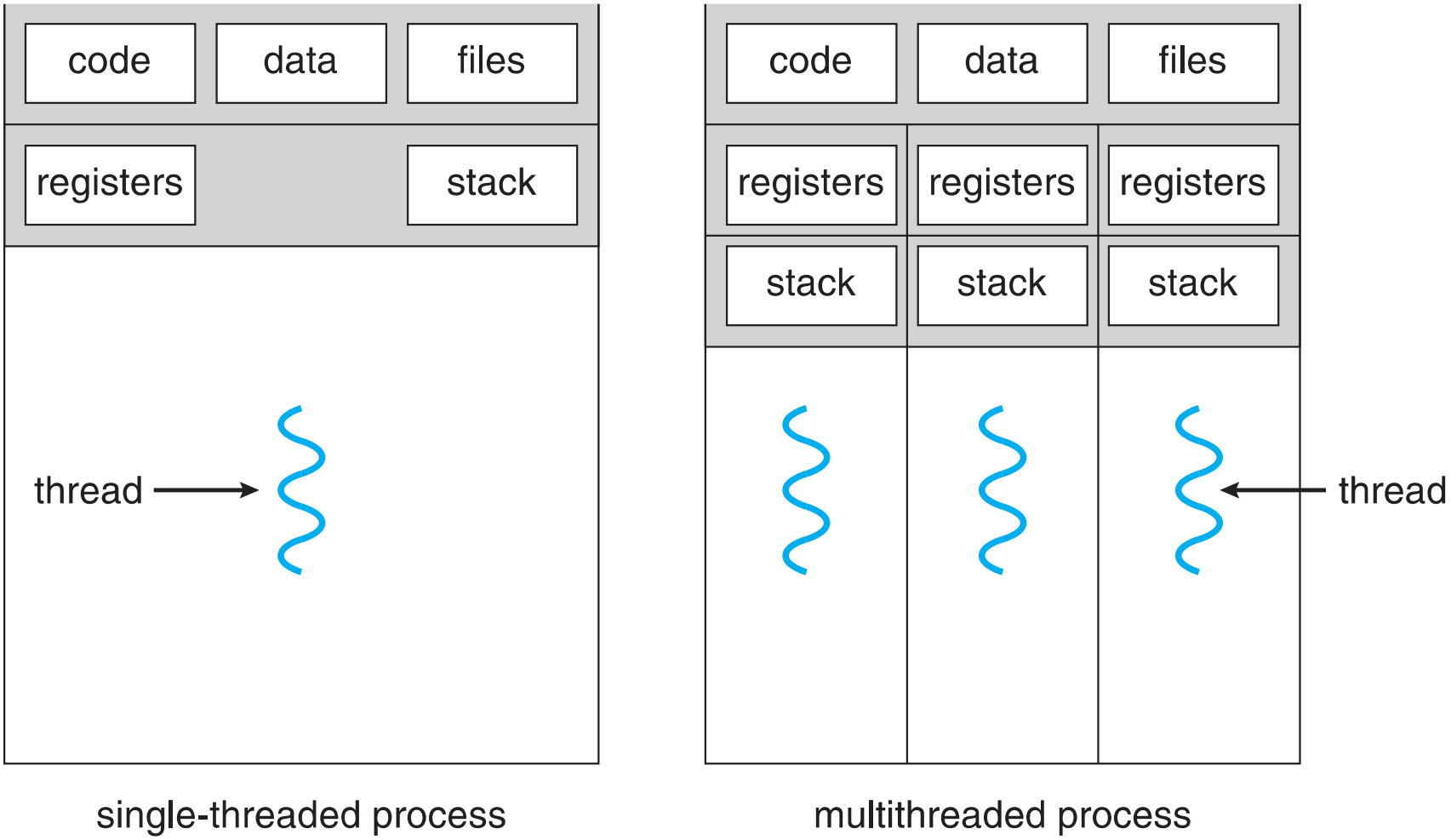
- Tipos de paralelismo
  - Paralelismo a nivel de datos – Distribuir un subconjunto de los datos a través de los cores, y ejecutar la misma operación en cada uno de ellos
  - Paralelismo a nivel de tareas – Distribuir threads a través de los cores, y que cada thread realice operaciones únicas o independientes
- Dado que el # de threads aumenta, también lo hace el soporte de threads a nivel de arquitectura
  - Los CPUs tienen múltiples núcleos con soporte de threads
  - Considere el Oracle SPARC T4 con 8 núcleos, y 8 threads por núcleo

Concurrencia vs. Paralelismo

- Ejecución concurrente en un sistema single-core vs. paralelismo en un sistema multi-core



Procesos Single y Multithreaded



single-threaded process

multithreaded process

Ley de Amdahl

- Permite identificar la ganancia a nivel de desempeño al agregar cores adicionales a un aplicación que posee tanto componentes seriales como paralelos
- S es la porción serial
- N representa en número de cores de procesamiento
- Por ejemplo, si una aplicación es 75% paralela y 25% serial, y se mueve de 1 core a 2 cores esto resulta en un incremento de la velocidad de 1.6 veces
- Si N tiende a infinito, la ganancia en velocidad tiende a 1/S
- **La porción serial de una aplicación tiene un efecto desproporcionado en la ganancia en cuanto al desempeño cuando se agregan cores adicionales**
- Con base a lo discutido acá
  - ¿Cree usted que la ley de Amdahl considera los sistemas multicore de la actualidad?

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Threads de Usuario y Kernel

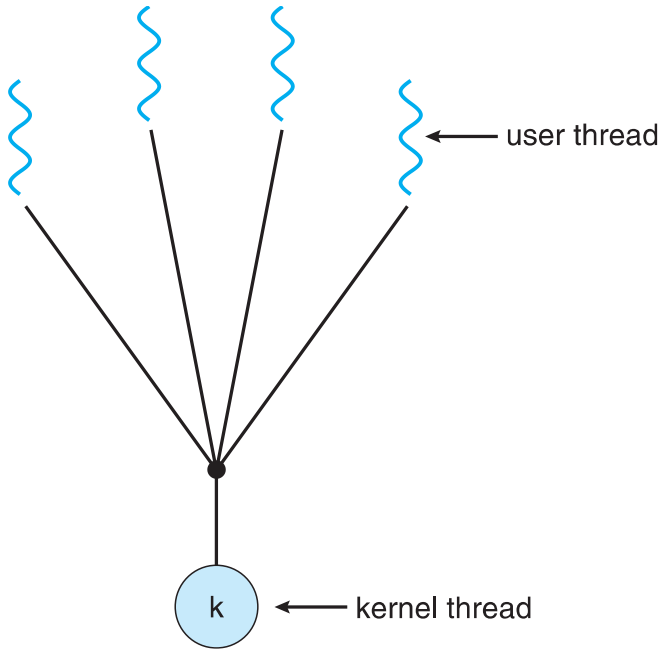
- Threads de usuario – Manejados vía una biblioteca de threads a nivel de usuario
- Existen tres bibliotecas de threads bien conocidas:
  - POSIX threads
  - Win32 threads
  - Java threads
- Threads de kernel – Suportados directamente por el kernel del SO
- Ejemplos – Casi todos los SO de propósito general lo incluyen:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Modelos de Multithreading

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

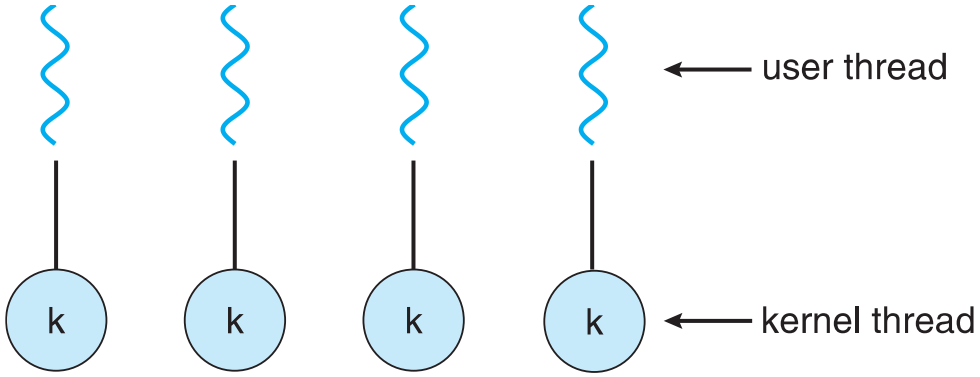
Muchos a Uno

- Muchos threads a nivel de usuario se mapean a un único thread a nivel de kernel
- Si un thread se bloquea, esto causa que todos los demás se bloqueen
  - ¿Por qué?
- Múltiples threads no pueden ejecutarse en paralelo en sistemas multicore
  - Solo un thread de usuario se mapea con un thread de kernel
- Algunos sistemas actuales utilizan este modelo
  - Solaris Green Threads
  - GNU Portable Threads



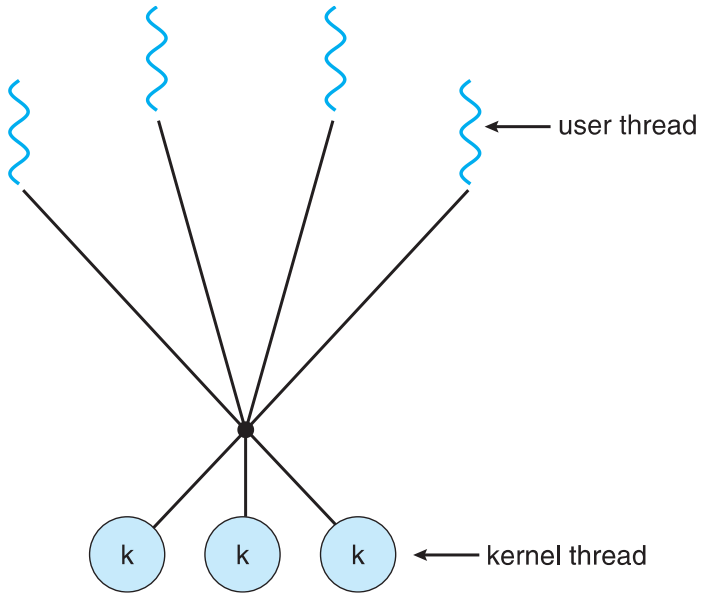
Uno a Uno

- Cada thread a nivel de usuario se mapea a un thread a nivel de kernel
- La creación de un thread a nivel de usuario implica la creación de un thread a nivel de kernel
- El nivel de concurrencia es superior que en el nivel Muchos a Uno
- En ocasiones el número de threads por proceso se restringe debido a decisiones de desempeño (sobrecarga)
- Ejemplos
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 y posteriores



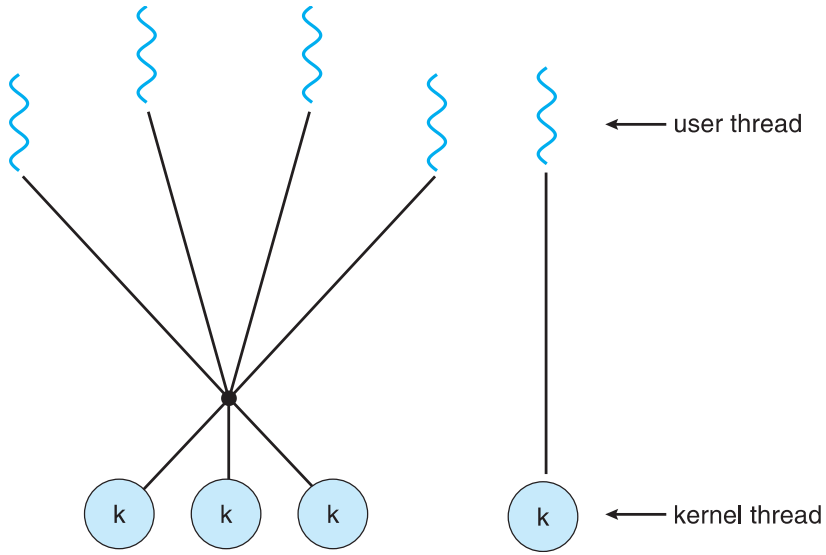
Muchos a Muchos

- Permite que múltiples threads de usuarios sean mapeados a múltiples threads de kernel
- Permite que el SO se encargue de la creación de thread a nivel de kernel de forma eficiente
- Ejemplos:
  - Solaris previo a su versión 9
  - Windows NT/2000 si se instala y configura el paquete ThreadFiber



Modelo en Dos Niveles

- Similar a M:M, excepto que permite que un thread de usuario sea confinado a un único thread de kernel
- Ejemplos:
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 y anteriores



Bibliotecas de Threads

- Una biblioteca de threads provee al programador de un API para crear y manejar threads
- Las dos principales formas de implementación son:
  - Bibliotecas enteramente en el espacio de usuario
  - Bibliotecas en el espacio de kernel soportadas por el SO

Pthreads

- Puede proveer threads a nivel de usuario y a nivel de kernel
- El estándar POSIX IEE 1003.1c provee un API para la creación y sincronización de threads
  - Provee una especificación, no una implementación
- El API especifica el comportamiento de la biblioteca de threads, mientras que la implementación lidia con el desarrollo de la biblioteca
- Es muy común en SO UNIX
  - Solaris
  - Linux
  - Mac OS X

Ejemplo Pthreads

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Ejemplo Pthreads

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Ejemplo Pthreads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.

Ejemplo Win32 API

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Ejemplo Win32 API

```
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

   if (ThreadHandle != NULL) {
       /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Java Threads

- Los threads en Java son manejados por la JVM
- Típicamente implementados usando el modelo de threads provisto por el SO subyacente
- Los threads en Java pueden ser creados:
  - Heredando la clase Thread
  - Implementando la interface Runnable

```
public interface Runnable
{
    public abstract void run();
}
```

Programa Multithread en Java

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

Programa Multithread en Java

```
public class Driver
{
   public static void main(String[] args) {
    if (args.length > 0) {
     if (Integer.parseInt(args[0]) < 0)
      System.err.println(args[0] + " must be >= 0.");
     else {
      Sum sumObject = new Sum();
      int upper = Integer.parseInt(args[0]);
      Thread thrd = new Thread(new Summation(upper, sumObject));
      thrd.start();
      try {
         thrd.join();
         System.out.println
                  ("The sum of "+upper+" is "+sumObject.getSum());
      } catch (InterruptedException ie) { }
     }
    }
    else
     System.err.println("Usage: Summation <integer value>"); }
}
```

Manejo Implícito de Threads

- La creciente popularidad de lo threads en conjunto con un uso masivo a nivel de programación, han ocasionado que sea difícil una programación correcta usando thread de forma explícita
- ¿Entonces? → Threads Implícitos
- La creación y manejo de threads es tarea del compilador y la biblioteca de threads, en lugar del programador
- Tres métodos se han explorado:
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Otros métodos incluyen:
  - Microsoft Threading Building Blocks (TBB)
  - El paquete java.util.concurrent

Thread Pools

- Se crea un número N de threads en un pool donde esperan para la iniciar su ejecución
- Ventajas:
  - Usualmente es más rápido servir una solicitud a través de un thread existente, que tener que crear uno para atenderla
  - Permite confinar el numero de threads de la aplicación de acuerdo al tamaño del pool
  - Con esta estrategia es posible implementar diferentes esquemas de ejecución o planificación:
    - Por ejemplo, las tareas podrían ejecutarse de forma periódica
- El API de threads en Windows soporta pools

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

OpenMP

- Se establecen directivas en el compilador y en los APIs para C, C++, FORTRAN
- Se provee soporte de programación paralela en un ambiente de memoria compartida
- Es requerido la identificación de regiones paralelas – Bloque de código que puede ser ejecutado en paralelo
- #pragma omp parallel
  - Crea tantos threads como cores tenga el sistema
- #pragma omp parallel fof
  - for(i=0;i<N;i++)
    - c[i] = a[i]+b[i],
- El código anterior ejecuta el ciclo en paralelo

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

Grand Central Dispatch

- Tecnología Apple para los SO Mac OS X y iOS
- Es una extensión de los APIs de los lenguajes C y C++, además de algunas bibliotecas
- Permite la identificación de secciones paralelas
- Gestiona la mayor parte de los detalles vía threads
- Bloques de código ente "^{}" - ^{printf(I´m a block);}
- Estos bloques se colocan en una cola de despacho
  - Se asigna a cada bloque en la cola de despacho un thread de un pool al removerlo del pool

Grand Central Dispatch

- Dos tipos de cola de despacho:
  - Serial – Los bloques se remueven en orden FIFO, la cola se procesa, y es llamada main queue
    - Los programadores pueden crear queues seriales dentro de sus programas
  - Concurrente – Los bloques se remueven en orden FIFO, pero varios bloques pueden removerse a la vez
    - Existen tres colas en el sistema basadas en prioridades, las cuales son: low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Consideraciones en Threads

- Semántica de las llamadas al sistema fork() y exec()
- Manejo de señales
  - Síncronas y asíncronas
- Cancelación del thread
  - Asíncrona o diferida
- Almacenamiento local del thread
- Activación del planificador

Semántica de fork() y exec()

- ¿Una llamada a fork() duplica el espacio de direcciones del thread llamante o de todos los threads embebidos en el proceso?
  - Algunas versiones de UNIX implementan dos versiones de fork
- La llamada al sistema exec() generalmente se comporta normal – Reemplaza el espacio de direcciones del proceso que se encuentra en ejecución, incluyendo todos sus threads

Manejo de Señales

- Las señales se usan en sistemas UNIX para notificar a un proceso que un evento en particular ha ocurrido
- Un manejador de señales es utilizado para procesar señales
  1. Una señal es generada por un evento en particular
  2. La señal es entregada al proceso
  3. La señal es manejada por manejador de señales:
     1. Por defecto
     2. Definido por el usuario
- Cada señal tiene asociado un manejador de por defecto a nivel de kernel
  - Si el manejador de señales se define por el usuario, este reemplaza o sobrescribe al manejador por defecto
  - Para ambientes single-thread, una señal se entrega al proceso

Manejo de Señales

- ¿A dónde o a quién debe entregarse una señal cuando el ambiente es multi-thread?
  - Se entrega la señal al thread al cual aplica la señal
  - Se entrega la señal a cada thread en el proceso
  - Se entrega la señal a determinados threads del proceso
  - Se asigna a un thread para que este reciba todas las señales del proceso

Cancelación de Threads

- Terminar todos los threads antes de finalizar.
- El thread a ser cancelado se conoce como: **target thread**
- Dos posible aproximaciones:
  - Una cancelación asíncrona que termine al target thread de inmediato
  - Una cancelación diferida la cual permite que el target thread verifique por si mismo si debe ser cancelado. Este chequeo debe realizarse de forma periódica
- Código Pthread para crear y cancelar un thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Cancelación de Threads

- La invocación a una cancelación de thread en realidad depende del estado del thread
- Si un thread tiene la cancelación deshabilitada, la cancelación se mantendrá pendiente hasta que el thread la habilite.
- Por defecto la cancelación es diferida
  - La cancelación ocurre solo cuando el thread alcanza el punto de cancelación
    - Por ejemplo, pthread_testcancel()
    - Entonces se procede a la cancelación
- En sistemas Linux, la cancelación de threads se maneja a través de señales

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

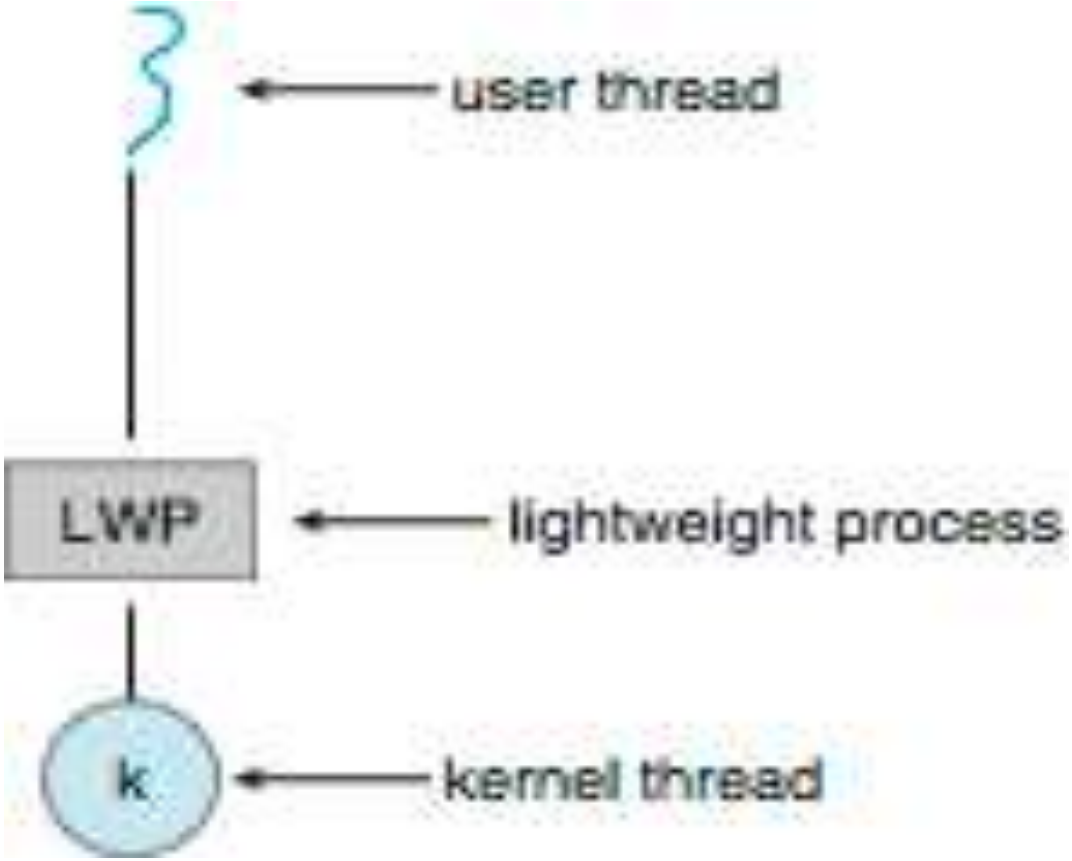
Espacio de Almacenamiento del Thread

- El Thread-Local Storage (TLS) permite que cada thread tenga su propia copia de datos
- Es útil cuando no se tiene control sobre el proceso de creación del threads (por ejemplo, cuando se utiliza un pool de threads)
- Diferente de las variables locales
  - Las variable locales solo son visibles durante la invocación de una única función
  - El TLS es visible a través de la invocación de funciones
- Similar a un dato **static**
  - Un thread solo tiene asociado un TLS

Activación del Planificador

- Tanto los modelos M:M y dos niveles requieren comunicación para mantener el número apropiados de threads de kernel mapeados a las aplicaciones
- Típicamente se utiliza una estructura de datos intermedia entre los threads de usuario y kernel – LightWeight Process (LWP)
  - Aparece como un procesador virtual en el cual se pueden planificar los threads que se ejecutan a nivel de usuario
  - Cada LWP se asocia a un thread de kernel
  - ¿Cuántos LWPs se crean?
- La activación del planificador provee **upcalls** – Un mecanismo de comunicación desde el kernel hasta el **upcall handler** en la biblioteca de threads
- Esta comunicación permite mantener la asociación correcta entre threads de usuario y threads de kernel

Activación del Planificador



Manejo de Threads en SO

- Windows XP Threads
- Linux Threads

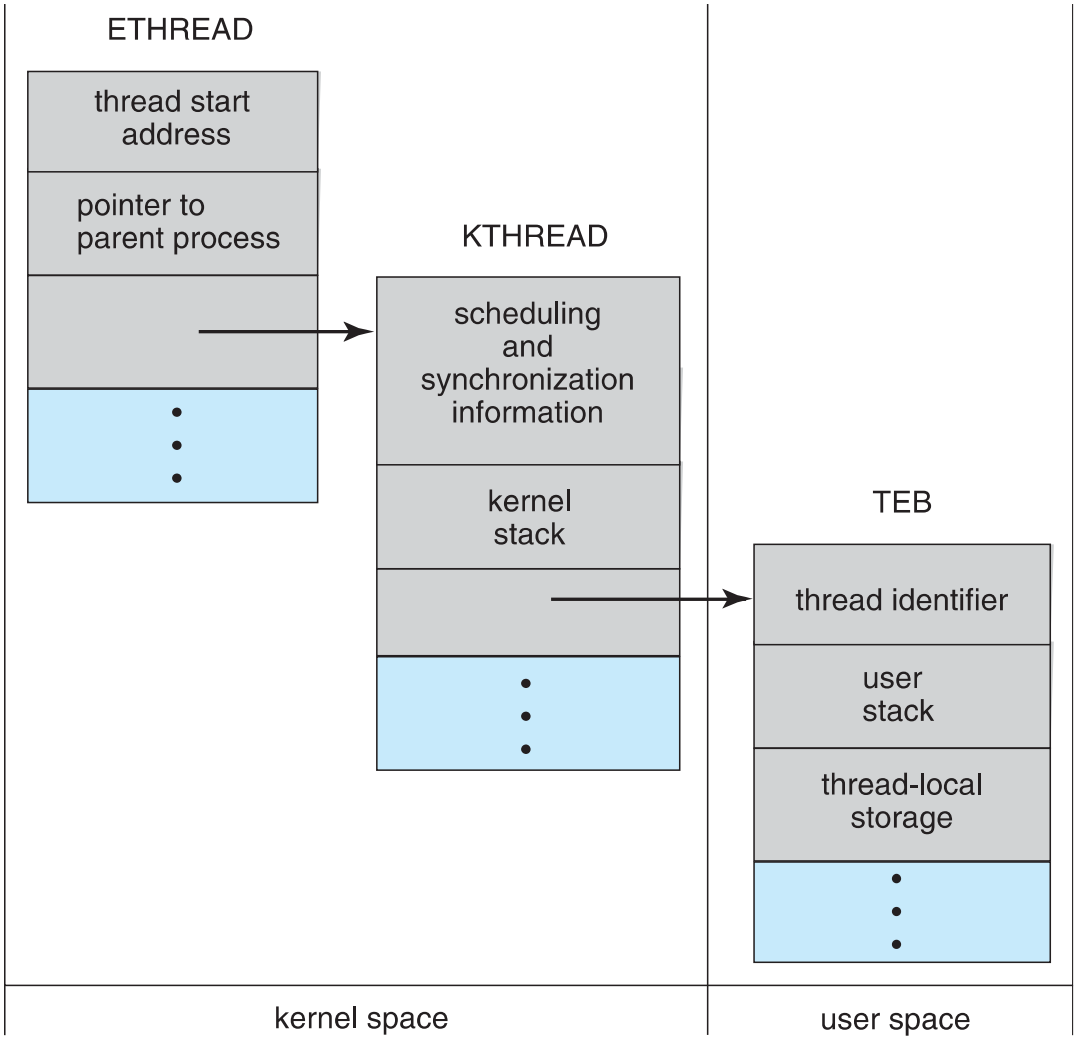
Windows Threads

- Windows implementa el API de Windows – Existe un API para Win98, Win NT, Win 2000, Win XP, y Win 7
- Implementa el modelo uno a uno a nivel de kernel
- Cada thread contiene:
  - Un ID de thread
  - El valor de los registro que representan el estado del procesador
  - Una pila separada para el espacio de usuario y kernel, para cuando el thread se ejecute en cada uno de los modos
  - Un área de almacenamiento privado utilizado por las bibliotecas dinámicas usadas por el thread
- El valor del conjunto de registro del procesador, las pilas, y el área de almacenamiento privado se conoce como el **contexto** del thread

Windows Threads

- Las estructuras de datos primarias del thread incluyen:
  - ETHREAD (executive thread block) – Incluye un apuntador al proceso al cual pertenece y un apuntador a KTHREAD → Espacio de kernel
  - KTHREAD (kernel thread block) – Información de planificación y sincronización, pila a nivel de kernel, apuntador a TEB → Espacio de kernel
  - TEB (thread environment block) – ID del thread, pila a nivel de usuario, almacenamiento local del thread → Espacio de usuario

Estructuras de Datos en Windows XP Threads



Linux Threads

- Linux se refiere a ellos como **tasks** en lugar de threads
- La creación de threads se realiza a través de la llamada al sistema clone()
- clone() permite a la task hijo compartir el espacio de dirección del task padre (proceso)
- Las siguientes banderas controlan el comportamiento de la llamada clone()

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.