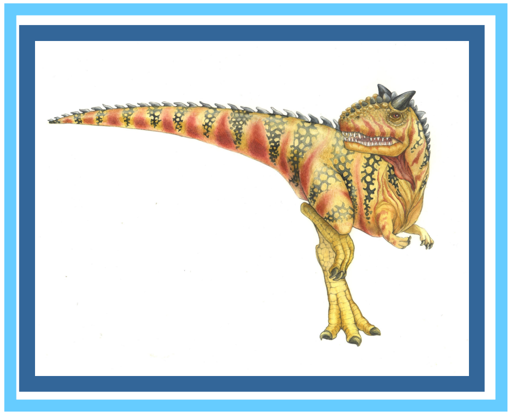
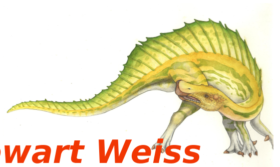
Chapter 4: Threads



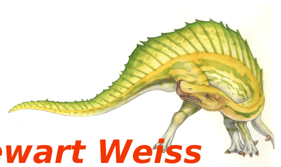
Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Linux Threads



Objectives

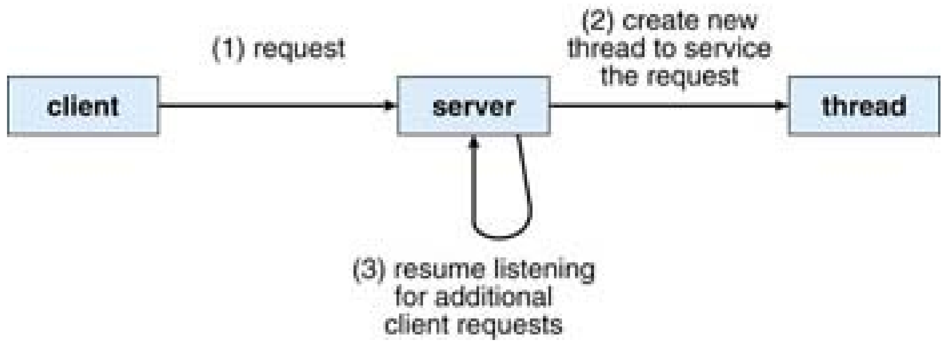
- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads thread library
- To examine issues related to multithreaded programming



modified by Stewart Weiss

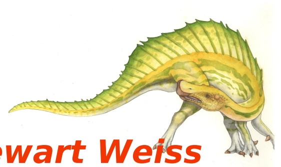
Why?

- Context switches are expensive – entire context must be copied (slow!!)
- Sometimes new process needs to share part of context of parent, so it is wasteful to copy the entire context
- Threads were invented to overcome these two problems.
- Example: a concurrent web server is one that creates a new process each time a client attempts a connection. A traditional server would fork a new process with its own address space, slowing things down. A multi-threaded server would fork a thread, much faster, that could use parent's context.


client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

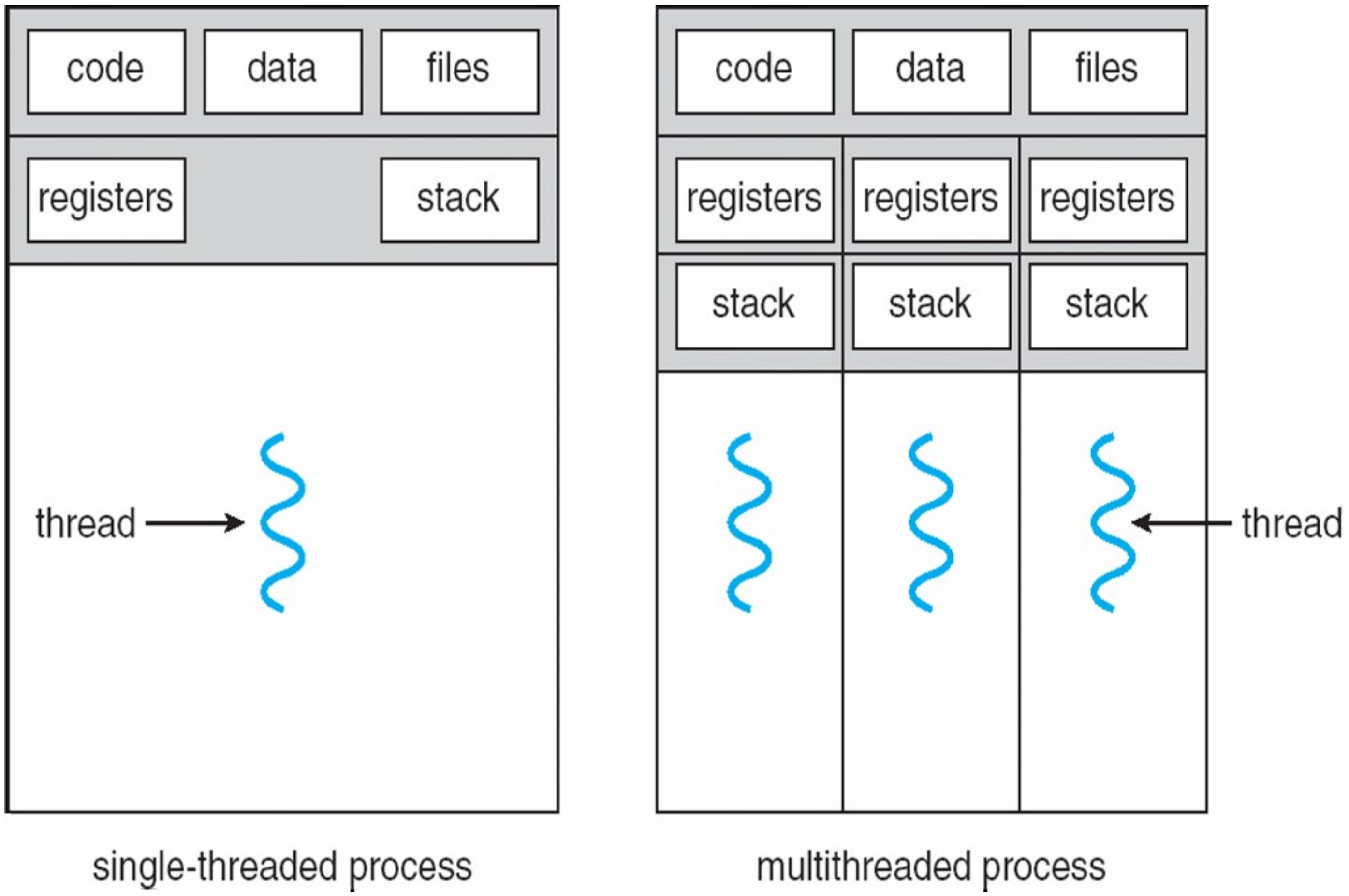
What is a Thread?

- A ***thread*** is a basic unit of CPU utilization consisting of a program counter, register set, a stack., local data, and other thread-specific properties. It is also called a ***lightweight process*** (***LWP***). In contrast, a traditional process is a ***heavyweight*** process.
- Threads can share the context of their parents, and hence siblings can share context, and thus a whole process tree can share it. E.g. -- can share code, variables, open files, data.
- The collection of all threads is called a ***task***.



modified by Stewart Weiss

Single and Multithreaded Processes

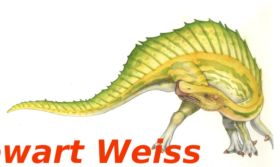


Benefits

- ***Responsiveness***: multithreading an application increases responsiveness. One thread can issue I/O request and be blocked while another does computation. While data loads, another user request may be satisfied.
- ***Resource Sharing***: threads share memory and resources without special OS primitives like message-passing or shared memory system calls, making it easier and faster. ***BUT*** – puts burden on programer to prevent race conditions (Chapter 6)
- ***Economy***: much faster to create a thread than a process; also because resources are shared, fewer resources are used in total.
- ***Scalability***: threads can be scheduled on separate processors when they are available, making execution faster. So can processes, but they require more resources, so mroe threads can be scheduled than processors, when the number of processors is very large.

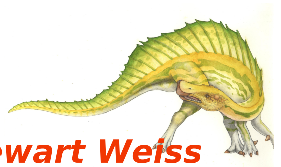
Comparison with Processes

- The differences between a thread and a process:
  - Exists within a process and uses the process resources
  - Has its own independent flow of control as long as its parent process exists and the OS supports it
  - Duplicates only the resources it needs to be independently schedulable
  - May share process resources with other threads that act equally independently (and dependently)
  - Is terminated if the parent process dies



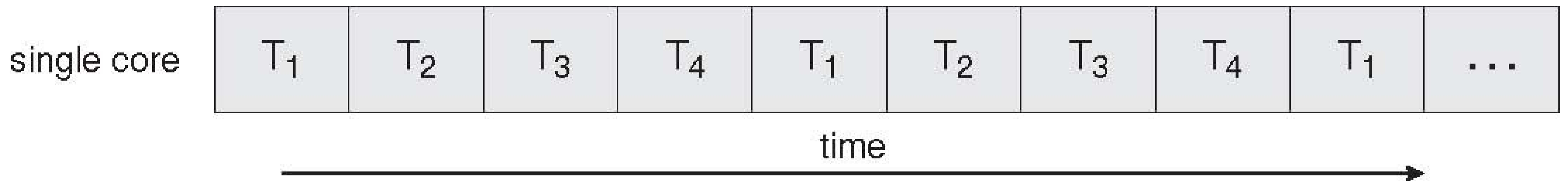
Multicore Programming

- A multicore computer has multiple processors on a single chip. Each core is an independent CPU. An operating system has greater challenges when using a multicore system, including
  - **Multiprocessor scheduling:** deciding which CPU to put each process on
  - **Load Balancing** : deciding how to rebalance queues to make sure each CPU is kept equally busy
  - **Cache and Data Coherence:** when processes move from one CPU to another, making that caches and data remain consistent and coherent

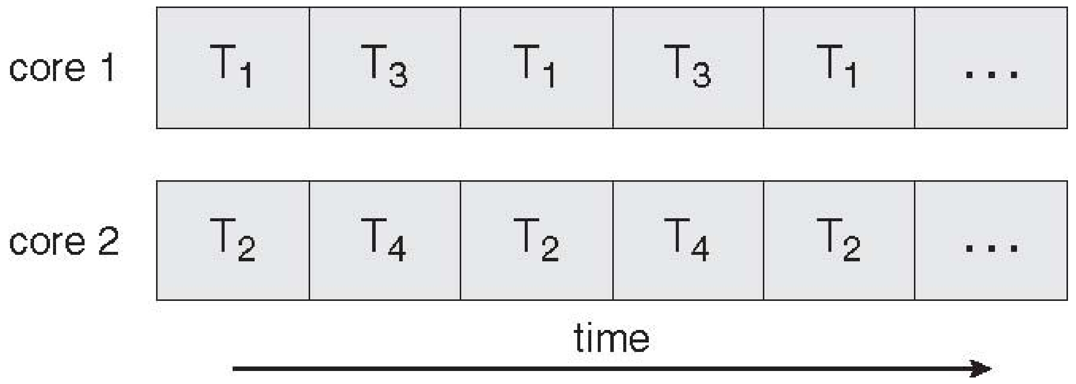


modified by Stewart Weiss

Concurrent Execution



Threads serialized on Single Core System



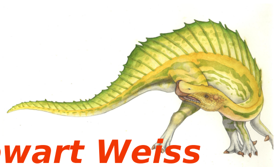
core 1
T_1 T_3 T_1 T_3 T_1 . . .
core 2
T_2 T_4 T_2 T_4 T_2 . . .
time

Threads executed in parallel on Two Core System

modified by Stewart Weiss

Multicore Programming

- Multicore systems putting pressure on application programmers also; challenges include
  - **Dividing activities:** finding tasks in applications that can run concurrently
  - **Balance**: trying to balance the amount of work allocated to each thread of execution
  - **Data splitting**: dividing up the data into that which is global and that which is local to each thread
  - **Data dependency**: analyzing which data will be accessed by multiple threads in order to decide how to synchronize access to that data
  - **Testing and debugging**: testing multi-threaded programs is much harder because executions are not repeatable.



User Threads

- There are two types of threads: ***user threads*** and ***kernel threads***.
- User threads are supported by user-level libraries outside of the kernel and do not need calls to the OpSys for switching.
- Easier to implement cooperation among concurrent processes with user threads
- Disadvantage: If kernel is single-threaded and a user thread makea system call, entire task is blocked.
- Three primary user level thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

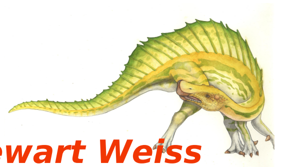
modified by Stewart Weiss

Kernel Threads

- Supported by the Kernel: kernel manages creation, deletion, and all thread operations.
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- User threads are mapped onto kernel threads.
  - Many-to-One: many user threads per kernel thread
  - One-to-One: one user thread per kernel thread
  - Many-to-Many: user threads multiplexed across multiple kernel threads

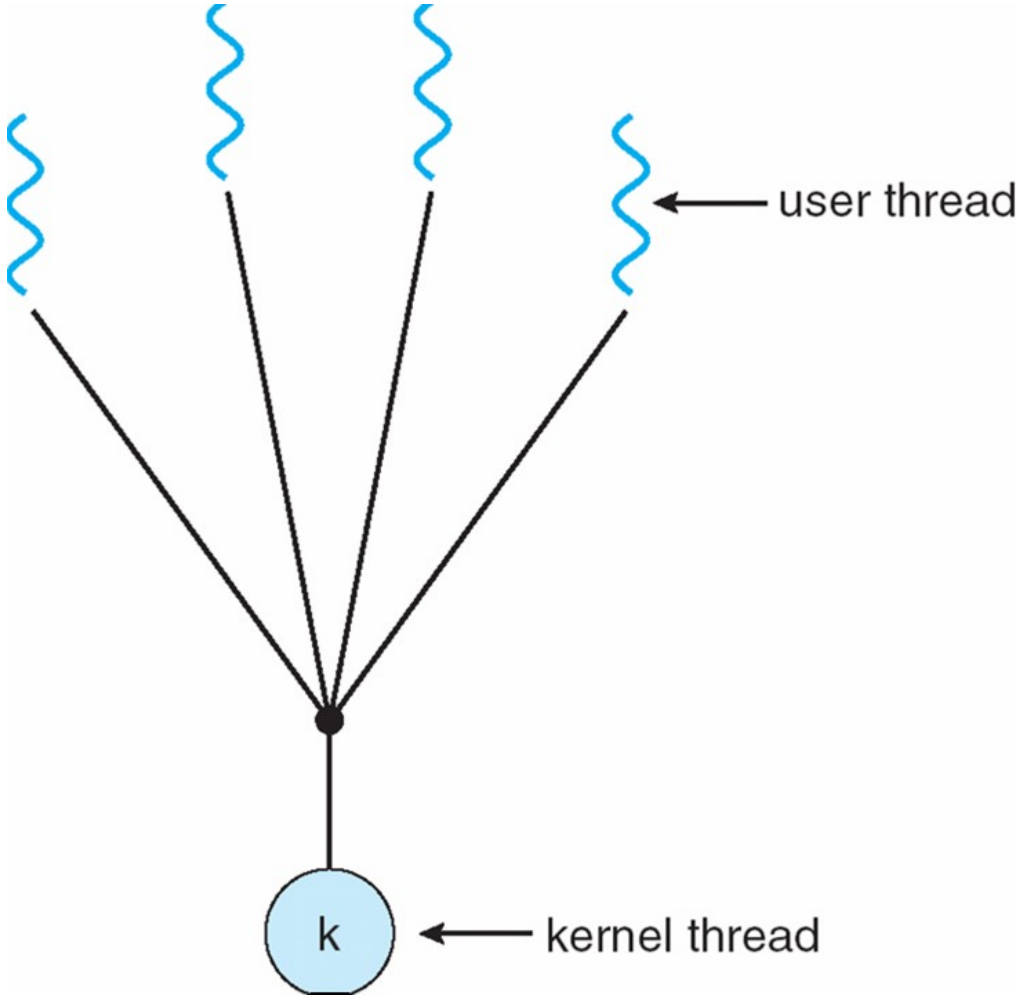


modified by Stewart Weiss

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

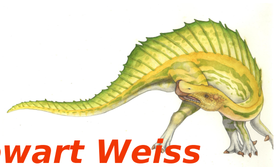
Many-to-One Model



modified by Stewart Weiss

Many-to-One: Pros & Cons

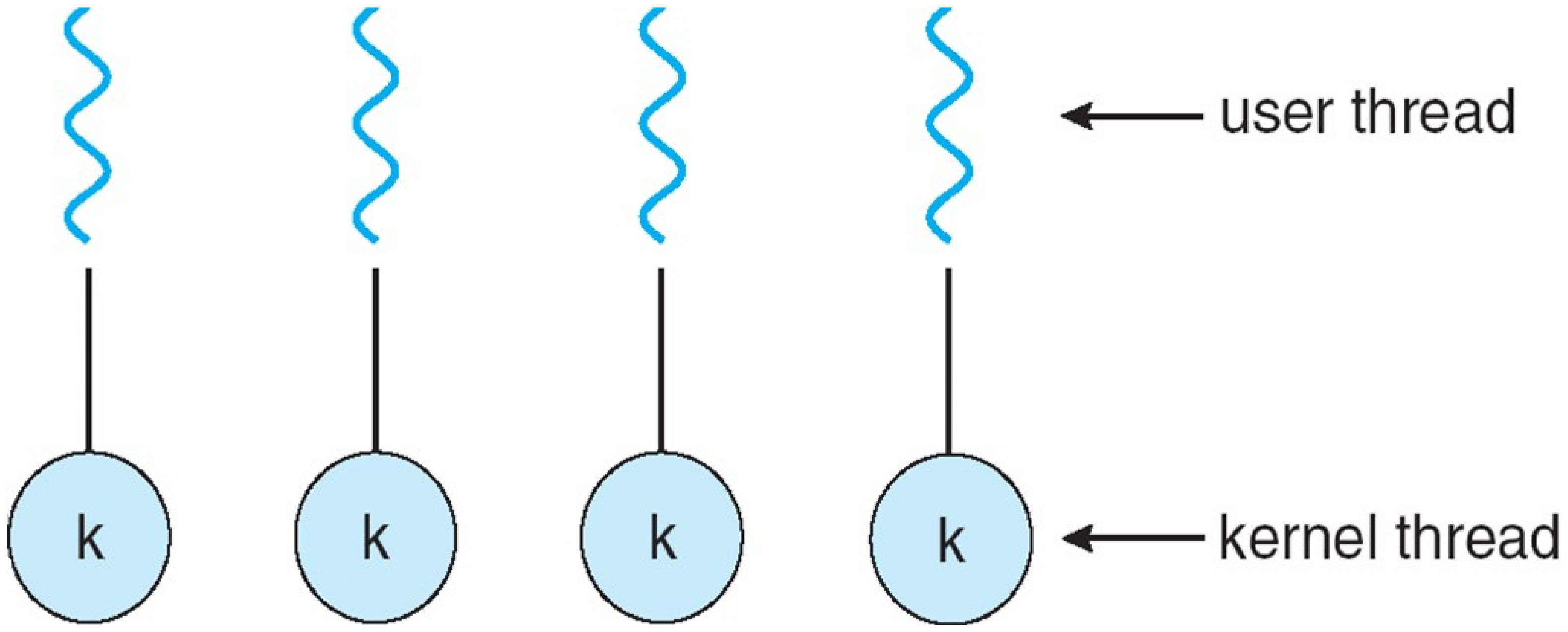
- Pros
  - Efficient and fast
- Cons
  - If any one of the threads mapping to the same kernel thread makes a blocking call, all threads of the group are blocked because the kernel thread gets blocked.
  - It is not true concurrency; only one user thread per kernel thread can run at a time.



One-to-One

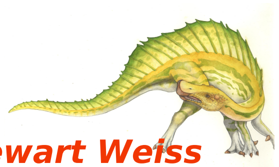
- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

One-to-one Model



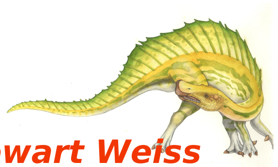
One-to-One: Pros & Cons

- Pros:
  - More concurrency because each user thread can run even if others are blocked
- Cons
  - More overhead in OpSys

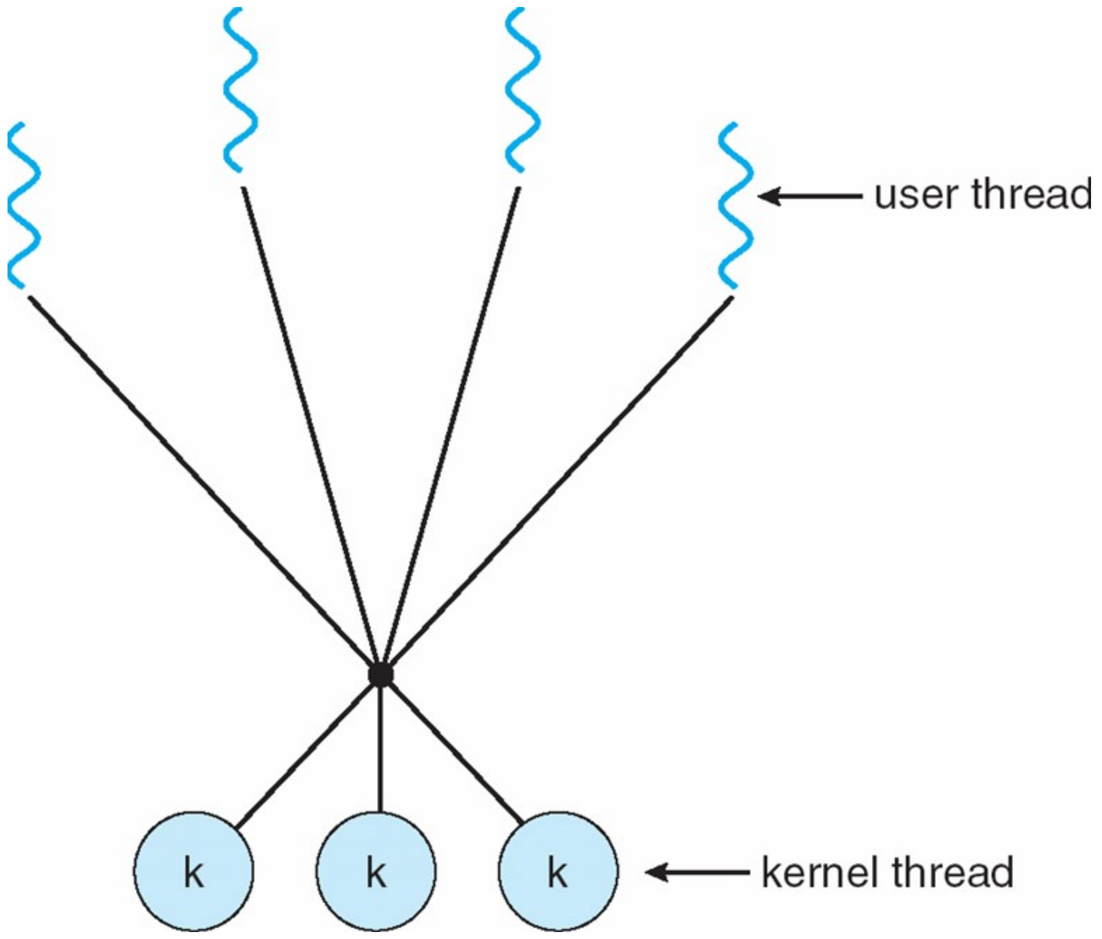


Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Many-to-Many Model



Many-to-Many Model: Pros and Cons

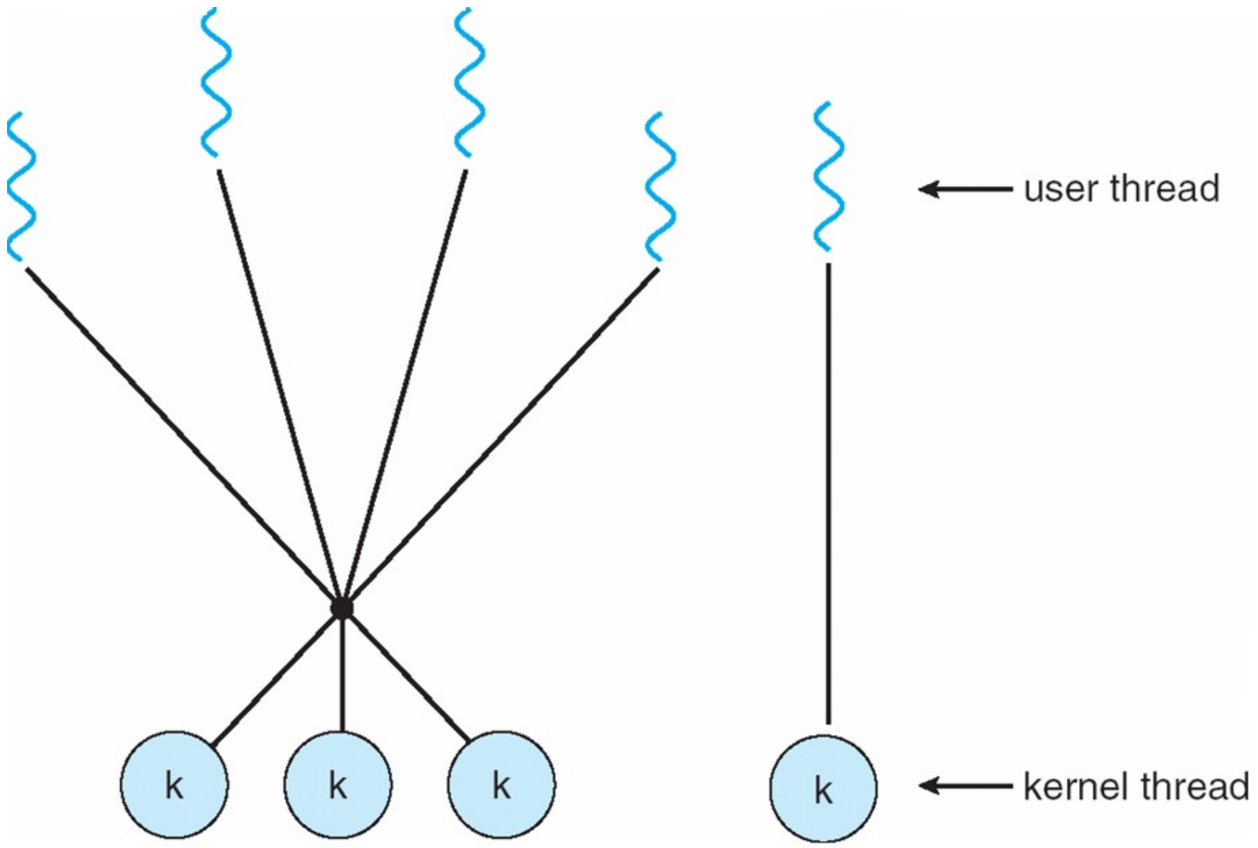
- Pros
  - When a user thread blocks, the kernel thread that it was running on is blocked, but other user threads can be scheduled on other kernel threads.
  - The degree of concurrency is equal to the number of kernel threads.
- Cons
  - Greater complexity in the kernel.

modified by Stewart Weiss

Two-level Model

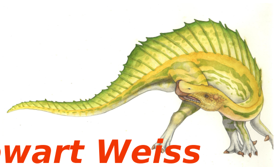
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread also, so it is both one-one and many-many.
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model (2)



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS



modified by Stewart Weiss

Pthreads

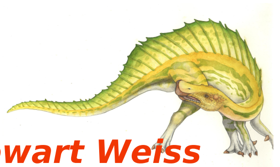
- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Threading Issues

- Semantics of **fork()** and **exec()** system calls:
  - If a thread in a multithreaded program calls fork() are all threads duplicated, or just one? (Linux duplicates just the calling thread)
  - If a thread calls an exec() call, the entire task is replaced, including all other threads.
- Thread cancellation of target thread (Cancellation means termination.)
  - Asynchronous or deferred:
  - asynchronous – terminated by some other thread unpredictably
  - deferred – like polling; thread checks periodically whether to terminate
  - Issue is how to reclaim resources in an asynchronous cancellation



modified by Stewart Weiss

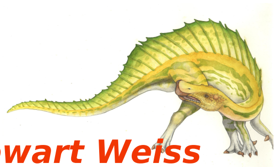
Example: Linux Threads

- Linux provides clone() to create threads, tkill() to send signals to or kill them, and a few other system calls to mage their resources.
- The clone() call is passed parameters that specify how the thread is to be created: who is the parent, what resources are shared, etc.
- Examples of those parameters:

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.

Linux Threads continued

- Linux refers to them as *tasks* rather than *threads*
- **clone()** allows a child task to share the address space of the parent task (process)



End of Chapter 4