

PART 5

MULTITHREADING MODELS

gettyimages®

Daniel Grill

Threads Implementation

User-Level Threads

Kernel-Level Threads

Multithreading Models

One-to-one

Many-to-one

Many-to-many

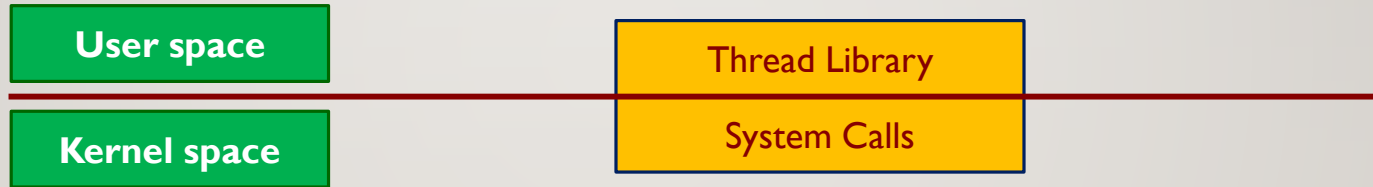
Two-level

INTRODUCTION

There are two broad categories of threads implementation

- **User-level threads:** these are managed by **thread libraries** in the user space
- **Kernel-level threads:** these are managed by **system calls** in the kernel space

The following figure illustrates both categories:




USER-LEVEL THREADS (ULTs) (I) – THREAD LIBRARIES

As previously mentioned, ULTs are managed by **thread libraries**.

Therefore, a thread library belongs to the user space.

A **thread library** is a collection of programs whose target is to manipulate ULTs.

Examples of the programs in the thread library include:

- 
1. Create threads
 2. Destroy (kill) threads
 3. Passing messages between threads
 4. Scheduling thread execution
 5. Saving & restoring thread contexts (context switching between threads)

USER-LEVEL THREADS (ULTs) (2) – KERNEL ACTIVITY

Since ULTs are managed by the thread library, then the kernel is not aware of ULTs.

When a thread makes a **system call**, the whole process is blocked. However, the thread is still in the running state from the point of view of the thread library.

In other words, thread states are independent of process states.

The advantages of ULTs may be summarized in the following points:

- The OS is not invoked in thread switching
- Threads scheduling is application-specific. The best algorithm is selected accordingly.
- Threads are portable; ie. they can run on any OS. They just need the thread library.

The disadvantages of ULTs may be summarized in the following points:

- Most system calls are blocking. Therefore, the kernel blocks the relevant process. Consequently, all threads of this process are blocked from the perspective of the kernel.
- Since the kernel is not aware of the ULTs, two threads belonging to the same process cannot run simultaneously on two CPUs. In such case, the OS assigns processes, rather than threads, to the CPUs.

KERNEL-LEVEL THREADS (KLTs)

In this model, there is no thread library. Threads are therefore managed by the kernel.

Accordingly, switching between threads invokes the kernel (OS). Therefore, the kernel maintains context information for both processes and threads.

Scheduling is performed on a thread-basis.

The advantages of ULTs may be summarized in the following points:

- Since the OS is aware of the KLTs, a multithreaded process may be assigned to multiple CPUs simultaneously.
- Blocking is performed on thread-basis rather than process-basis.
- Kernel routines can be multithreaded.

The disadvantages of ULTs may be summarized in the following points:

- Since the OS is invoked in thread switching, then the latter is relatively slow as compared to ULTs.

UTLs vs. KLTs

The following table summarizes the differences between ULTs and KLTs:

User-Level Thread	Kernel-Level Thread
Faster to create and manage	Slower to create and manage
Implemented by a thread library in the user space	OS supports creation of KLTs
A ULT is portable; ie. can run on any OS	A KLT is specific to an OS
Multithreaded applications cannot take advantage of multiprocessing	Multithreaded applications take advantage of multiprocessing.
	Kernel routines themselves can be multithreaded

MULTITHREADING MODELS (I) – INTRODUCTION

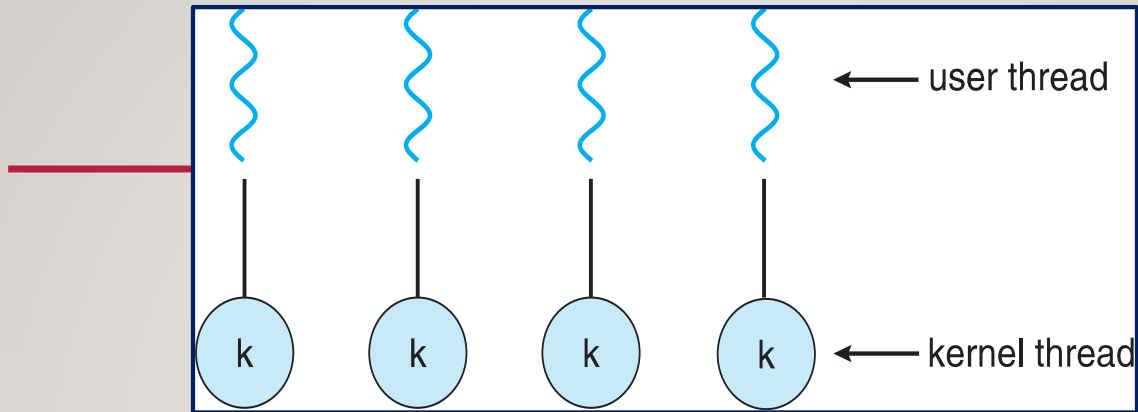
Some OSs provide a combined ULT/KLT approach.

Within this concept, there are four multithreading models:



MULTITHREADING MODELS (2) – ONE-TO-ONE

The following figure depicts the one-to-one multithreading model:



Each ULT is mapped to a different KLT.

The advantages of one-to-one multithreading model may be summarized in:

- When a ULT is blocked due to a system call, another thread may run.
- Provides concurrency since it allows the simultaneous execution of ULTs on multiple CPUs.

The disadvantage of one-to-one multithreading model is:

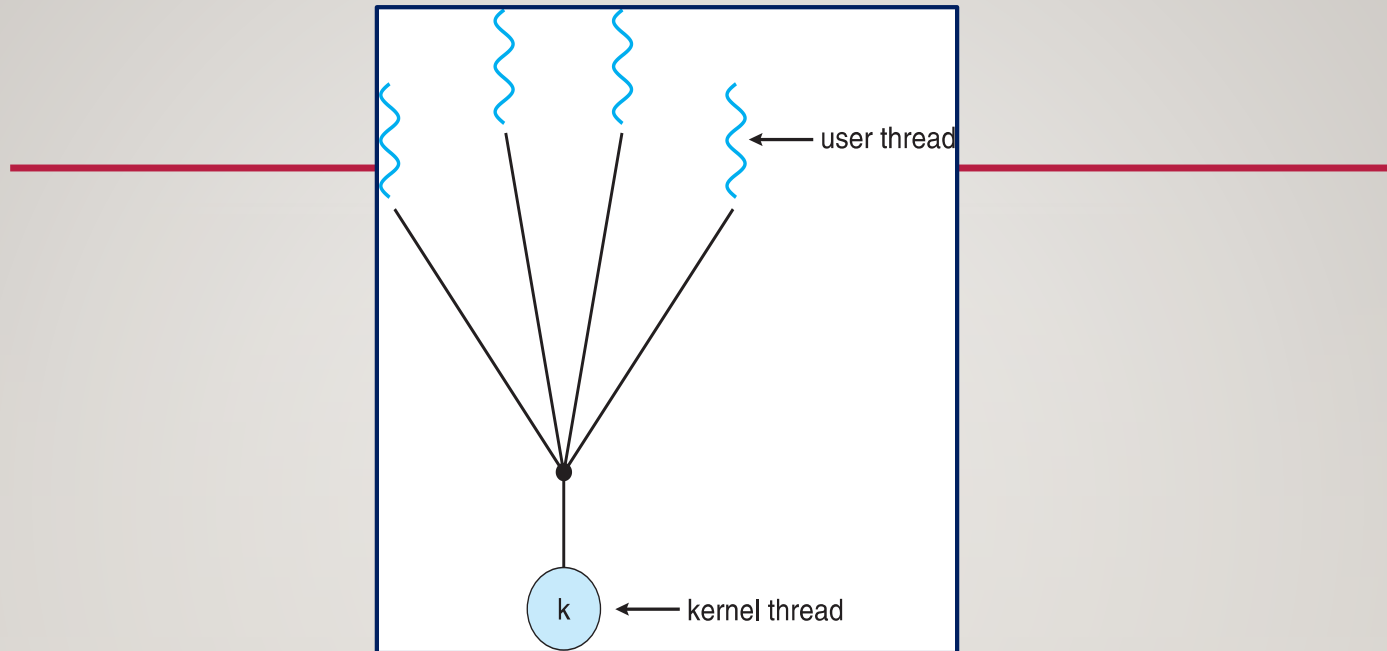
- Creation of KLT for each ULT causes an overhead. Accordingly, some systems restrict the number of threads per process.

The one-to-one multithreading model is implemented in:

- Solaris 9
- Windows
- Linux

MULTITHREADING MODELS (3) – MANY-TO-ONE

The following figure depicts the many-to-one multithreading model:



Many ULTs are mapped to a single KLT

The disadvantages of ULTs may be summarized in the following points:

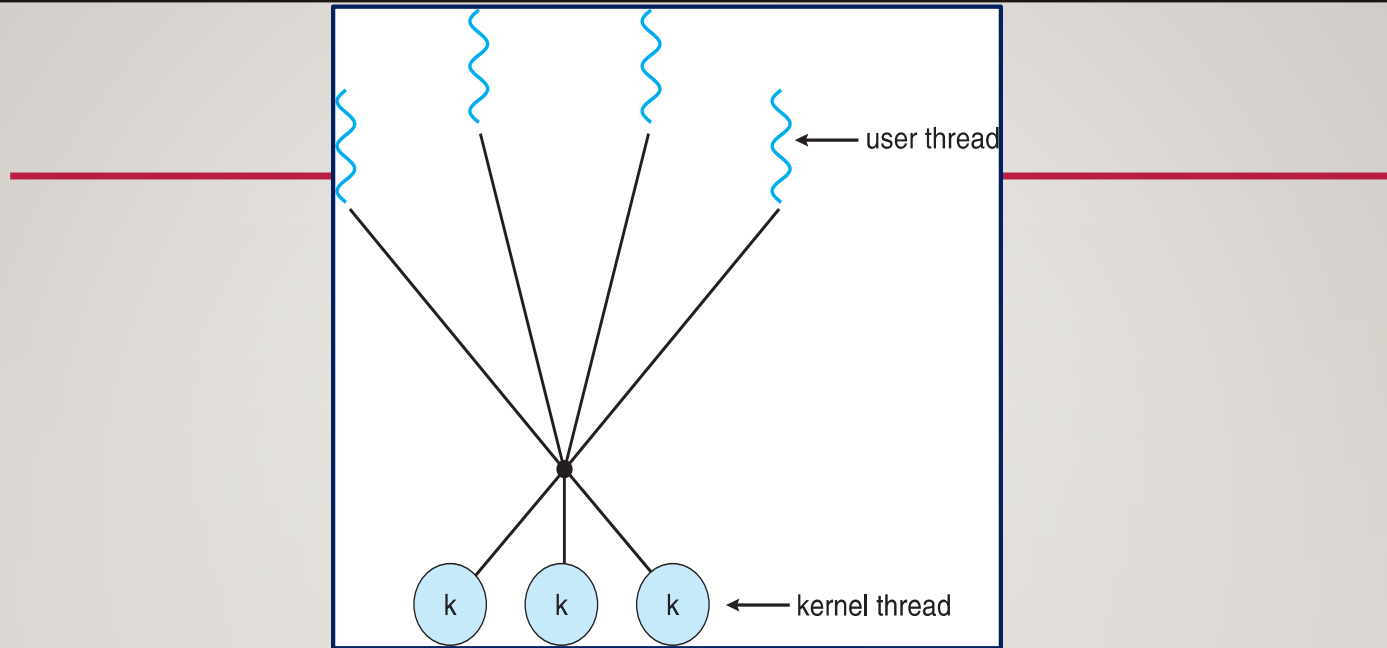
- When one ULT is blocked, all other threads are blocked.
- Multiple threads cannot run simultaneously on a multicore system because only one ULT can access the kernel at a time.

The many-to-one multithreading model is implemented currently in few systems:

- Solaris Green Threads
- GNU Portable Threads

MULTITHREADING MODELS (4) – MANY-TO-MANY

The following figure depicts the many-to-many multithreading model:



Many ULTs are mapped to many KLTs.

However, the number of KLTs may be less or equal to the number of ULTs.

The number of KLTs to be created is specified by the hardware.

The advantage of many-to-many is mainly:

→ Simultaneous execution is possible.

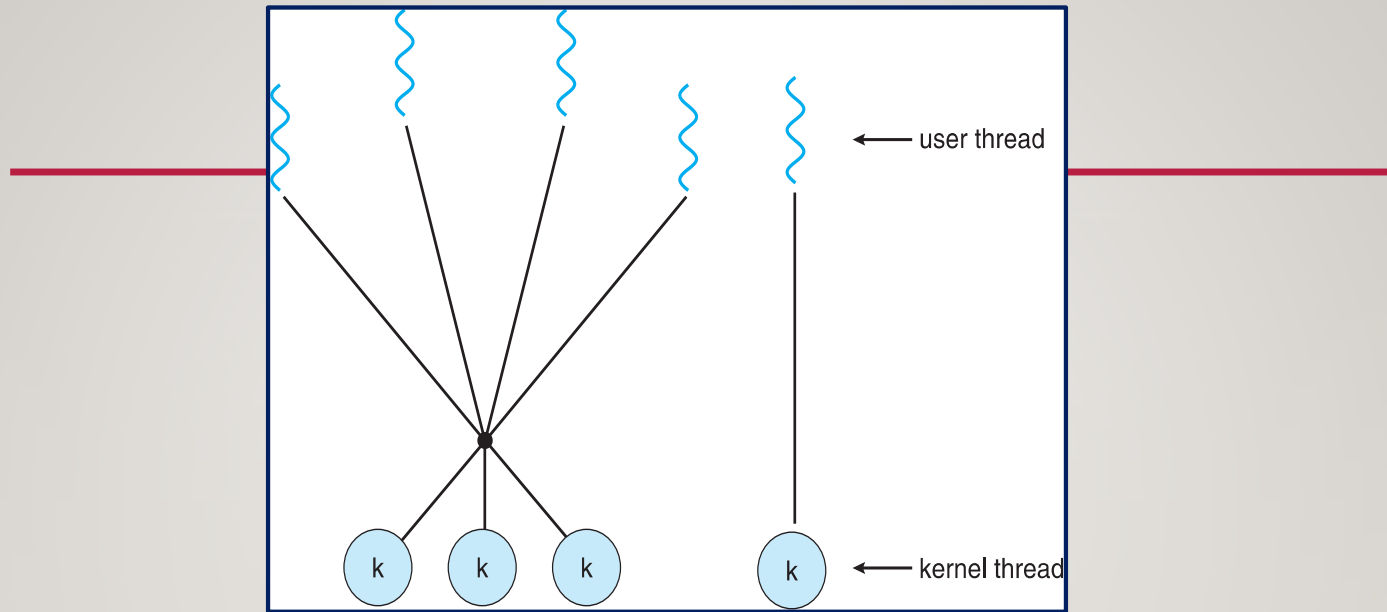
The many-to-many multithreading model is implemented in:

→ Solaris prior to version 9

→ Windows with `ThreadFiber` package

MULTITHREADING MODELS (5) – TWO-LEVEL

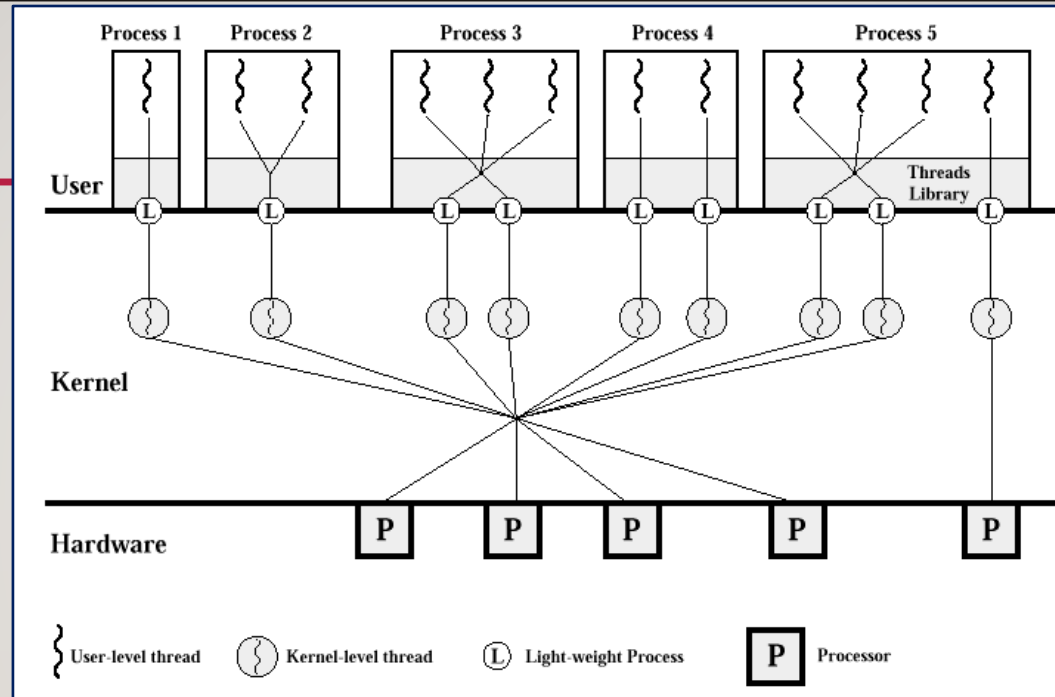
The following figure depicts the two-level multithreading model:



Similar to many-to-many multithreading model In addition, a ULT may be separately bound to a KLT.

MULTITHREADING MODELS (6) – EXAMPLE

The following figure depicts the ULT/KLT approach in Solaris 9:



ULTs are created in the user space using the threads library. Therefore, they are invisible to the OS.

In addition, synchronization and scheduling of threads are also performed in the user space.

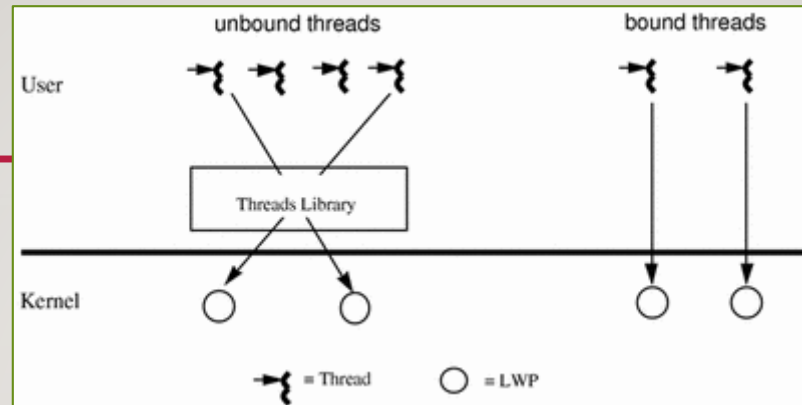
The kernel thread is the unit that can be dispatched to the CPU.

The programmer may adjust the number of KLTs.

Each lightweight process (LWP) supports one or more ULTs and maps to exactly one KLT.

MULTITHREADING MODELS (7) – LIGHTWEIGHT PROCESSES (LWP)

Threads libraries use underlying threads of control known as **lightweight processes (LWPs)**.



LWPs are thought of as virtual CPUs that execute code or system calls.

However, LWPs are transparent to the programmer: (s)he does not bother himself about them.

LWPs bridge the ULT and KLT.

The creation of a thread does not include a creation of a LWP.

Each **lightweight process (LWP)** is a kernel resource in the kernel pool. It is allocated and deallocated on a thread basis. This happens when a thread is created and destroyed.