CS 370: OPERATING SYSTEMS

[THREADS]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

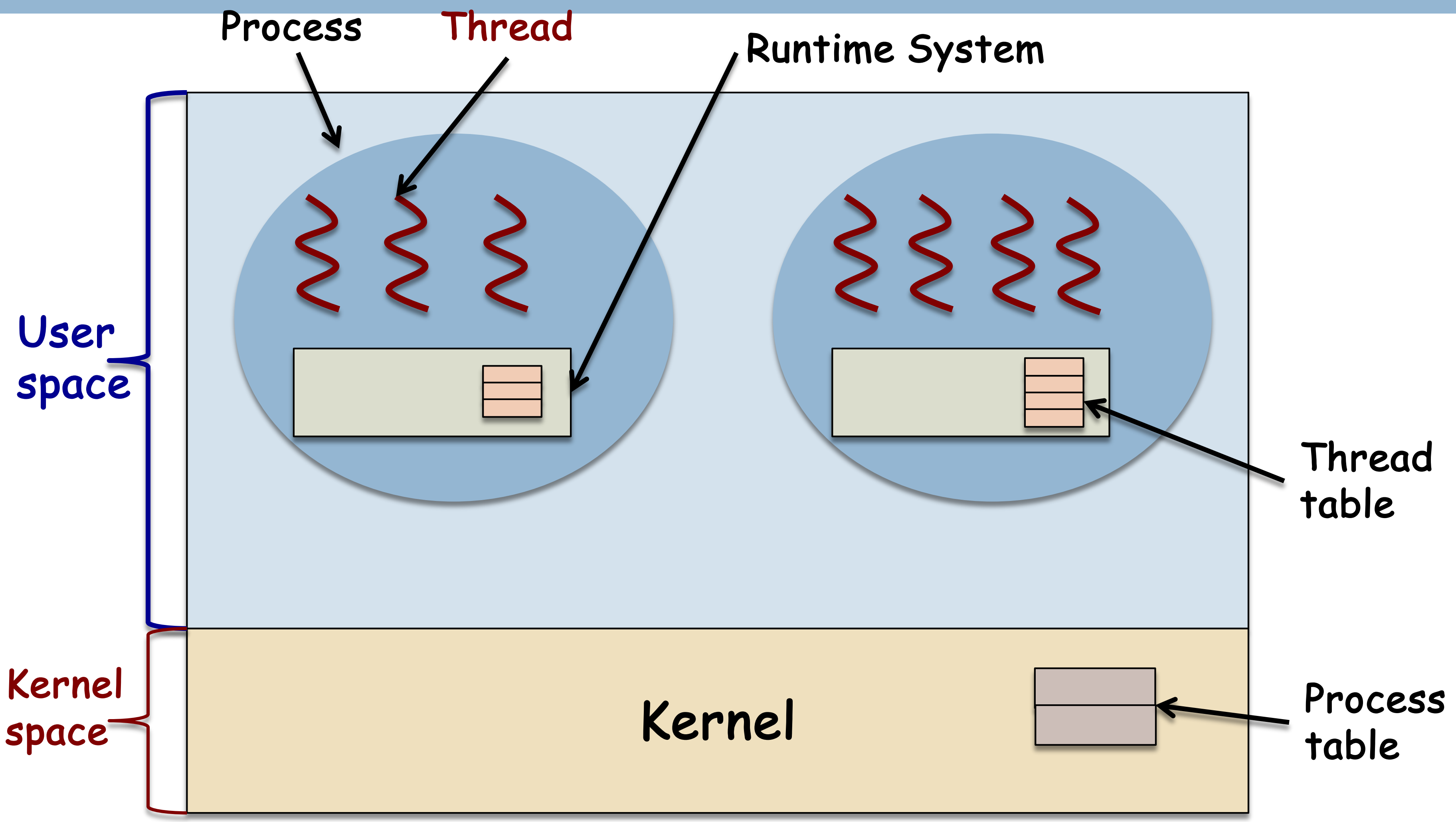
** Lecture slides created by: SHRIDEEP PALLICKARA

Topics covered in this lecture

- User- and kernel-level threads
- Thread Models
- Thread Libraries

User-level Threads

User-level threads: Overview



User threads are invisible to the kernel and have low overhead

- **Compete among themselves** for resources allocated to their encapsulating process
- Scheduled by a *thread runtime* system that is **part** of the process code
- Programs link to a special library
  - Each library function is enclosed by a **jacket**
  - Jacket function calls thread runtime to do thread management
    - Before (and possibly after) calling jacketed library function.

User level thread libraries: Managing blocking calls

- **Replace** potentially blocking calls with non-blocking ones
- If a call does not block, the runtime invokes it
- If the call *may block*
  1. Place thread on a list of *waiting* threads
  2. Add call to list of actions to *try later*
  3. Pick another thread to run
- ALL control is **invisible** to user and OS

Disadvantages of the user level threads model (1)

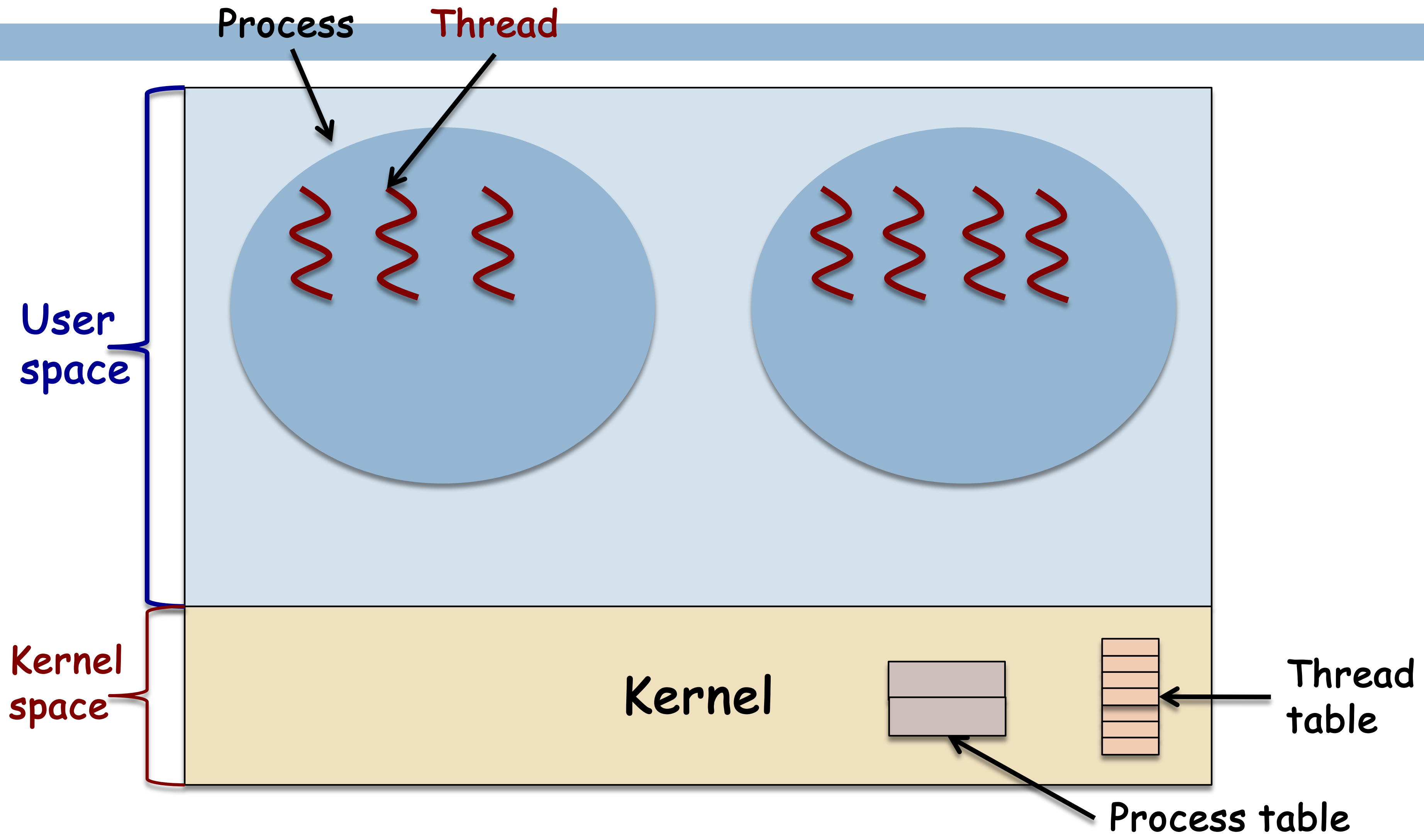
- Assumes that the runtime will **eventually regain** control, this is thwarted by:
  - CPU bound threads
  - Thread that *rarely* perform library calls …
    - Runtime can't regain control to schedule other threads
- Programmer must avoid **lockout** situations
  - Force CPU-bound thread to *yield* control

Disadvantages of the user level threads model (2)

- Can only share processor resources allocated to encapsulating process
  - **Limits** available parallelism

Kernel Threads

Kernel-level threads: Overview



Kernel threads

- Kernel is aware of kernel-level threads as **schedulable entities**
  - Kernel maintains a thread table to keep track of all threads in the system
- **Compete systemwide** for processor resources
  - Can take advantage of multiple processors

Kernel threads: Management costs

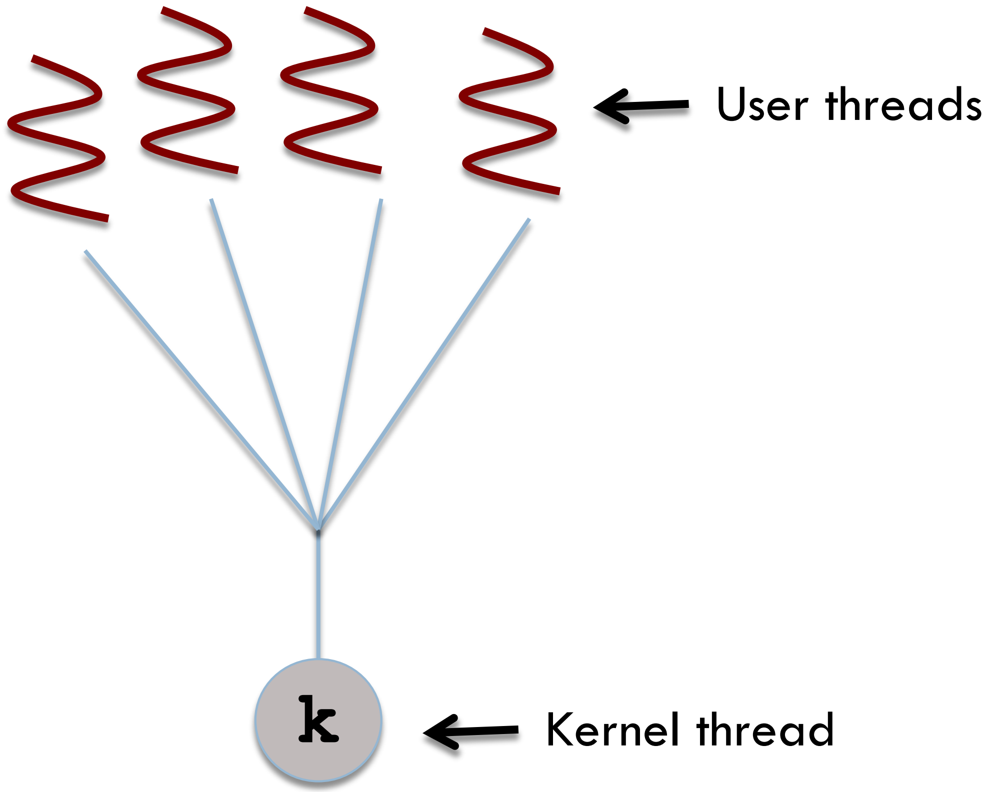
- Scheduling is **almost as expensive** as processes
  - Synchronization and data sharing *less expensive* than processes
- More expensive to manage than user-level threads

Hybrid thread models

- Write programs in terms of user-level threads
- Specify number of schedulable entities associated with process
  - **Mapping at runtime** to achieve parallelism
- Level of user-control over mapping
  - Implementation dependent

THREAD MODELS

The Many-to-One threading model



Many-to-One Model maps many user level threads to 1 kernel thread

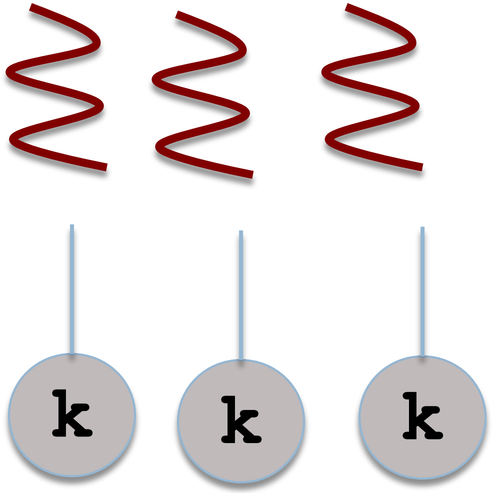
- Thread management done by thread library in **user-space**
- What happens when one thread makes a *blocking system call*?
  - The entire process blocks!

Many-to-One Model maps many user level threads to 1 kernel thread

- Only 1 thread can access kernel at a time
  - Multiple threads **unable** to run in parallel on multi-processor/core system

- E.g.: Solaris Green threads, GNU Portable threads

The One-to-One threading model



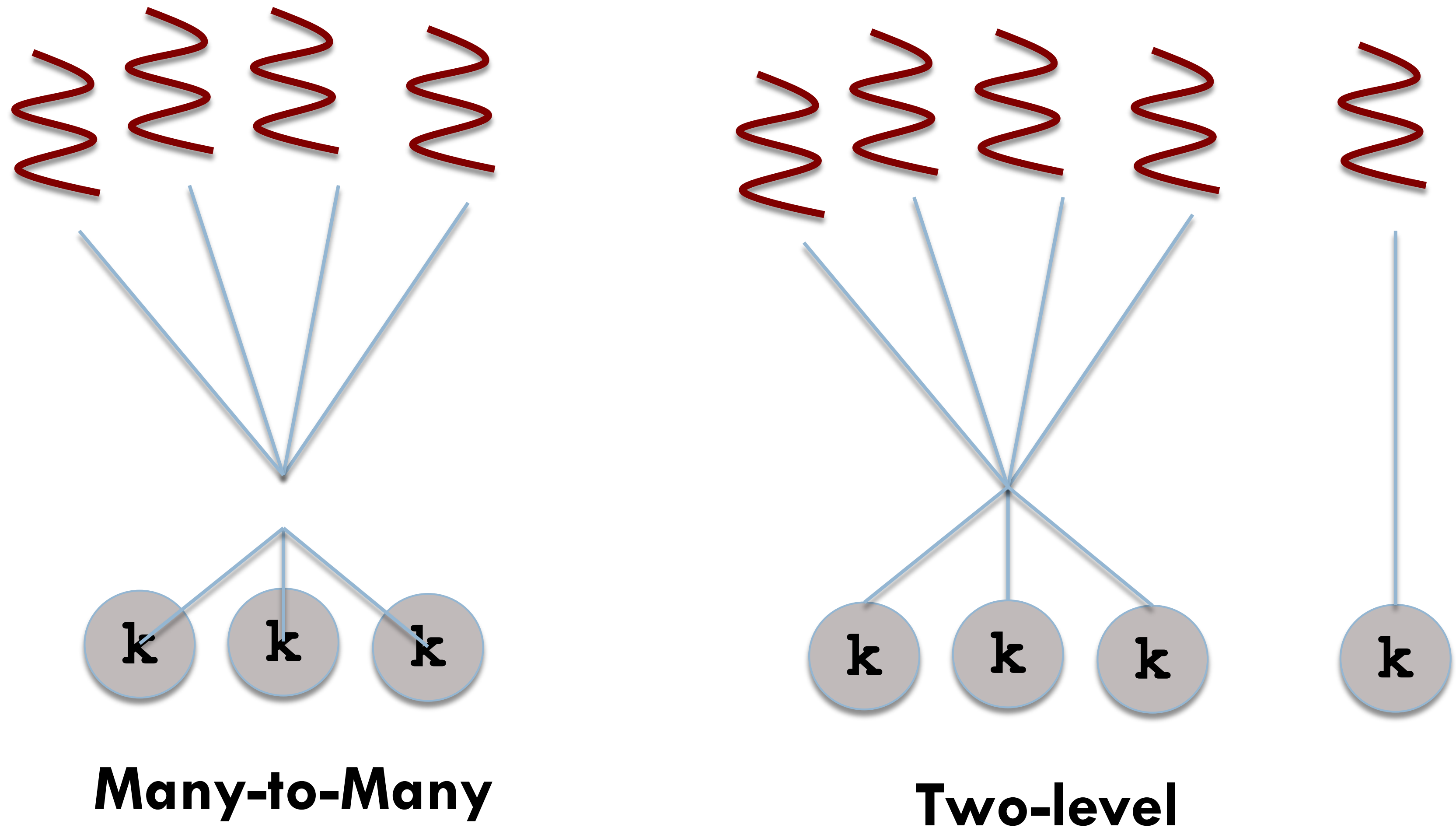
One-to-One Model: Maps each user thread to a kernel thread

- More **concurrency**
  - Another thread can continue to run, when a thread invokes a blocking system call
- Threads run in **parallel** on multiprocessors

One-to-One Model: Maps each user thread to a kernel thread

- Disadvantages:
  - There is an **overhead** for kernel thread creation
    - Multiple user threads can degrade application performance
  - Uses more kernel threads so uses more resources
- Supported by:
  - Linux
  - Windows family: NT/XP/2000
  - Solaris 9 and up

Many-to-Many threading Model: 2-level is a variant of this



Many-to-Many model

- **Multiplex** many user-level threads on a smaller number of kernel threads
- Number of kernel threads may be specific to
  - Particular application
  - Particular machine
- Supported in
  - IRIX, HP-US, and Solaris (prior to version 9)

A comparison of the three models

	Many-to-one	One-to-One	Many-to-Many
Kernel Concurrency	NO	YES if many threads	YES
During blocking system call?	Process Blocks	Process DOES NOT block if other threads	Process DOES NOT block
Kernel thread creation	Kernel thread already exists	Kernel thread creation overhead	Kernel threads available
Caveat	Use system calls (blocking) with care	Don't create too many threads to not use too much resources	

***Provide an API** for creating and managing threads*

Thread Libraries

Thread libraries provide an API for managing threads

- Includes functions for :
  1. Thread creation and destruction
  2. Enforcement of mutual exclusion
  3. Conditional waiting

- Runtime system to manage threads
  - Users are **not aware** of this

User level thread libraries

- **Uses kernel support for mapping user threads to kernel ones**
- Library code & data structures reside in user space
- Invoking a library function **does not** result in a system call
  - Local function call in user space

Kernel level thread libraries

- Library code & data structures in kernel space
- Invoking library function typically **results in a system call**
- Typical approach: user-level thread libraries accesses kernel-level thread library API to map user threads to kernel threads, but this is hidden in the user thread library runtime implementation

Thread libraries provide an API for creating and managing threads

	User level library	Kernel level library
Library code and data structures	Reside in user space	Reside in kernel space
Thread creation requires a system call?	NO	YES
OS/Kernel support	NO	YES

Dominant thread libraries (1)

- POSIX pthreads
  - Extends POSIX standard (IEEE 1003.1c)
  - Provided as user- or kernel-level library
  - Solaris, Mac OS X, Linux, …
- Win32 thread library
  - Kernel-level library

Dominant thread libraries (2)

- Java threading API
  - Implemented using **thread library on host system**
    - On Windows: Threads use Win32 API
    - UNIX/Linux: Uses pthreads

POSIX THREADS

This is a specification for thread *behavior*,
not an *implementation*

POSIX thread management functions: Return 0 if successful

POSIX function	Description
pthread_cancel	Terminate another thread
pthread_create	Create a thread
pthread_detach	Set thread to release resources
pthread_exit	Exit a thread without exiting process
pthread_kill	Send a signal to a thread
pthread_join	Wait for a thread
pthread_self	Find out own thread ID

Functions return a non-ZERO error code
Do NOT set `errno`

POSIX: Thread creation
`pthread_create()`

- Automatically makes the thread runnable

- Takes 3 parameters:
  1. Points to **ID** of newly created thread
  2. **Attributes** for the thread
     - Stack size, scheduling information, etc.
  3. Pointer to **function** that the thread calls when it begins execution

POSIX: Detaching and Joining

- When a thread exits it does not release its resources
  - Unless it is a **detached thread**
- `pthread_detach()`
  - Sets *internal options* to specify that storage for thread can be **reclaimed** when it exits
  - 1 parameter: Thread ID of the thread to detach

POSIX: Thread joins

- Threads that are not detached are `joinable`
- Undetached threads don't release resources until
  - Another thread calls `pthread_join` for them
  - Process exits
- `pthread_join`
  - Takes ID of the thread to **wait** for
  - **Suspends** calling thread till target terminates
  - Similar to `waitpid` at the process level
  - `pthread_join(pthread_self())?`
    - **Deadlock!**

POSIX: Exiting and cancellation

- If a function running calls `exit`, **all** threads terminate
- Call to `pthread_exit` causes only the calling thread to terminate
- Threads can force other threads to return through a *cancellation* mechanism
  - `pthread_cancel`: takes thread ID of target
  - Depends on *type* and *state* of thread

More info on pthread_cancel

- State: `pthread_setcancelstate` to change state
  - `PTHREAD_CANCEL_ENABLE`
  - `PTHREAD_CANCEL_DISABLE`
    - Cancellation requests are held pending
- Cancellation type allows thread to control when to exit
  - `PTHREAD_CANCEL_ASYNCHRONOUS`
    - Any time
  - `PTHREAD_CANCEL_DEFFERED`
    - Only at specified cancellation points

Pthreads example

- We will use a thread to perform summation of a non-negative integer

$$sum = \sum_{i=0}^{N} i$$

- If N=5, we compute the sum of 0 through 5
  - 0 + 1 + 2 + 3 + 4 + 5 = 15

Using Pthreads (1)

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

int sum; /* this data is shared by the thread(s) */

void *runner(void *param); /* the function to compute
                              sum */
```

Using Pthreads (2)

```
int main(int argc, char *argv[]){

   pthread_t tid;    pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);

   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);

   /* now wait for the thread to exit */
   pthread_join(tid, NULL);

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

Using Pthreads (3)

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

   if (upper > 0) {
      for (i = 1; i <= upper; i++)
         sum += i;
   }

   pthread_exit(0);
}
```

Java Threads

Harnesses the thread model of the **host** OS

Java

- Designed from the ground-up to support **concurrent** programming
  - Basic concurrency support in the language and class libraries
- Java 1.5 and higher
  - Powerful high-level concurrency APIs

JVMs harness the thread models of the host OS

- Windows XP has a one-to-one model
  - So a thread maps to a kernel thread
- Tru64 UNIX uses the many-to-many model
  - Java threads mapped accordingly
- Solaris
  - Initially, used Green Threads → many-to-one
  - Version 9 onwards: one-to-one model

Creating Threads in Java

① Create a new class **derived** from `Thread`

- Override its `run()` method

② More commonly used, `Runnable` interface

- Has 1 method `run()`
- Create new `Thread` class by passing a `Runnable` object to its constructor

③ The `Executor` interface (`java.util.concurrent`)

- Has 1 method `execute()`

Java Threads: Interrupts

- Invoke `interrupt()` on the Thread
- Threads must support their **own** interruption
- An **interruptible thread** needs to
  1. Catch the `InterruptedException`
     - Methods such as `sleep()` throw this, and are designed to cancel the operation and return
  2. Periodically invoke `Thread.interrupted()` to see if it has been interrupted

Java Threads: `joins`

- If thread object `threadA` is currently executing
- Another thread can call `threadA.join()`
  - Causes current thread to pause execution **until `threadA` terminates**
- Variants of `join()`
  - Specify a waiting period

Java threads example

- We will be performing the same summation operation that we did for pThreads

Using Java Threads (1)

```
class Sum {
    private int sum;

    public int get() {
        return sum;
    }

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

Using Java Threads (2)

```java
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.set(sum);
    }
}
```

Using Java Threads (3)

```
public class Driver {
   public static void main(String[] args) {

      Sum sumObject = new Sum();
      int upper = Integer.parseInt(args[0]);

      Thread worker = new Thread(new Summation(upper,
sumObject));
      worker.start();
      try {
         worker.join();
      } catch (InterruptedException ie) {
         ie.printStacktrace()
      }
      System.out.println("The sum of " + upper + " is " +
                         sumObject.get());
   }
}
```

Win32 Threads

- `CreateThread`
  - Security Information, size of stack, flag (start in suspended state?)
- `WaitForSingleObject`
- `CloseHandle`

Thread Pools

Thread Pools

① **Create** a number of threads at start-up

② **Place** them into a pool

③ These threads **sit and wait** for work

- ADVANTAGES:
  - Servicing requests is *faster* with existing threads
  - *Limits* total number of threads

Thread Pools: When work needs to be performed

① **Awaken** a thread from this pool

② **Assign** it work

③ Once the worker thread completes, it **returns itself** to the pool

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 4]
- *Andrew S Tanenbaum and Herbert Bos. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620.* [Chapter 2].
- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2.* [Chapter 12]