Chapter 4: Threads

A fundamental unit of CPU utilization

Thanks to the author of the textbook [**SGG**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.
Turgay Korkmaz

Chapter 4: Threads

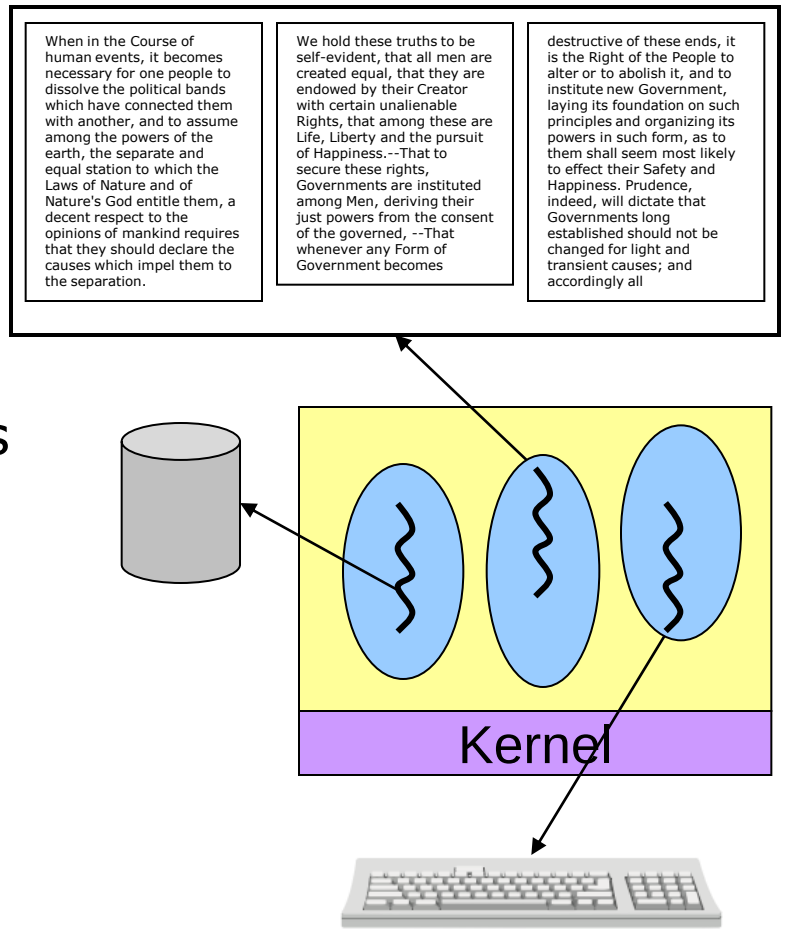
- Overview *
- Multithreading Models *****
- Thread Libraries ****
  - **Pthreads** and **Java thread**
- Threading Issues ***
- Operating System Examples *
- Windows XP Threads *
- Linux Threads ***

Objectives

- To introduce the notion of a thread
  - a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To examine issues related to multithreaded programming
- To discuss the APIs for the **Pthreads** and **Java thread** libraries (optional Win32)

Example: A Multi-Activity Text Editor

- **Process approach** on data
  - P1: read from keyboard
  - P2: format document
  - P3: write to disk



The processes will *actively* access the **same set of data.**

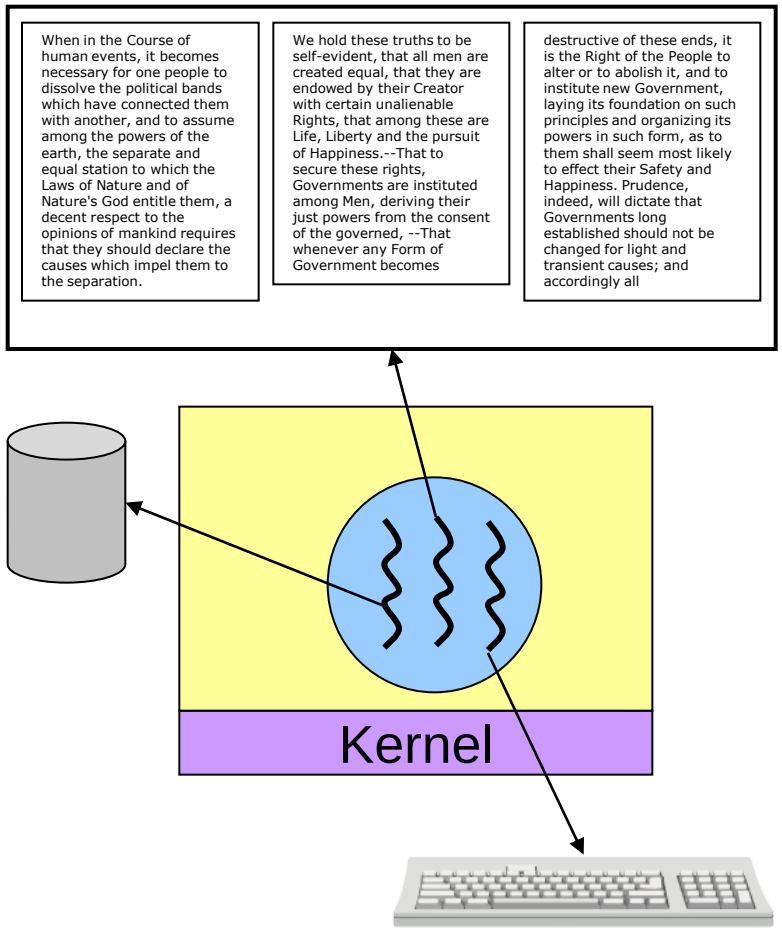
How do the* processes *exchange data?

Context Switch for Processes- **costly**

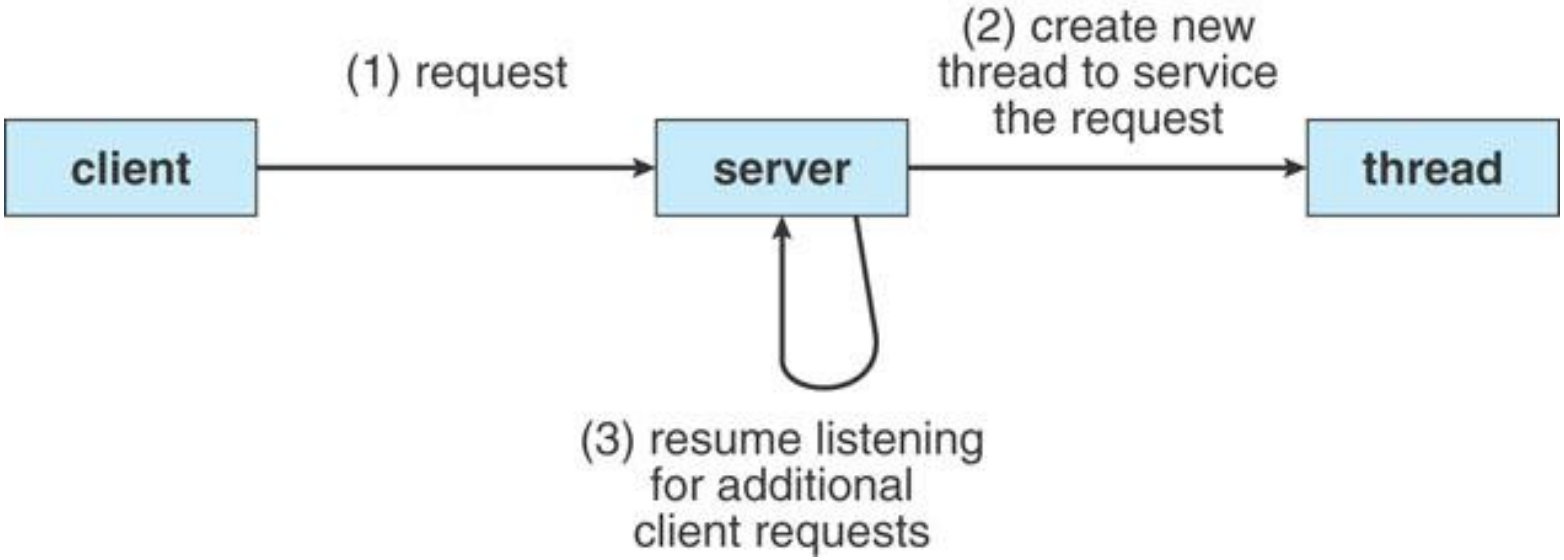
Ideal Solution for the Text Editor

Threads

- Three *activities* within one process
  - Single address space
  - Same execution environment
  - Data shared easily
- Switch between activities
  - Only *running context*
  - ***No change in address space***



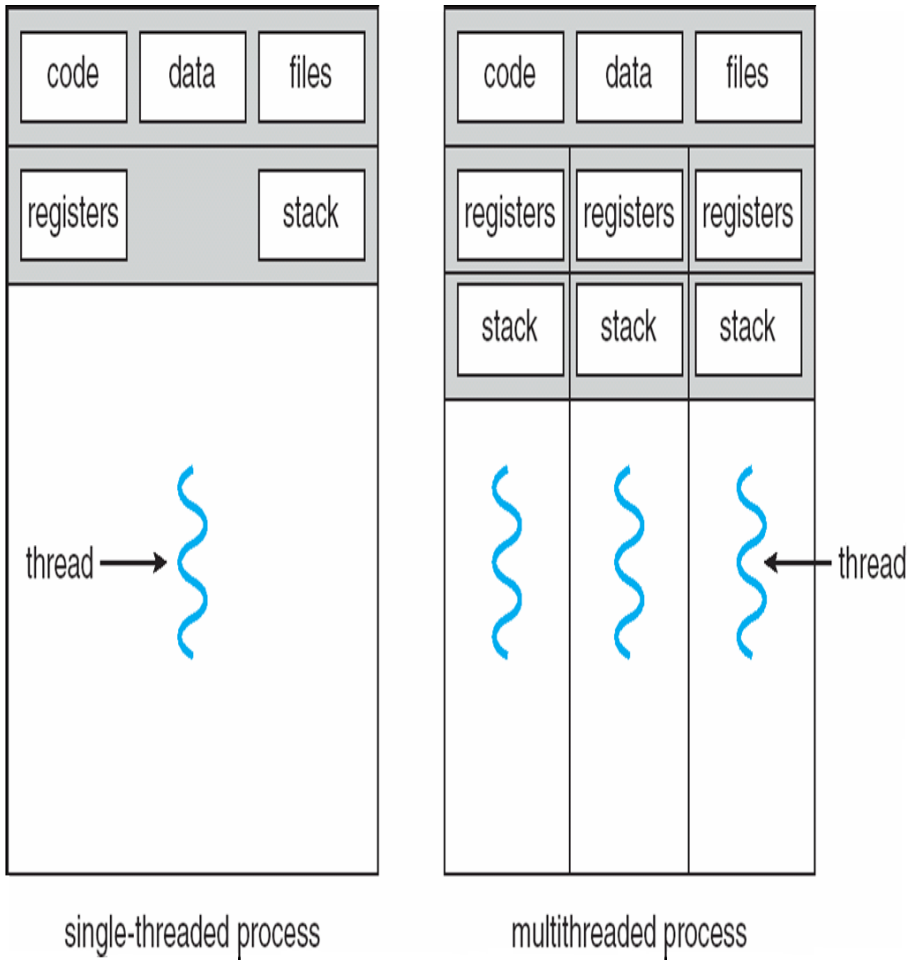
Another Example: Web servers



Thread vs. Process

Both are methods for Concurrency and Parallelism

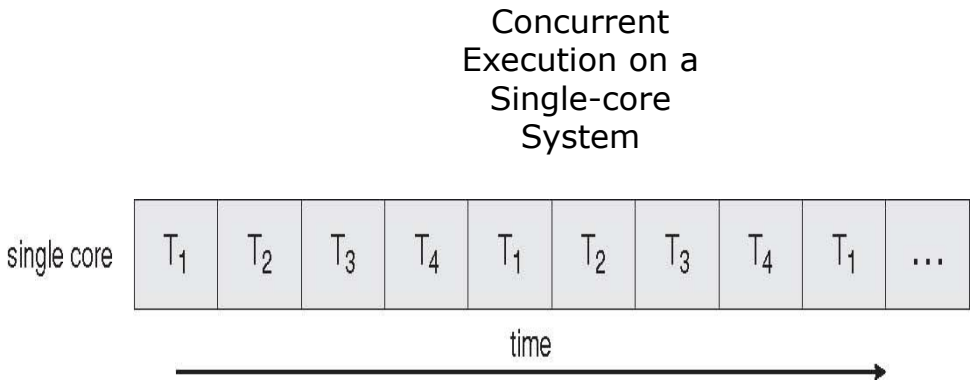
- Processes
  - Independent execution units use their own states, address spaces, and interact with each other via IPC
  - Traditional Processes have single flow of control (thread)
- Thread
  - Flow of control (activity) within a process
  - A single process on a modern OS may have *multiple* threads
  - All threads share the code, data, and files while having some separated resources
  - Threads directly interact with each other using shared resources

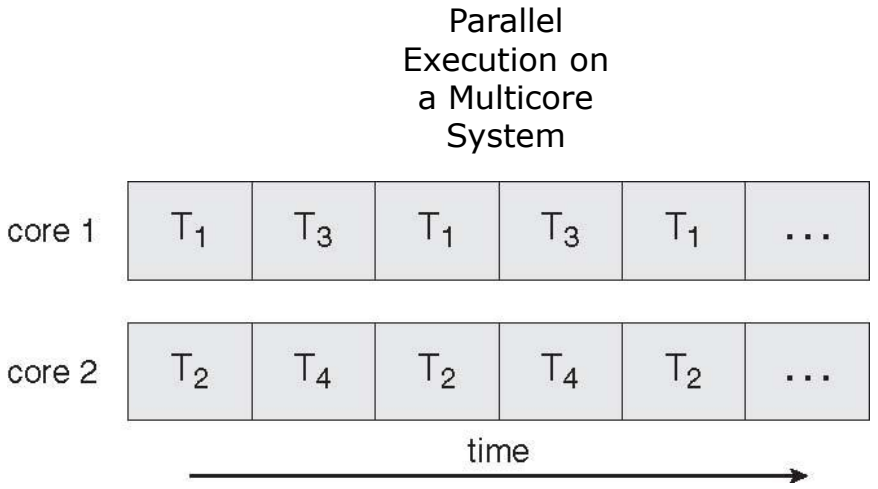


Concurrency vs. Parallelism?

Benefits

- **Responsiveness**
  - Interactive application
- **Resource sharing**
  - Address space and other resources
- **Economy: less overhead**
  - Solaris: process creation 30X overhead than thread;
  - Context switching threads within a process 5X faster
- **Scalability**
  - Better utilization of multiprocessor/multicore systems





Multicore Programming

- Multithreaded programming provides a mechanism for efficient use of multicore systems
- Programmers face new challenges
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- Multicore programming will require entirely new approach to design SW

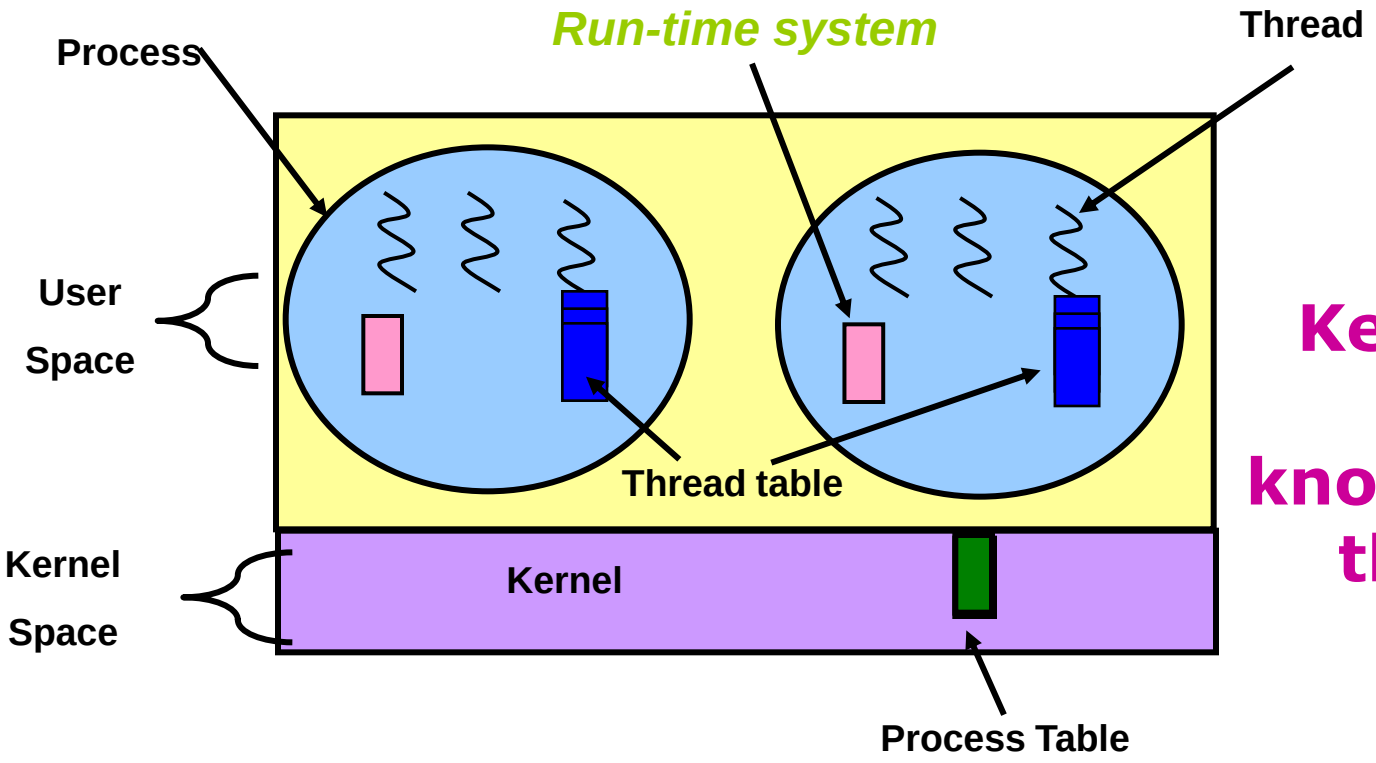
MULTI-THREADING MODELS

Thread Implementations: Issues

- Process usually starts with a single thread and creates others
- Thread management operations (similar to process management)
  - **Creation***:* procedure/method for the new thread to run
  - **Scheduling**: runtime properties/attributes
  - **Destruction**: release resources
  - Thread Synchronization
    - join, wait, etc.
- ***Who*** manages threads and ***where***
  - **User space**: managed by applications using some libraries
  - **Kernel space**: managed by OS
    - all modern OSes have kernel level support (more efficient)

User-Level Threads

- User threads: ***thread library*** at the user level
- **Run-time system** provides support for thread creation, scheduling and management



Kernel has *NO* knowledge of threads!

User-Level Threads (cont.)

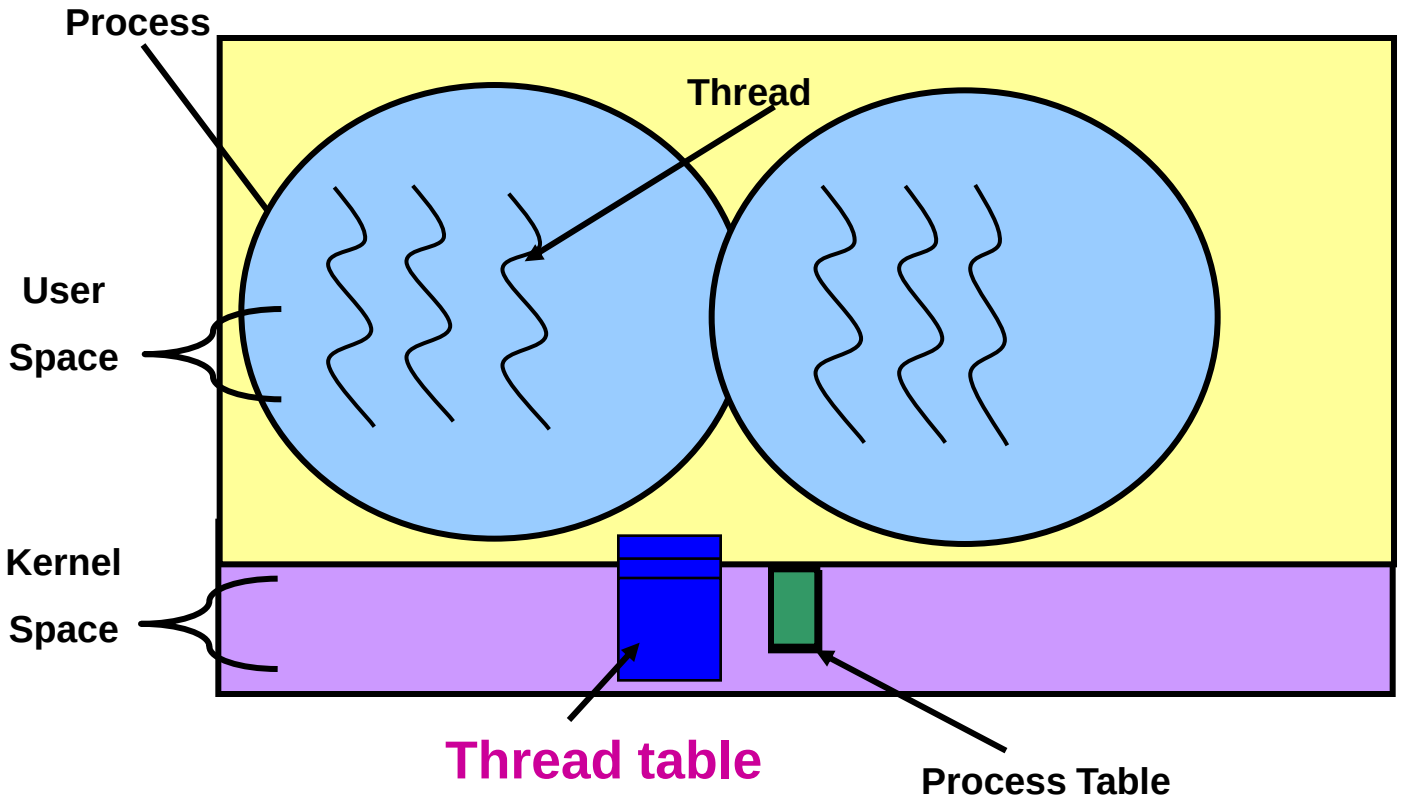
- Each process needs its own **private *thread table*** to keep track of the threads in that process.
- The thread-table keeps track of the per-thread items (program counter, stack pointer, register, state..)
- When a thread does something that *may* cause it to become blocked ***locally*** (e.g. wait for another thread), it calls a **run-time system** procedure.
- If the thread must be put into blocked state, the procedure performs ***thread switching***

- Advantages
  - **Fast** thread switching: no kernel activity involved
  - Customized/**flexible** thread scheduling algorithm
  - Application **portability**: different machines with library
- Problems/disadvantages:
  - Kernel only knows process
  - **Blocking** system calls: kernel blocks process, so one thread blocks all activities (many-to-one mapping)
  - All threads share **one CPU,** so cannot use multi-proc./core

Kernel-Level Threads

Supported directly by OS

Kernel performs thread creation, scheduling & management in kernel space



Kernel-Level Threads (cont.)

- Advantages
  - User activity/thread with blocking I/O does NOT block other activities/threads from the same user
  - When a thread blocks, the kernel may choose another thread from the **same** or **different** process
  - Multi-activities in applications can use multi-proc/cores
- Problems/disadvantages
  - Thread management could be relatively costly:
    - all methods that might block a thread are implemented as system calls
  - Non-flexible scheduling policy
  - Non-portability: application can only run on same type of machine

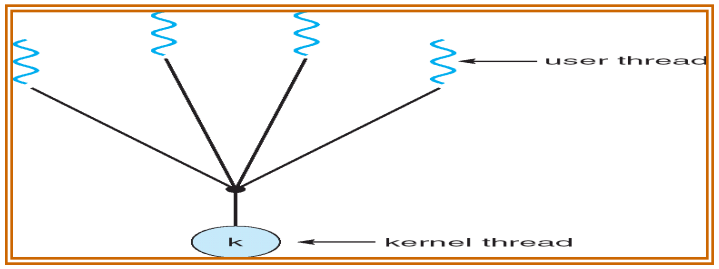
What is the relationship between **user** level and **kernel** level threads?

How to map **user** level threads to **kernel** level threads?

Mapping: User → Kernel Threads

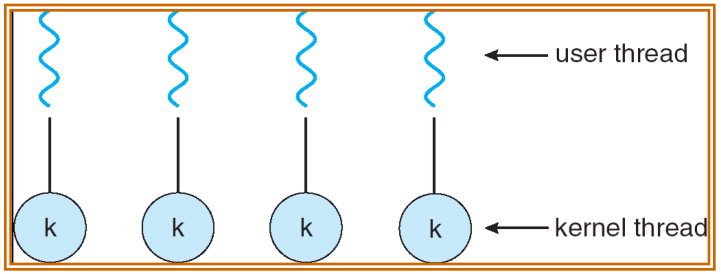
- **Many-to-one**
  - Many user threads → one kernel thread (-/+ are same as in user-level threads)

    Examples: Solaris Green Threads, GNU Portable Threads



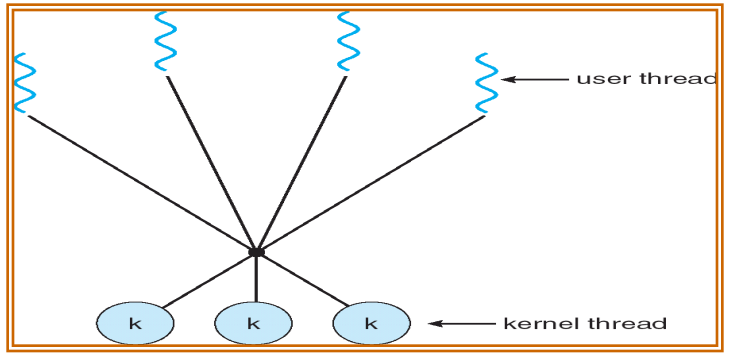
- **One-to-One**
  - One user thread → one kernel thread;
  - + more concurrency
  - - limited number of kernel threads

  Examples: Windows NT/XP/2000, Linux, Solaris 9 and later



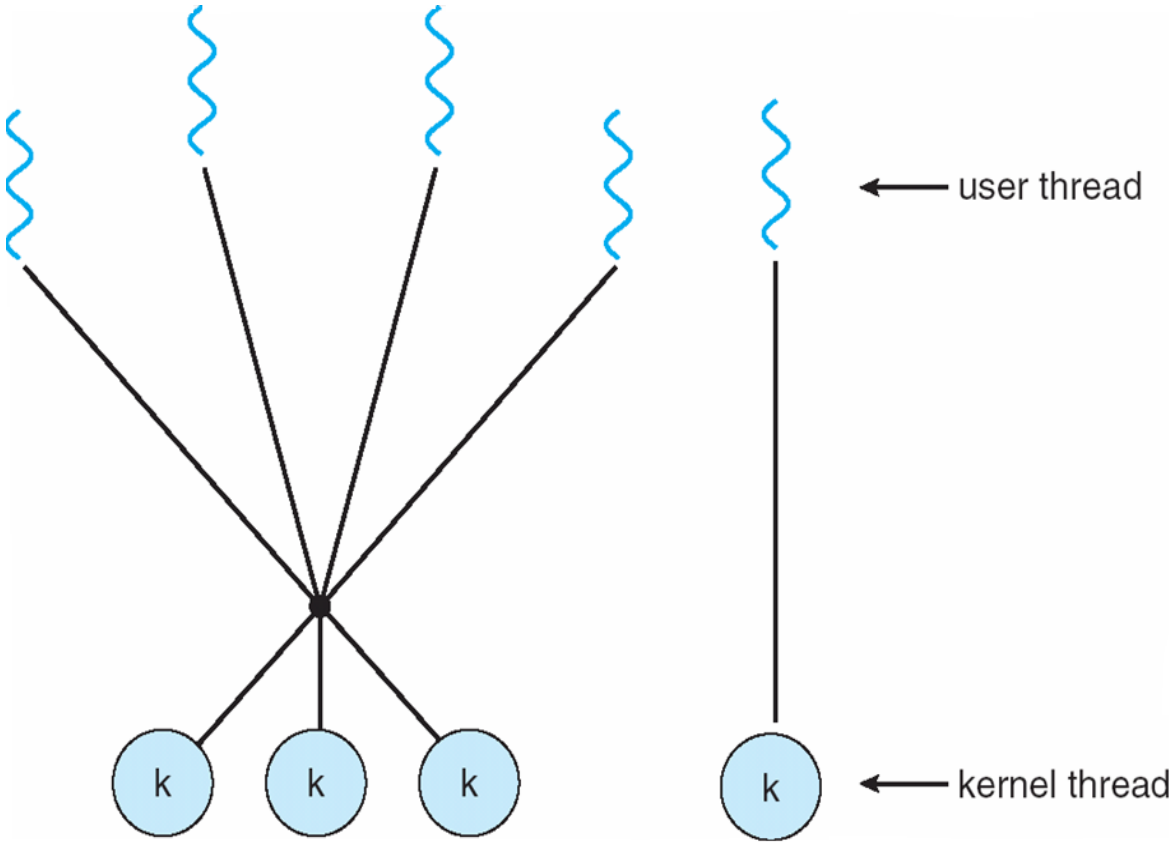
- **Many-to-Many**
  - Many user threads → many kernel threads
  - + no limit on the number of user threads
  - - not true concurrency because kernel has limited number of threads

    Examples: Solaris prior to version 9, Windows NT/2000 with the ThreadFiber package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

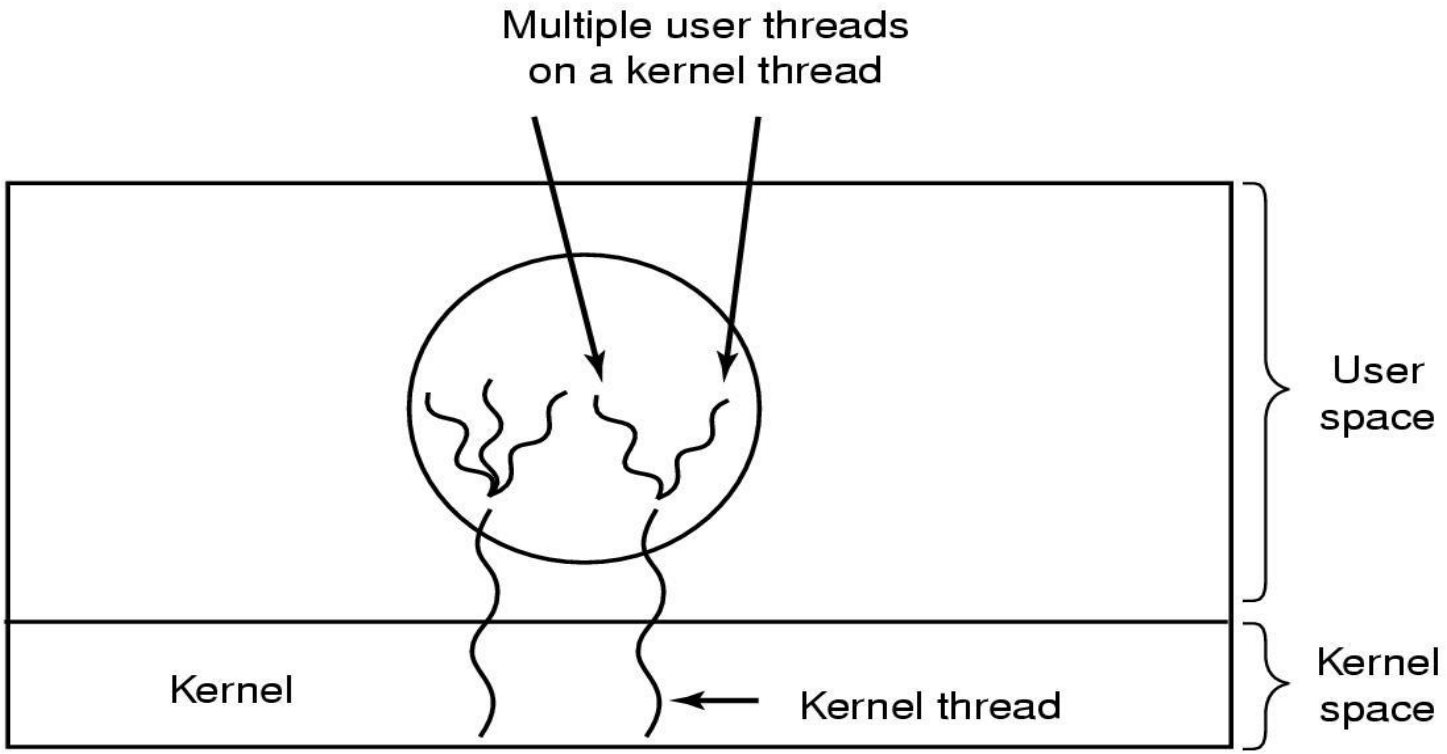


*

Hybrid Implementation

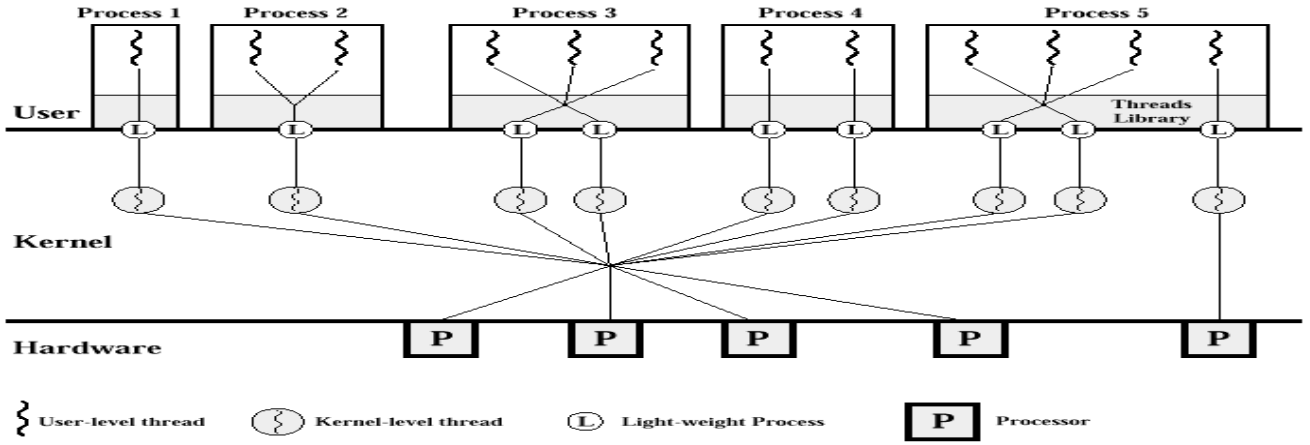
combine the best of both approaches

- Use kernel-level threads, and then multiplex user-level threads onto some or all of the kernel threads.



Light-Weight Process (LWP)

- **Lightweight process (LWP):** intermediate data structure
  - For user-level threads, LWP is a **Virtual processor**
  - **Each LWP attaches to a kernel thread**
- Multiple user-level threads → a single LWP
  - Normally from the same process
- **A process may be assigned multiple LWPs**
  - Typically, an LWP for each blocking system call
- OS schedules kernel threads (hence, LWPs) on the CPU



LWP: Advantages and Disadvantages

- + User level threads are easy to create/destroy/sync
- + A blocking call will not suspend the process if we have enough LWP
- + Application does not need to know about LWP
- +LWP can be executed on different CPUs, hiding multiprocessing
- Occasionally, we still need to create/destroy LWP (as expensive as kernel threads)
- Makes up calls (scheduler activation)
  - + simplifies LWP management
  - - Violates the layered structure

Provide programmers with API for creating and managing threads

THREAD LIBRARIES

Thread Libraries

- Two primary ways of implementing
  - **User-level** library
    - Entirely in user space
    - Everything is done using function calls (no system calls)
  - **Kernel-level** library supported by the OS
    - Code and data structures for kernels are in kernel space
    - Function calls result in system calls to kernel
- Three primary thread libraries:
  - POSIX Threads **Pthread** *(either a user-level or kernel-level)*
  - Win32 threads *(kernel-level)*
  - Java threads *(JVM manages threads by using host system threads)*
    - *Threads are fundamental model of prog exec,*
    - *Java provides rich set of features for thread creation and mng.*

POSIX Threads: Pthread

- POSIX
  - **P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**]
  - Standardized programming interface
- **Pthread**
  - 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 (Defined as a set of C types and procedure calls)
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Implementations
  - User-level vs. kernel-level

https://computing.llnl.gov/tutorials/pthreads/

Pthread APIs: Four Groups

- **Thread management**
  - Routines for creating, detaching, joining, etc.
  - Routines for setting/querying thread attributes
- **Mutexes**: abbreviation for "mutual exclusion"
  - Routines for creating, destroying, locking/unlocking
  - Functions to set or modify attributes with mutexes.
- **Conditional variables**
  - Communications for threads that share a mutex
  - Functions to create, destroy, wait and signal based on specified variable values
  - Functions to set/query condition variable attributes
- **Synchronization**
  - Routines that manage read/write locks and barriers

Chapter 6

Thread Call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Thread Creation

pthread_t threadID;

*pthread_create (&threadID, *attr, methodName, *para);*

- 1st argument is the ID of the new thread
- 2nd argument is a pointer to pthread_attr_t
- 3rd argument is **thread (function/method) name**
- 4th argument is a pointer to the arguments for the thread's method/function

Thread: Join and Exit

- Join with a non-detached thread by using

  *pthread_join (pthread_t thread, void **status)*

  (All threads are created non-detached by default, so they are “joinable” by default)

- Exit from threads:
  - If threads use *exit( ),* process terminates.
  - A thread (main, or another thread) can exit by calling *pthread_exit( ),* this does not terminate the process.

- More information about Pthread programming

  https://computing.llnl.gov/tutorials/pthreads/

.

An Example: *testthread.c*

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS        5

void *PrintHello(void *threadid){
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}
int main(int argc, char *argv[]){
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
         }
    }
   // for(t=0;t<NUM_THREADS;t++)
   //    pthread_join(threads[t], NULL);  // wait for other threads
   pthread_exit(NULL);    //to return value;
}
```

To compile, link with the pthread library.

> gcc testthread.c -o test –lpthread

```
> test
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

Threads are fundamental model of program execution in Java. So, Java provides a rich set of features for thread creation and management

JAVA THREADS

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

How to Create Threads in Java (1)

There are two ways in Java:

1. Create a class `MyTh` that directly extends `Thread` class
   - The code in `MyTh.run()` will be the new thread
   - Then in a driver program
     - `MyTh th = new MyTh( … );`
     - `th.start();`
   - Not recommended (why?)
     - A bad habit for industrial strength development
     - The methods of the worker class and the Thread class get all tangled up
     - Makes it hard to migrate to Thread Pools and other more efficient approaches

```
public class MyTh
                extends Thread {
  public MyTh(…){
  …
  }
  public void run() {
   //overwrite this …
  }
}
```

```
public class Thread {
…
public String getName();
public void interrupt();
public boolean isAlive();
public void join();
public void setDaemon(boolean on);
public void setName(String name);
public void setPriority(int level);
public static Thread currentThread();
public static void sleep(long ms);
public static void yield();
}
```

How to Create Threads in Java (2)

2. Define a class `MyTh` that implements `Runnable` interface
   - The code in `MyTh.run()` will be the new thread
   - Then in a driver program
     - `Thread th = new Tread( new MyTh(…) );`
     - `th.start();`

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

```
public class MyTh
        implements Runnable {
  public MyTh(…){
    …
  }
  public void run() {
   //overwrite this …
  }
}
```

Example 1: Extend Thread Class

```
public class SimpleThread extends Thread {
  String msg;
  int repetition;
  public SimpleTread(String msg, int r){
      this.msg = msg;
      this.repetition = r;
  }
  public void run() {
     //overwrite run method
      for (int i = 0; i < repetition; i++)
         System.out.println("[" + i + "]" + msg);
  }
}
```

```
public class SimpleThreadMain {
  public static void main(String[] args) {
      SimpleThread t1 = new SimpleThread("T1", 100);
      t1.start();

      SimpleThread t2 =  new SimpleThread("T2", 100);
      t2.start();
  }
}
```

Example 2: Implement Runnable interface

```
public class SimpleRunnable implements Runnable {
   String msg;
   int repetition;
   public SimpleRunnable(String msg, int r) {
      this.msg = msg;
      this.repetition = r;
   }
   public void run( ) {
      //overwrite run method
      for (int i = 0; i < repetition; i++)
         System.out.println("[" + i + "]" + msg);
   }
}
```

```
[0]T1
[0]T2
[1]T1
[1]T2
[2]T1
[2]T2
[3]T1
[3]T2
[4]T1
...
```

```
public class SimpleRunnableMain {
  public static void main(String[] args) {
      SimpleRunnable r1 = new SimpleRunnable("T1", 100);
      Thread t1 = new Thread(r1);
      t1.start( );

      SimpleRunnable r2 = new SimpleRunnable("T2", 100);
      Thread t2 = new Thread(r2);
      t2.start( );

   }
}
```

Use *join()* to wait for a thread to finish

http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

The `join` method of `Thread` throws `InterruptedException` and must be placed in a `try-catch`.

```
…
try {
    t1.join( );
    t1.join( );
} catch (InterruptedException e) {}

System.out.println("Both are done ");
```

```
for (int i = 0; i < 100; i++)
    System.out.println("main");
```

```
main
[0]T1
[0]T2
[1]T1
main
[1]T2
[2]T1
[2]T2
[3]T1
main
[3]T2
[4]T1
……
Both are done
```

```
[0]T1
[0]T2
[1]T1
[1]T2
[2]T1
[2]T2
[3]T1
[3]T2
[4]T1
……
main
main
main
……
Both are done
```

Java Thread Example - Output

```
public class ThreadExample implements Runnable {
   public void run() {
       for (int i = 0; i < 3; i++)
             System.out.println(i);
   }
   public static void main(String[] args) {
       new Thread( new ThreadExample()).start();
       new Thread( new ThreadExample()).start();
       System.out.println("Done");
   }
}
```

What are the possible outputs?

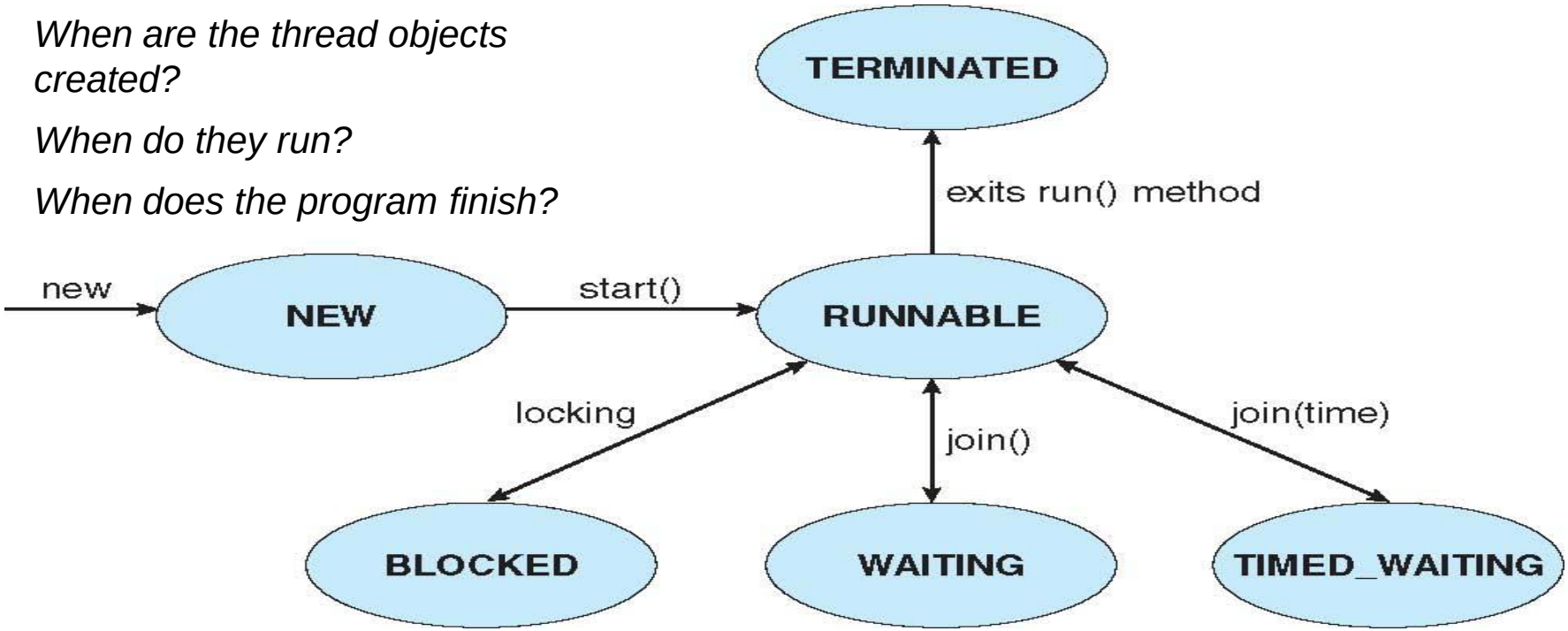
0,1,2,0,1,2,Done	// thread 1, thread 2, main()
0,1,2,Done,0,1,2	// thread 1, main(), thread 2
Done,0,1,2,0,1,2	// main(), thread 1, thread 2
0,0,1,1,2,Done,2	// main() & threads interleaved

Why doesn't the program quit as soon as "Done" is printed?

JVM shuts down when all non-daemon threads terminate!

Life-Time of Java Threads

- *When are the thread objects created?*
- *When do they run?*
- *When does the program finish?*



- *A thread object exists when it is constructed, but it doesn't start running until the start method is called.*
- *A thread completes (or dies) when its run method finishes or when it throws an exception. The object representing this thread can still be accessed.*

Transitions between Threads

- Transitions between states caused by
  - Invoking methods in class Thread
    - start(), yield(), sleep(), wait(), join()
    - The `join, wait`, and `sleep` methods of `Thread` throw `InterruptedException` and must be placed in a `try-catch`.
- Other (external) events
  - Scheduler, I/O, returning from run()…
- Scheduler (ch5)
  - Determines which runnable threads to run
  - Part of OS or Java Virtual Machine (JVM)
  - Many computers can run multiple threads simultaneously (or nearly so)

Another Example: Java Threads

Define a class that implements **Runnable** interface

```
class MutableInteger
{
  private int value;
  public int getValue() {
    return value;
  }
  public void setValue(int value) {
    this.value = value;
  }
}

class Summation implements Runnable
{
  private int upper;
  private MutableInteger sumValue;
  public Summation(int upper, MutableInteger sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }
  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
      sum += i;
    sumValue.setValue(sum);
  }
}
```

```
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 {
        // create the object to be shared
        MutableInteger sum = new MutableInteger();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sum));
        thrd.start();
        try {
          thrd.join();
          System.out.println
                ("The sum of "+upper+" is "+sum.getValue());
        } catch (InterruptedException ie) { }
      }
    }
    else
      System.err.println("Usage: Summation <integer value>");
  }
}
```

Another Example: Producer-Consumer

Define a class that implements **Runnable** interface

```
import java.util.Date;

class Producer implements Runnable
{
   private Channel<Date> queue;

   public Producer(Channel<Date> queue) {
      this.queue = queue;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // produce an item and enter it into the buffer
         message = new Date();
         System.out.println("Producer produced " + message);
         queue.send(message);
      }
   }
}
```

```
import java.util.Date;

public class Factory
{
    public static void main(String args[]) {
       // create the message queue
       Channel<Date> queue = new MessageQueue<Date>();

       // Create the producer and consumer threads and pass
       // each thread a reference to the MessageQueue object.
       Thread producer = new Thread(new Producer(queue));
       Thread consumer = new Thread(new Consumer(queue));

       // start the threads
       producer.start();
       consumer.start();
    }

}
```

```
         // consume an item from the buffer
         message = queue.receive();

         if (message != null)
            System.out.println("Consumer consumed " + message);
      }
   }
}
```

Semantics of **fork()** and **exec()** system calls

Thread cancellation of target thread

Signal handling

Thread pools

Thread-specific data

Scheduler activations

THREADING ISSUES

Semantics of fork() and exec()

- What will happen if one thread in a process call **fork( )** to create a new process?
  - Does **fork()** duplicate only the calling thread or all threads?
  - How many threads in the new process?

- Duplicate only the invoking thread
  - exec(): will load another program
  - Everything will be replaced anyway

- Duplicate all threads
  - If exec() is not the next step after forking
  - What about threads performed blocking system call?!

Thread Cancellation

- Terminating a thread before it has finished
  - Examples
    - Threads search in parallel of database: one finds → others stop
    - Stop fetching web contents (images)
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
    - - Thread resources and data consistency
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - + Wait for self cleanup → **cancellation safety points**

Signal Handling

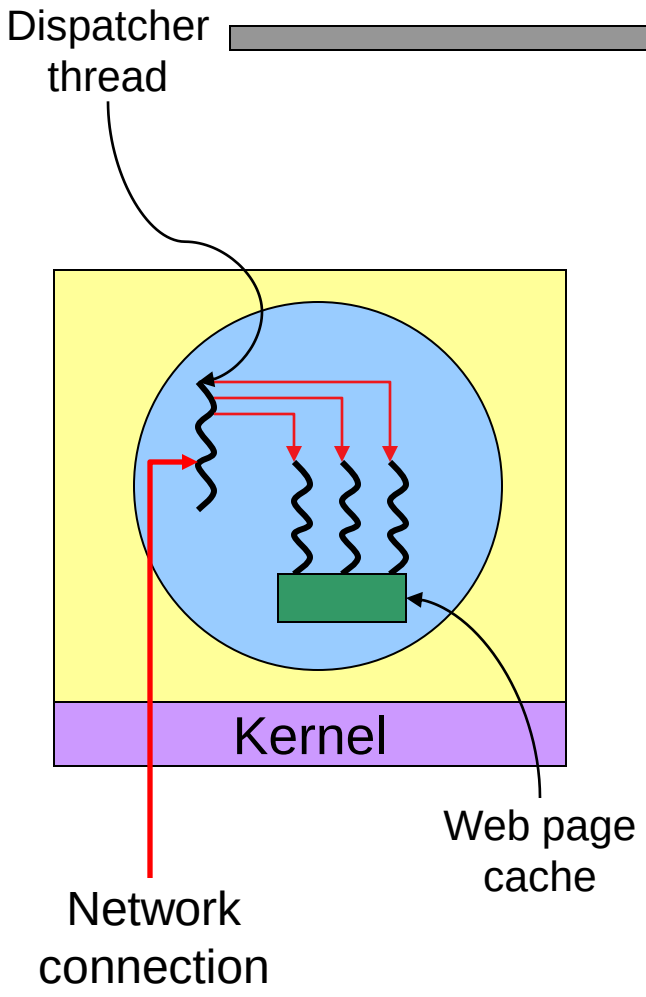
1. Signal is generated by particular event
2. Signal is delivered to a process
3. Signal is handled

- Signals are used in UNIX systems to notify a **process** that a particular event has occurred.
  - Depending on the source, we can classify them as
    - Synchronously [Running prog generates it] (e.g., div by 0, memory access)
    - Asynchronously [External src generates it] (e.g., ready of I/O or Ctrl+C)
- Which threads to notify?
  - All threads (Ctrl-C)
  - Single thread to which the signal applies (illegal memory, div by 0)
  - Subset of threads: thread set what it wants (**mask**)
  - **Thread handler: kernel default** or **user-defined**
- Unix allows threads to specify which one to block or accept
- Windows has no support for signals but it can be emulated

Thread Pool

- Recall web server example,
  - We created a thread for every request
  - This is better than creating a process, but still time consuming
  - No limit is put on the number of threads
- Pool of threads
  - Create some number of treads at the startup
  - These threads will wait to work and put back into pool
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- Adjust thread number in pool
  - According to usage pattern and system load

Thread Pool Example: Web server




```
while(TRUE) {
  getNextRequest(&buf);
  handoffWork(&buf);

}
```

Worker thread

```
while(TRUE) {
  waitForWork(&buf);
  lookForPageInCache(&buf,&page);
  if(pageNotInCache(&page)) {
    readPageFromDisk(&buf,&page);
  }
  returnPage(&page);

}
```

Java thread pool example

```
class Task implements Runnable
{
  public void run() {
   System.out.println("I am working on a task.");
  }
}
```

```
import java.util.concurrent.*;
public class TPExample
{
  public static void main(String[] args) {
     int numTasks = Integer.parseInt(args[0].trim());

     // create the thread pool
     ExecutorService pool =
           Executors.newCachedThreadPool();

     // run each task using a thread in the pool
     for (int i = 0; i < 5; i++)
         pool.execute(new Task());

     // sleep for 5 seconds
    try { Thread.sleep(5000); }
    catch (InterruptedException ie) { }
    pool.shutdown();
  }
}
```

- Work queue
  - Fixed number of threads

Possible problems

- Deadlock
- Resource thrashing
- Thread leakage
- Overload

Thread Specific Data

- Allows each thread to have its own copy of data
- We may not want to share all data
- Thread libraries have support for this
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

- Both M:M and Two-level models require **communication** to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
  - Events to invoke upcall
    - A thread make a blocking system calls
    - A blocking system call complete returns
    - To ask user-level thread scheduler (runtime systems) to select the next runnable thread
- This communication allows an application to maintain the correct number of kernel threads

SKIP

Windows XP Threads

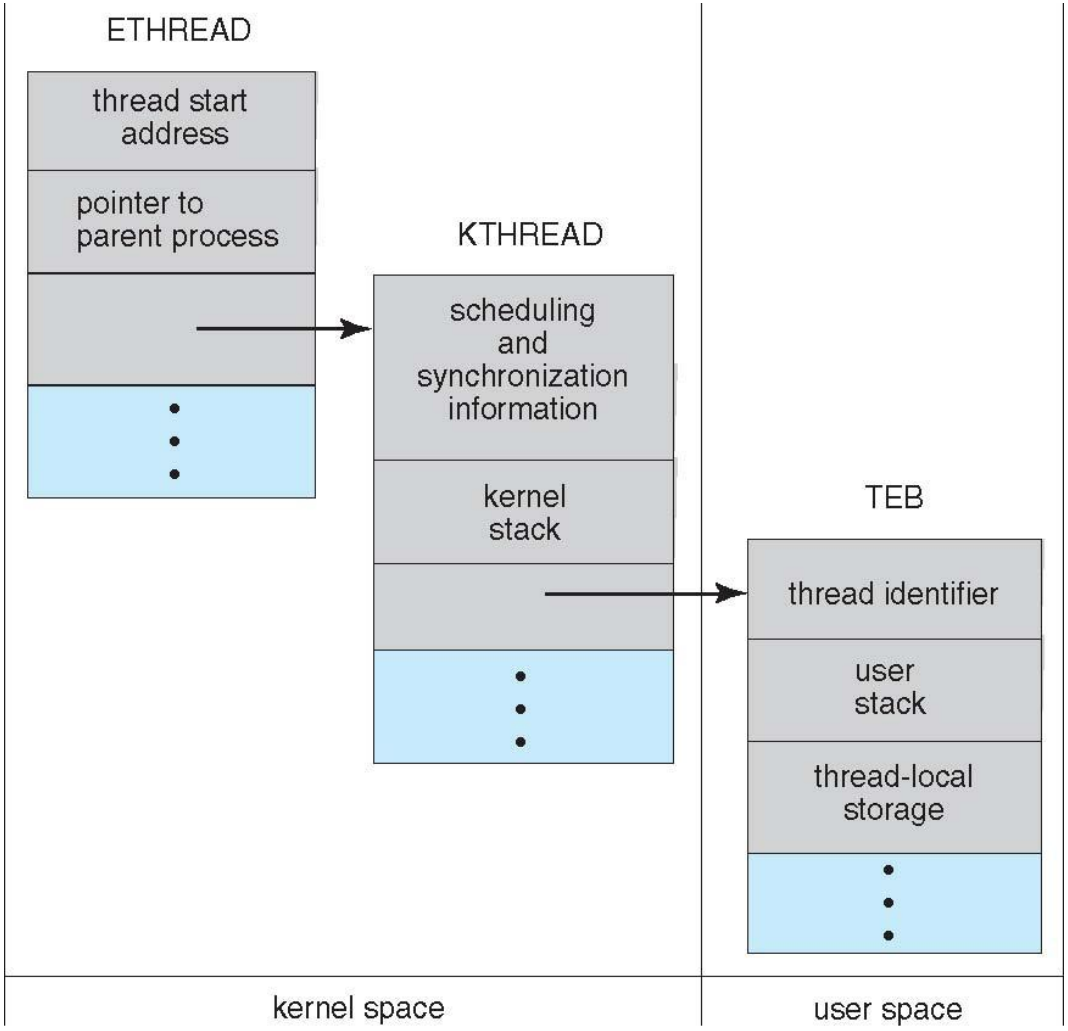
Linux Thread

OPERATING SYSTEM EXAMPLES

Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP Threads



*

Linux Threads

- Linux uses the term ***task*** (rather than process or thread) when referring to a flow of control

- Linux provides *clone()* system call to create threads
  - A set of flags, passed as arguments to the *clone()* system call determine how much sharing is involved (e.g. open files, memory space, etc.)

- Linux: 1-to-1 thread mapping
  - NPTL (Native POSIX Thread Library)

Linux Threads

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.

End of Chapter 4