# 4. Threads

ECE30021/ITP30002 Operating Systems

# Agenda

- Overview
- Multithreading models
- Thread libraries
- Threading issues
- Operating system examples

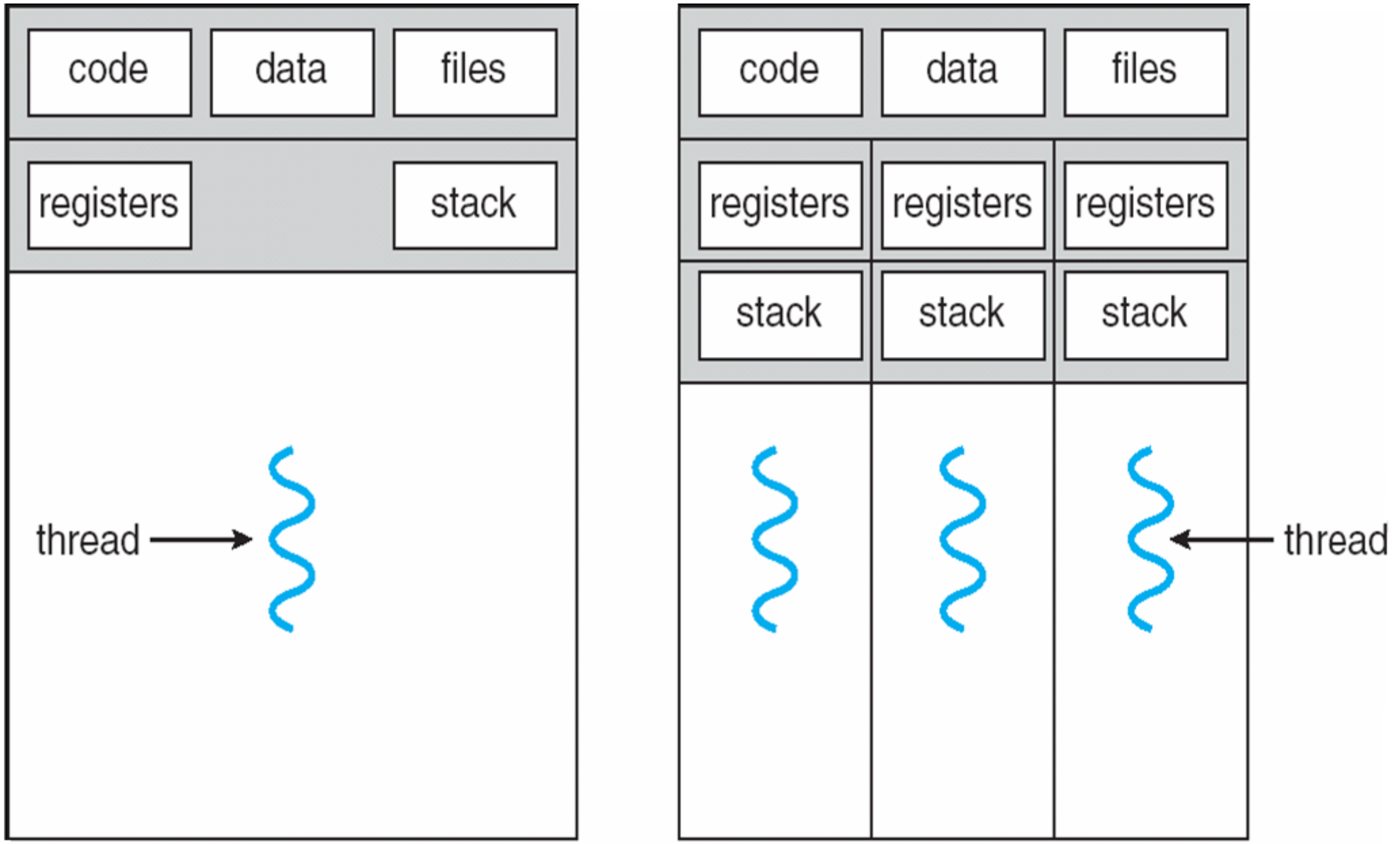
# Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display, fetch data, spell checking, answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

# Overview

- Process: program in execution
  - Each process occupies resources required for execution
- Thread: a way for a program to split itself into two or more simultaneously running tasks
  - Smaller unit than process
  - Threads in a process share resources
- A thread is comprised of
  - Thread ID, program counter, register set, stack, etc.

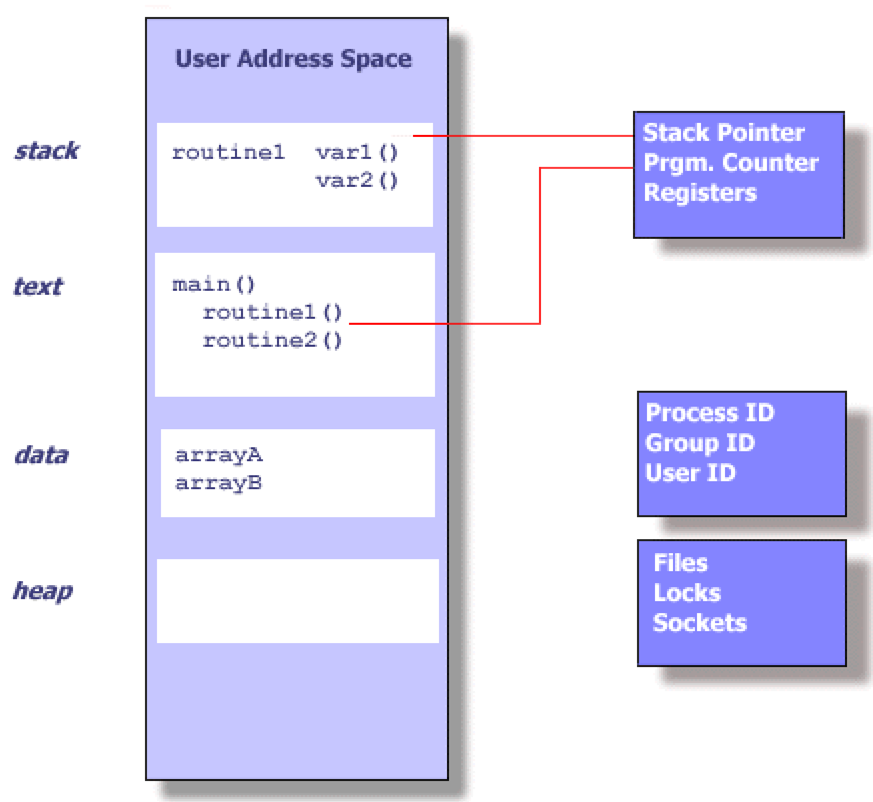
# Multithreaded Process



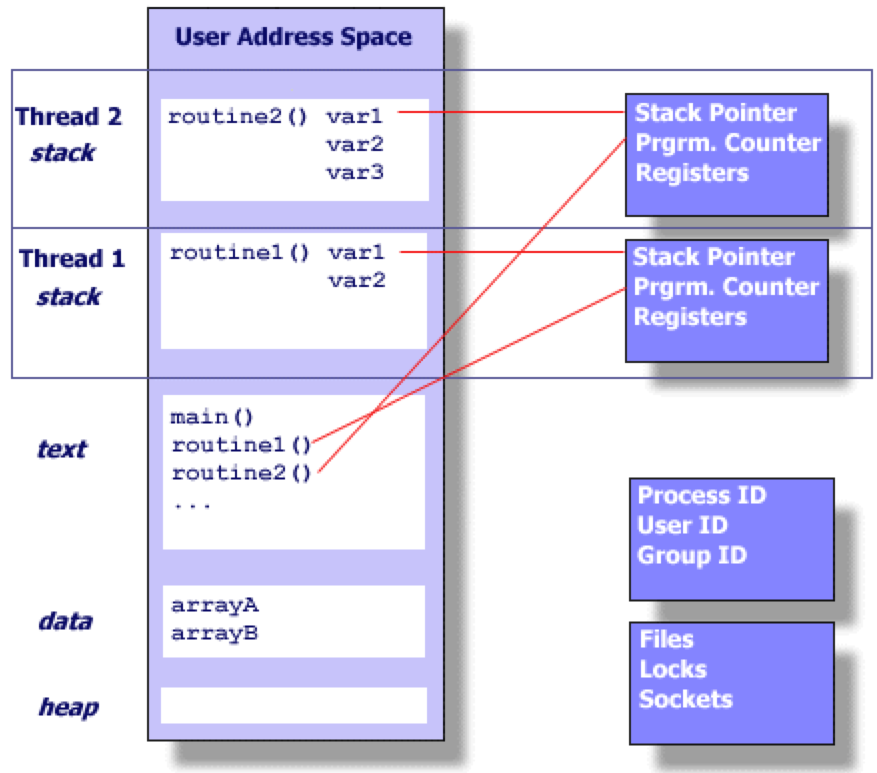

single-threaded process

multithreaded process

# Process vs. Thread



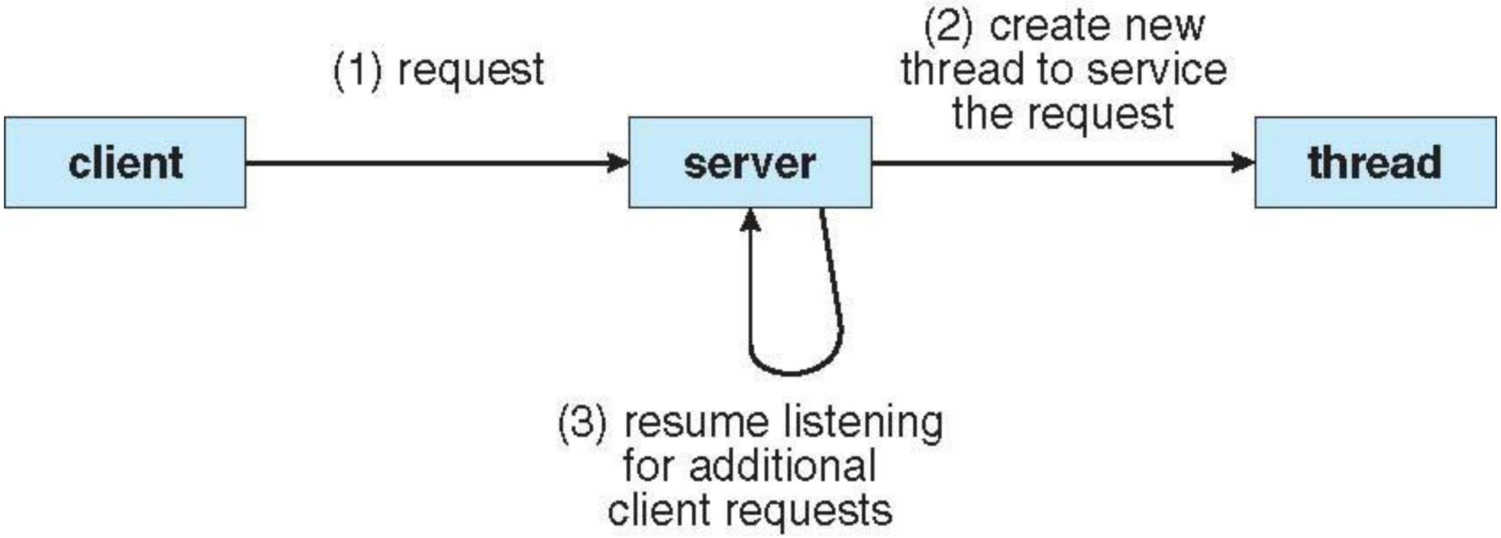

A process in UNIX




Threads within a Process

# Why Thread?

- Process creation is expensive in time and resource
  - Ex) Web server accepting thousands of requests



# Why Thread?

- Advantages of multithreaded programming
  - Responsiveness
  - Resource sharing
  - Economy
    - Creating process is about 30 times slower than creating thread
  - Scalability
    - Utilization of multiprocessor architectures

# Multicore Programming

- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

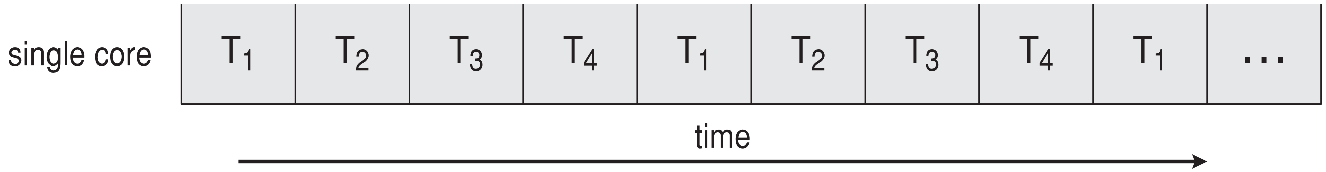
# Multicore Programming (Cont.)

- Types of parallelism
  - Data parallelism – distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as hardware threads
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

# Concurrency vs. Parallelism

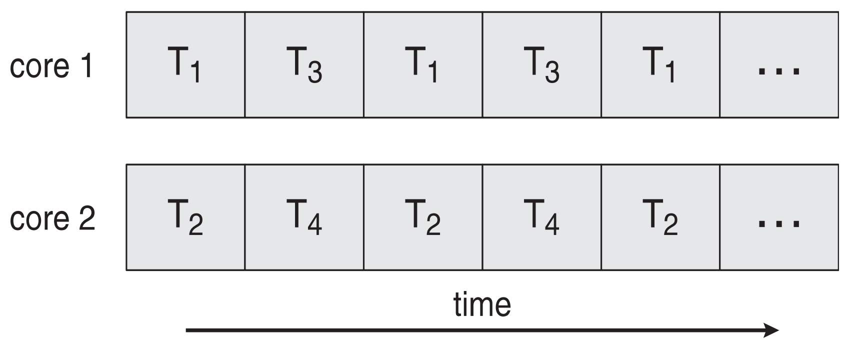
- Concurrency supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

  Ex) Concurrent execution on single-core system:




- Parallelism implies a system can perform more than one task simultaneously

  Ex) Parallelism on a multi-core system:



# Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
  - S is serial portion, N processing cores

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

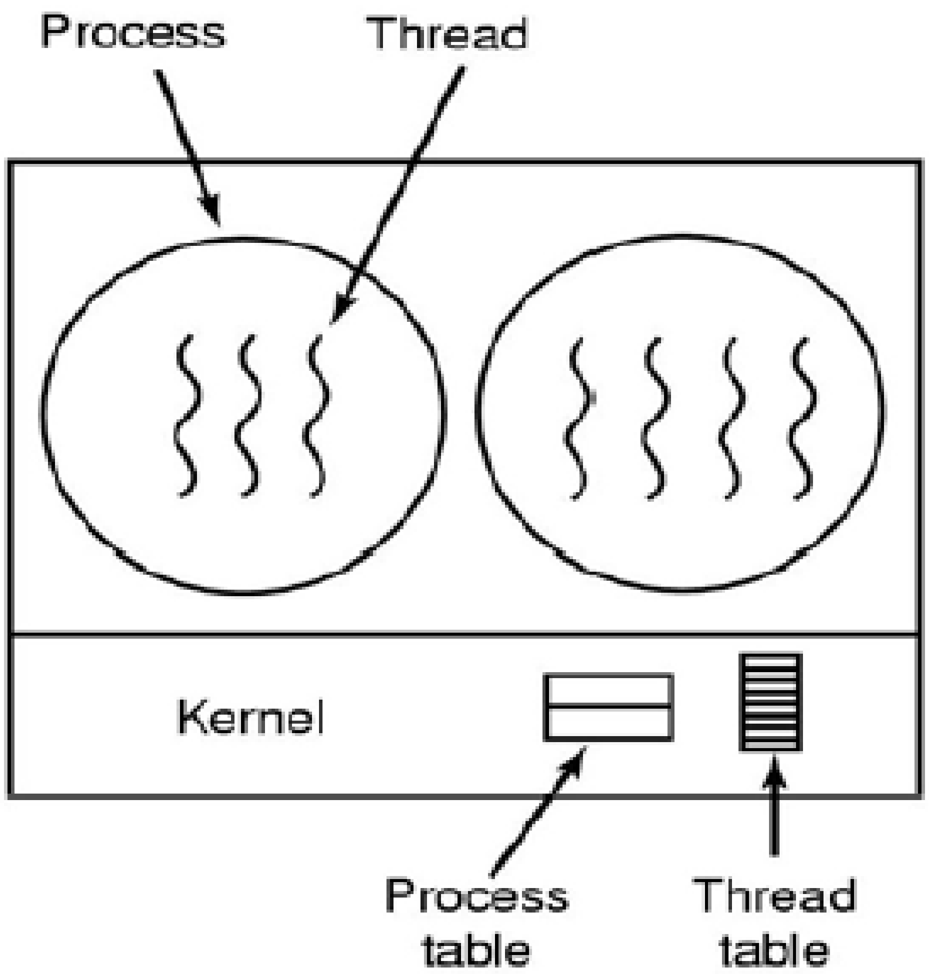
  Ex) Application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

  - As N approaches infinity, speedup approaches 1 / S
  - Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

# Types of Thread

- User thread: thread supported by thread library in user level
  - Created by library function call (not system call)
  - Kernel is not concerned in user thread.
  - Switching of user thread is faster than kernel thread.
- Kernel thread: thread supported by kernel
  - Created and managed by kernel
  - Scheduled by kernel
  - Cheaper than process
  - More expensive than user thread

# Kernel Thread

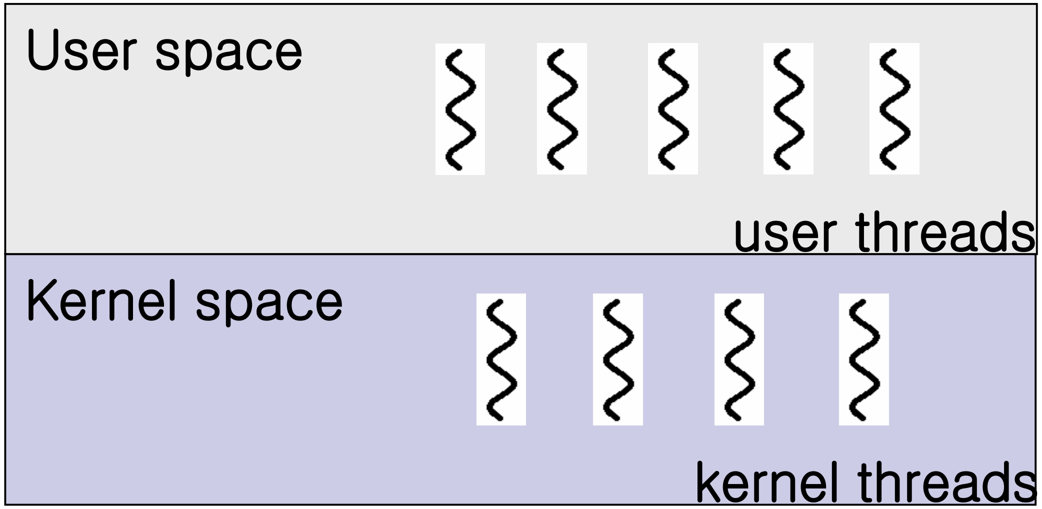


# Agenda

- Overview
- <u>Multithreading models</u>
- Thread libraries
- Threading issues
- Operating system examples

# Multithreading Models

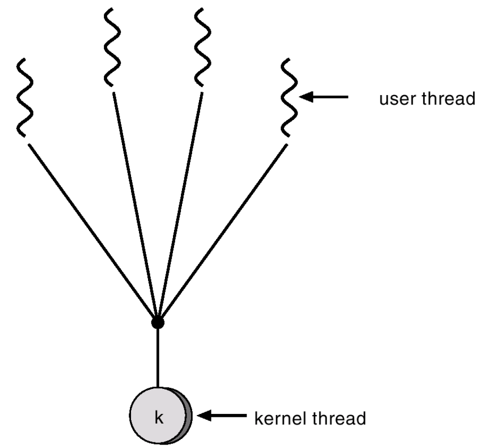
- Major issue: correspondence between user treads and kernel threads



# Multithreading Models

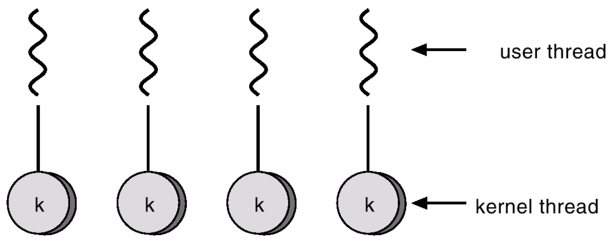
- Many-to-one model
  - Many user threads are mapped to single kernel thread
  - Threads are managed by user-level thread library

  Ex) Green threads, GNU Portable Threads




- One-to-one model
  - Each user thread is mapped to a kernel thread
  - Provides more concurrency
  - Problem: overhead

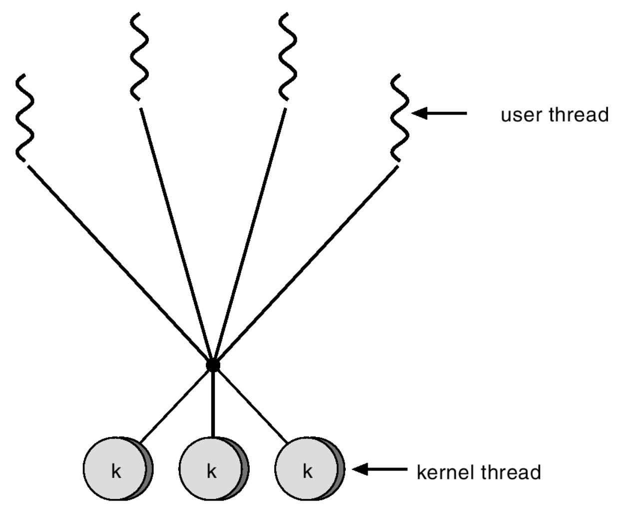
  Ex) Linux, Windows, Solaris

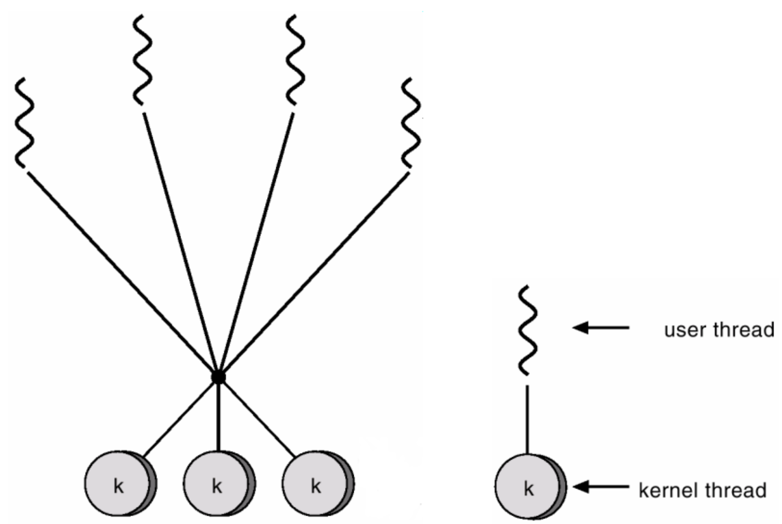


# Multithreading Models

- Many-to-Many model
  - Multiplex many user level threads to smaller or equal number of kernel threads
  - Compromise between n:1 model and 1:1 model
- Two-level model
  - Variation of N:M model
  - Basically N:M model
  - A user thread can be bound to a kernel thread

  Ex) IRIX, HP-UX, Tru64, Solaris (old)



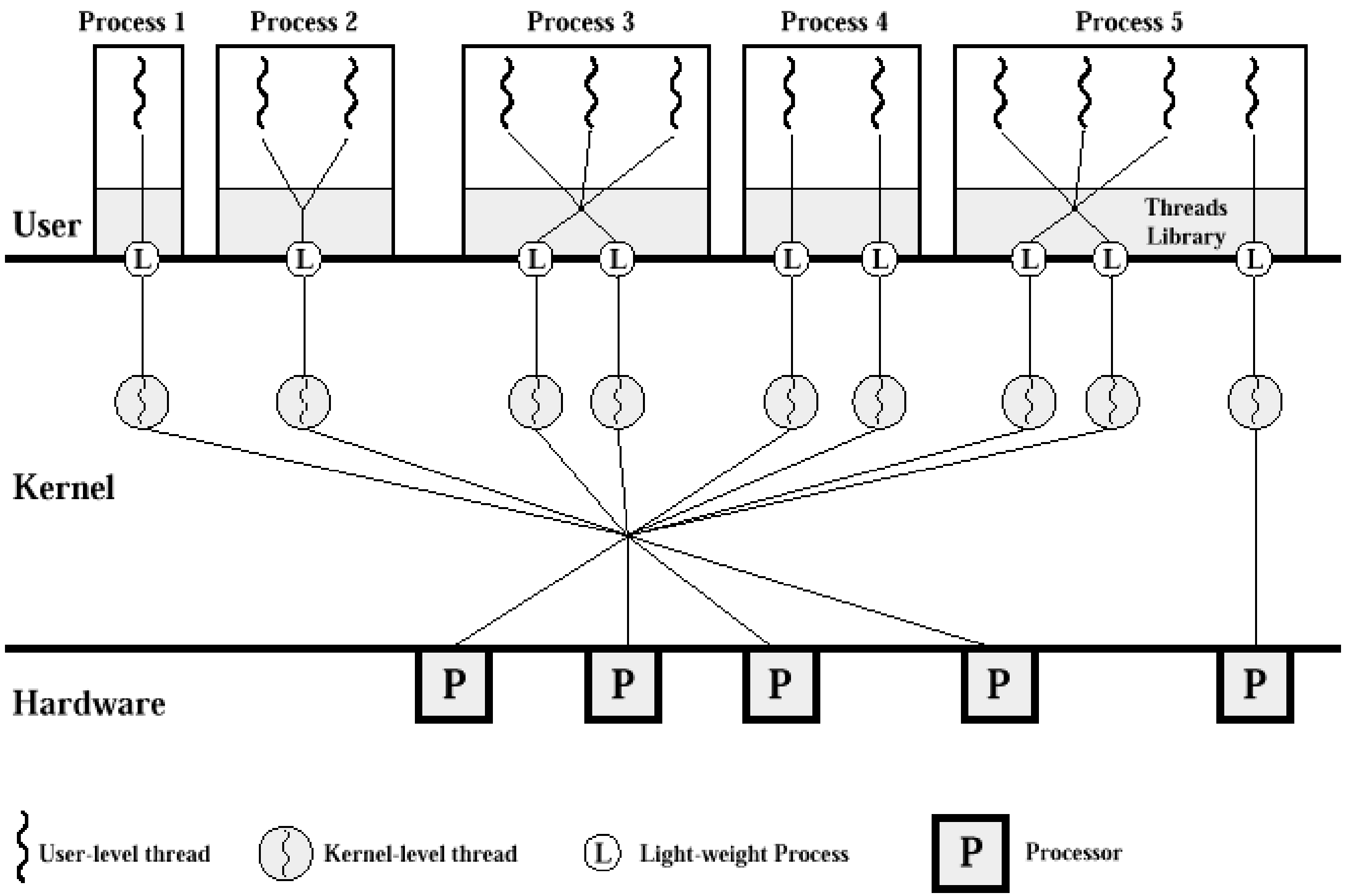



# Scheduler Activation

- An issue about many-to-many model and two-level model: communication between the kernel and the thread library.
- Scheduler activation: one scheme for communication between user thread library and kernel
- In many-to-many model and two-level model, user threads are connected with kernel threads through LWP
- Lightweight process (LWP)
  - A data structure connecting user thread to kernel thread.
  - Basically, a LWP corresponds to a kernel thread, but there are some exceptions.
  - To the user-level thread library, a LWP appears to be a virtual processor.

# Scheduler Activation and LWP

- Connection between user / kernel threads through LWP



# Scheduler Activation

- Scheduler activation
  - Kernel provides a set of virtual processors(LWP's).
  - User level thread library schedules user threads onto virtual processors.
  - If a kernel thread is blocked or unblocked, kernel notices it to thread library(upcall).
  - Upcall handler schedules properly.
    - If a kernel thread is blocked, assign the LWP to another thread
    - If a kernel thread is unblocked, assign an LWP to it.

# Agenda

- Overview
- Multithreading models
- <u>Thread libraries</u>
- Threading issues
- Operating system examples

# Thread Libraries

- Thread library: set of API's to create and manage threads
  - User level library
  - Kernel level library

Examples)

- POSIX Pthreads: a specification with various implementations.
  - LinuxThreads
  - NPTL (Native POSIX Thread Library)
  - GNU Portable Threads
  - Open source Pthreads for win32
  - Etc.
- Win32 threads
- Java threads

# POSIX Pthreads

- Reading assignment: Search the Internet for the following functions, and study them.
- API functions
  - int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);
    - Creates thread
  - int pthread_attr_init(pthread_attr_t *attr);
    - Initializes *attr* by default values
  - int pthread_join(pthread_t th, void **thread_return);
    - Waits for a thread *th*.
  - void pthread_exit(void *retval);
    - Terminates thread

# Example

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

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

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>\n");
        exit(0);
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be <= 0\n",atoi(argv[1]));
        exit(0);
    }
    /* 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);
    return 0;
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int upper = atoi (param);
    int i;
    sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper ; i++)
            sum += i;
    }
    pthread_exit(0);
}
```

# Win32 Threads

- Create

```
HANDLE WINAPI CreateThread(
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        SIZE_T dwStackSize,
        LPTHREAD_START_ROUTINE lpStartAddress,
        LPVOID lpParameter,
        DWORD dwCreationFlags,
        LPDWORD lpThreadId
);
```

  See http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createthread.asp

- Wait

```
DWORD WINAPI WaitForSingleObject(
        HANDLE hHandle,
        DWORD dwMilliseconds
);
```

- Close (deallocate) handle

```
BOOL CloseHandle(LPDWORD lpThreadId);
```

# Windows Multithreaded C Program

```
#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;
   }
```

```
   /* 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

- 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
    - For detail, search 'Java Thread class' from Internet
  - Implementing the Runnable interface

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

# Java Thread using Thread Class

- Extending Thread class

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

- Launching thread

```
PrimeThread p = new PrimeThread(143);
p.start();
```

# Java Thread using Running Interface

- Extending Thread class

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

- Launching thread

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

# Java Multithreaded Program

```
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);
  }
}
```

# Java Multithreaded Program (Cont.)

```
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>"); }
}
```

# Implicit Threading

- Creation and management of threads done by compilers and run-time libraries rather than programmers

- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch

- Other methods include Microsoft Threading Building Blocks (TBB), java.util.concurrent package

# Thread Pools

- Create a number of threads in a pool where they await work
- 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
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically
- Windows API supports thread pools:

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

# OpenMP

- Provides support for parallel programming in shared-memory environments
  - Set of compiler directives and an API for C, C++, FORTRAN
  - Identifies parallel regions – blocks of code that can run in parallel

- Create as many threads as there are cores

```
#pragma omp parallel
    printf("Hello, World!₩n");
```

- Run for loop in parallel

```
#pragma omp parallel for
    for(i=0;i<N;i++) {
        c[i] = a[i] + b[i];
    }
```

# Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in "^{ }" – ^{ printf("I am a block"); }
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue

# Grand Central Dispatch

- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called main queue
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

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

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

# Agenda

- Overview
- Multithreading models
- Thread libraries
- <u>Threading issues</u>
- Operating system examples

# Threading Issues

- fork() and exec()
- Cancellation
- Signal handling
- Thread-local storage
- Scheduler activation (already covered)

# fork() and exec()

- fork() on multithreaded process
  - Duplicates all threads in the process?
  - Duplicates only corresponding thread?
  
  → UNIX supports two versions of fork
    - fork(), fork1()

- exec() on multithreaded process
  - Replace entire process

# Cancellation

- Thread cancellation
  - Terminating a thread (target thread) before it has completed

```
pthread_t tid;

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

. . .

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

- Problem with thread cancellation
  - A thread share the resource with other threads

  cf. A process has its own resource.

  → A thread can be cancelled while it updates data shared with other threads

# Cancellation

- Cancellation scenarios
  - Asynchronous cancellation
    - Immediate termination
  - Deferred cancellation
    - Target thread checks periodically whether it should terminate.
      Ex) pthread_testcancel() of pthread
    - Safer than asynchronous cancellation

# Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

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

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - I.e. pthread_testcancel()
    - Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through signals

# Signal Handling

- Signal: mechanism provided by UNIX to notify a process a particular event has occurred
  - A signal can be generated by various sources.
  - The signal is delivered to a process.
  - The process handles it.
    - Default signal handler (kernel)
    - User-defined signal handler

- Types of signal
  - Synchronous: signal from same process

    Ex) illegal memory access, division by 0
  - Asynchronous: signal from external sources

    Ex) <Ctrl>-C

# What is a Signal?

- A signal is a software interrupt delivered to a process.
  - The operating system uses signals to report exceptional situations to an executing program
- A signal is a limited form of inter-process communication used in Unix and other POSIX-compliant operating systems.
  - Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred

# Examples of UNIX Signals

- Signals in UNIX (in signal.h)

```
#define SIGHUP     1  /* hangup */
#define SIGINT     2  /* interrupt */
#define SIGQUIT    3  /* quit */
#define SIGILL     4  /* illegal instruction (not reset when caught) */
#define SIGTRAP    5  /* trace trap (not reset when caught) */
#define SIGIOT     6  /* IOT instruction */
#define SIGABRT    6  /* used by abort, replace SIGIOT in the future */
#define SIGEMT     7  /* EMT instruction */
#define SIGFPE     8  /* floating point exception */
#define SIGKILL    9  /* kill (cannot be caught or ignored) */
#define SIGBUS     10 /* bus error */
#define SIGSEGV    11 /* segmentation violation */
#define SIGSYS     12 /* bad argument to system call */
#define SIGPIPE    13 /* write on a pipe with no one to read it */
#define SIGALRM    14 /* alarm clock */
#define SIGTERM    15 /* software termination signal from kill */
```

# Installing Signal Handler

- **Defining signal handler**

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum,
                    sighandler_t handler);
```

- **Example**

```
#include <signal.h>
#include <unistd.h>

void sig_handler(int signo);

// press CTRL-₩ to terminate this program
int main()
{
    int i = 0;
    signal(SIGINT, (void *)sig_handler);

    while(1){
        printf("%d₩n", i++);
        sleep(1);
    }
    return 1;
}

void sig_handler(int signo)
{
    printf("SIGINT was received!₩n");
}
```

# Signal Handling

- Question: To what thread the signal should be delivered?

- Possible options
  - To the thread to which the signal applies
  - To every thread in the process
  - To certain threads in the process
  - Assign a specific thread to receive all signals

  ➔ depend on type of signal

- Another scheme: specify a thread to deliver the signal

  Ex) pthread_kill(tid, signal) in POSIX

# Thread-Local Storage

- Thread-local storage (TLS) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to static data
  - TLS is unique to each thread

# Agenda

- Overview
- Multithreading models
- Thread libraries
- Threading issues
- Operating system examples

# Windows XP Threads

- Implements the one-to-one mapping, kernel-level
  - Additionally, many-to-many model is supported by fiber library

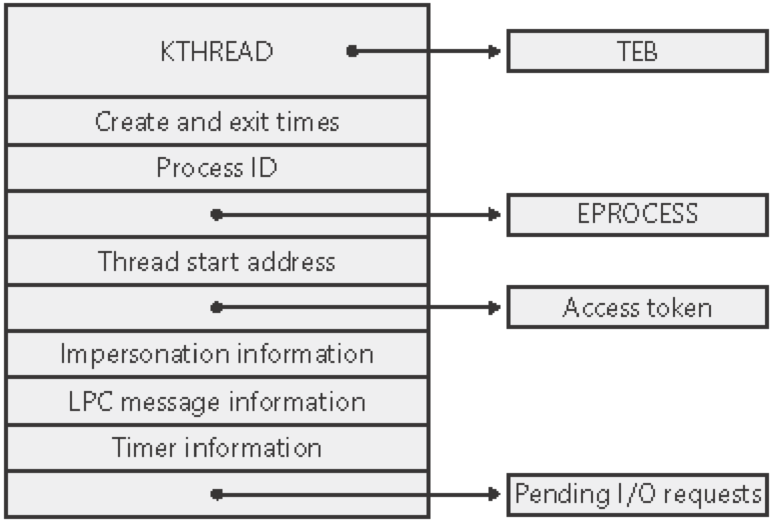
- Each thread contains
  - A thread id, register set, separate user and kernel stacks, private data storage area (for run-time libraries and DLLs)

  Cf. The register set, stacks, and private storage area are known as the context of the thread.

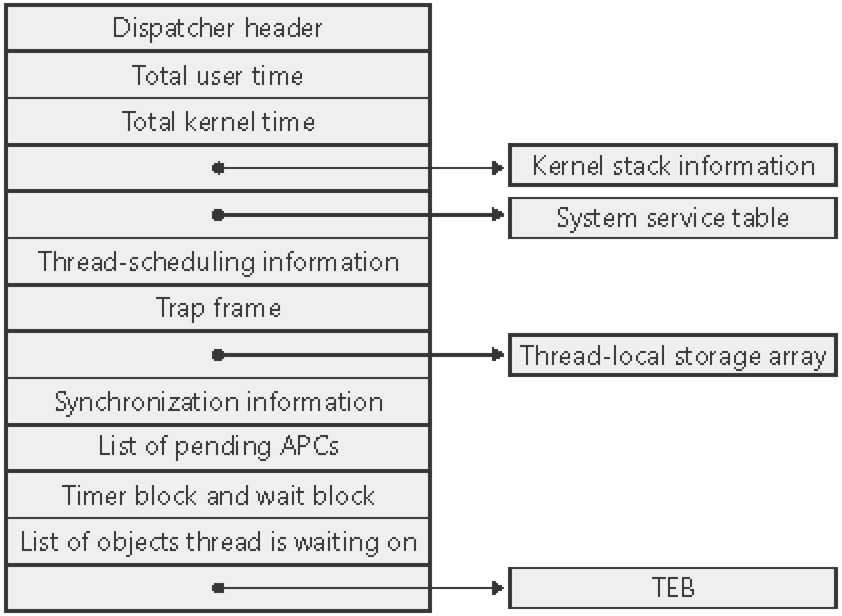
# Internal Data Structures

- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
    - Information for thread scheduling and synchronization
  - TEB (thread environment block)
    - Context information for the image loader and various Windows DLL

# Layout of ETHREAD and KTHREAD

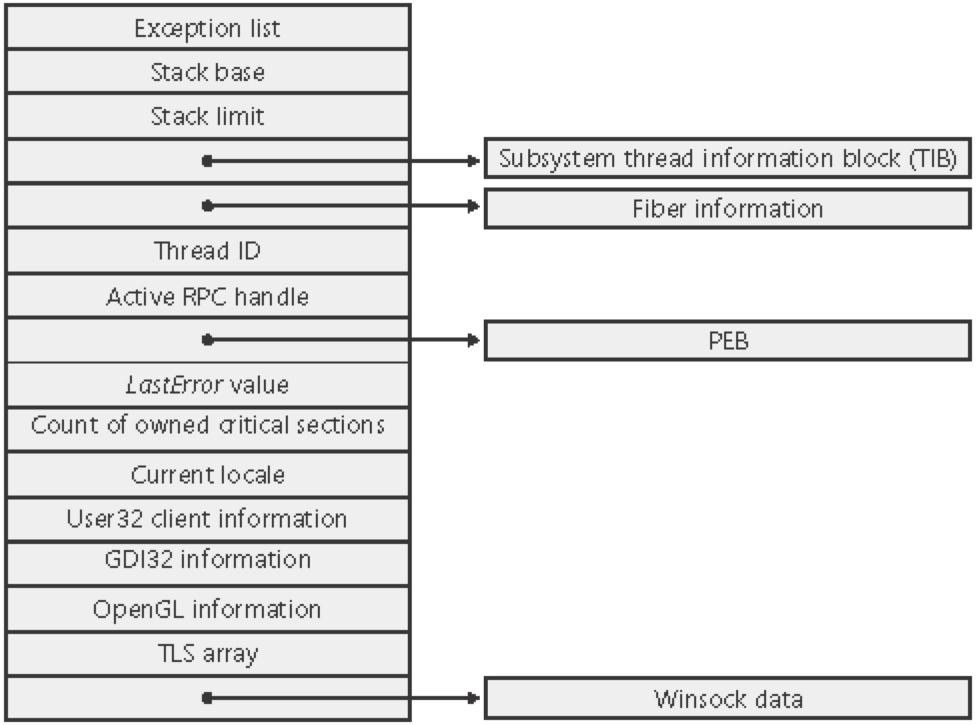



ETHREAD

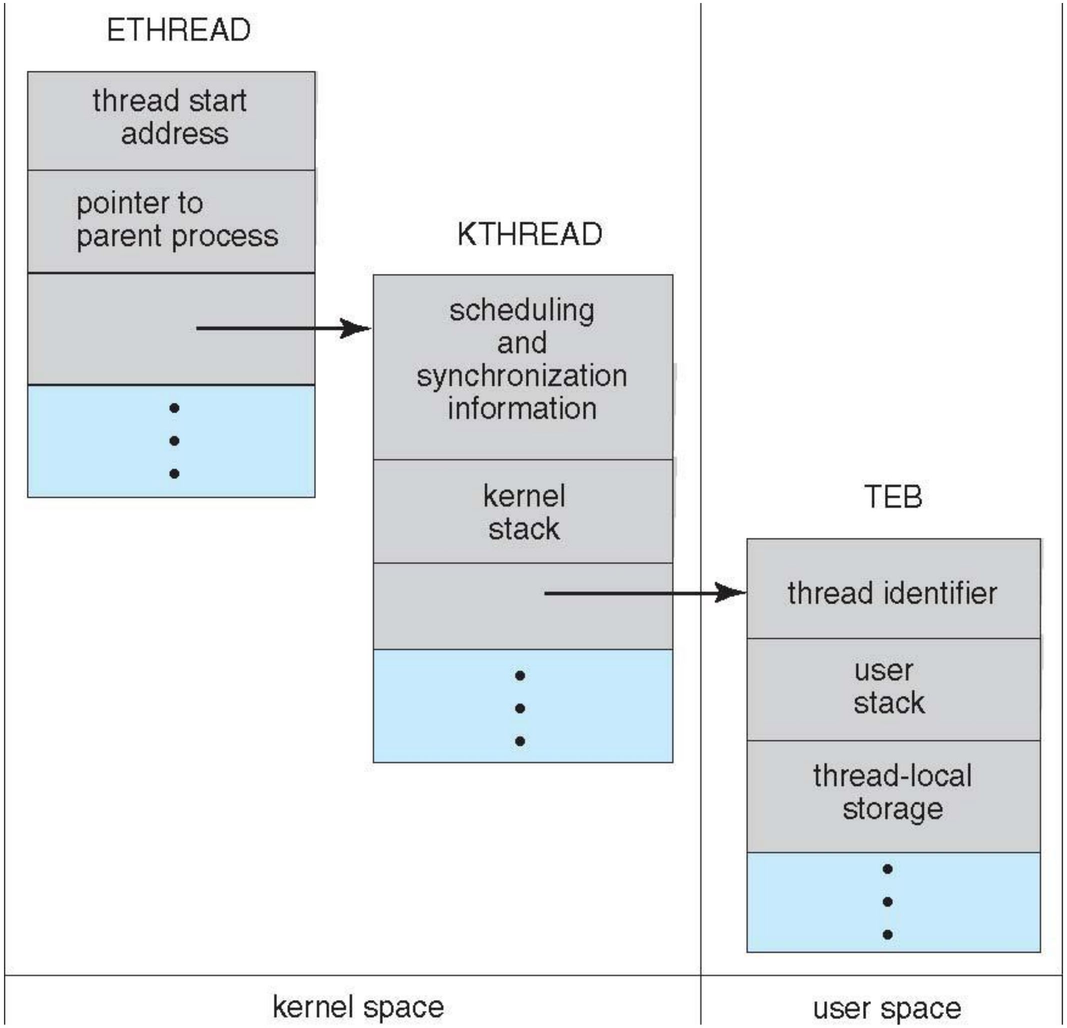



KTHREAD

# Layout of TEB



# Windows XP Threads



# Linux Threads

- First implementation: LinuxThreads
  - Thread is created by clone() system call
  - However, no difference between process and thread
    - Term '**task**' is used more frequently than 'process' or 'thread'
  - User can control how much the resource is shared between parent and child.

# Linux Threads

- Problems of *LinuxThreads*
  - It did not scale well,
    - There is a limitation on the number of threads, generally between 1024 and 8192.
    - The overhead of creating and destroying processes is relatively high.
  - The manager thread resulted in some fragility and another scaling bottleneck.
  - Some required POSIX semantics were not possible. Each thread had its own PID.

- NPTL (Native POSIX Thread Library)
  - Better than LinuxThreads in performance and compatibility