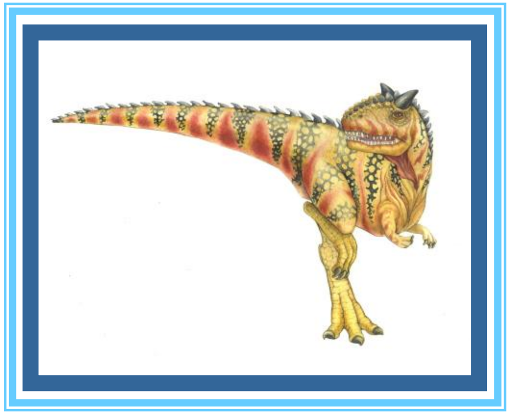
Chapter 4: Threads



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

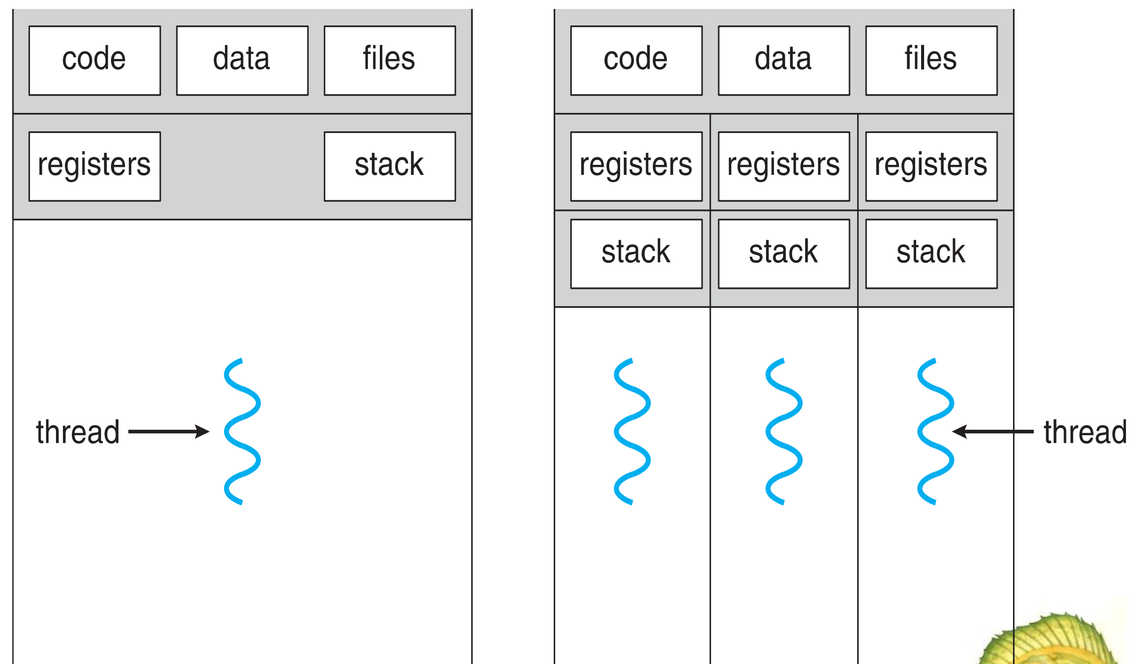
- To introduce the notion of a thread
  - A fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

Overview

- Thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)
- Traditional (heavyweight) process has a single thread

  -> It includes one program counter so only one sequence of instructions can be carried out at any given time.

- In figure, multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as opened files

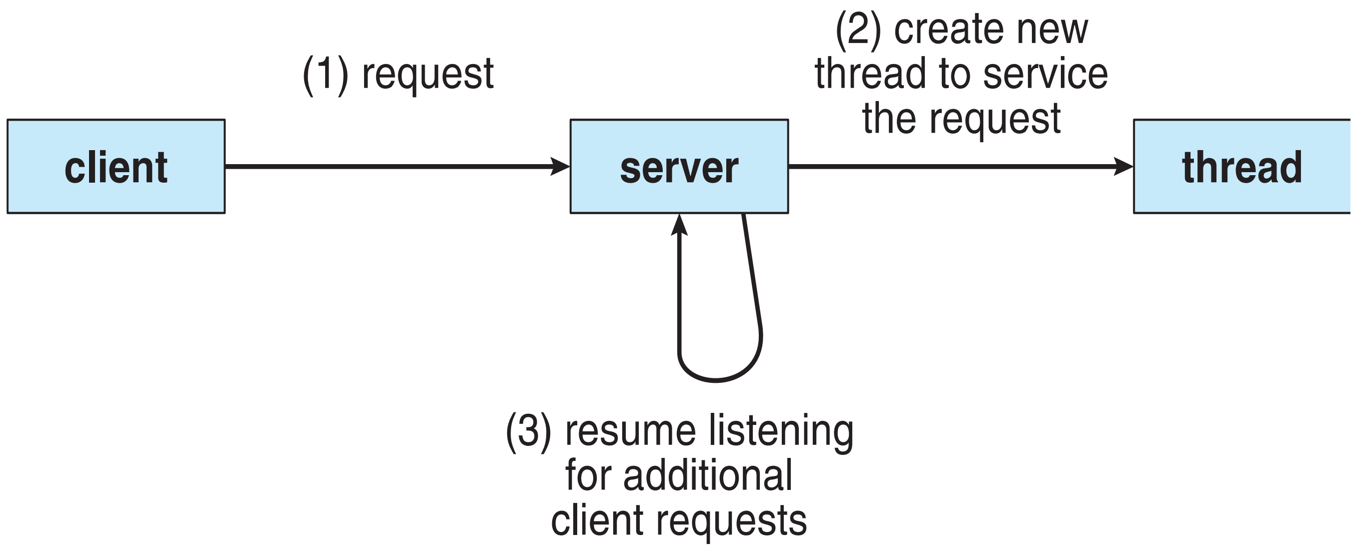


single-threaded process　　　multithreaded process

Motivation of Thread

- Most modern applications are multithreaded
- A word processor - A background thread may check spelling and grammar while a foreground thread handles user input (keystrokes), while yet the third thread loads images from the hard drive, and the fourth does periodic automatic backups of the file being edited
- A web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port.)
- Kernels are generally multithreaded

Multithreaded Server Architecture



Benefits

- There are four major categories of benefits to multi-threading:

1. **Responsiveness** – may allow continued execution if a part of process is blocked, especially important for user interfaces

2. **Resource Sharing** – threads share resources of process, which allows multiple tasks to be performed simultaneously in a single address space (Ex> multiple counter staffs in a hotel)

3. **Economy** – cheaper than process creation, thread switching lower overhead than context switching

4. **Scalability** – process can take advantage of multiprocessor architectures. A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors

Multicore Programming

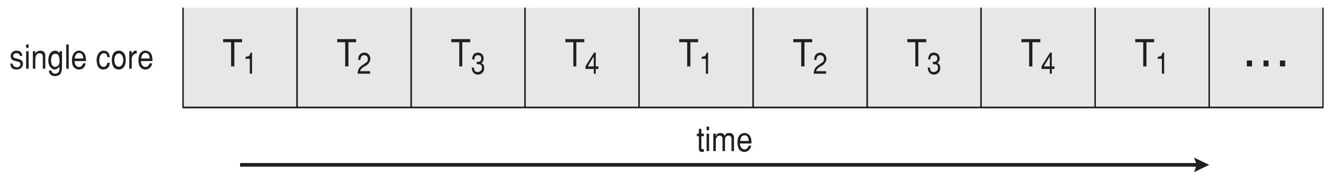
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- On a system with a single computing core, **concurrency** can be emulated by interleaving the execution of the threads over time
  - It is possible to have concurrency without parallelism
  - On a single processor / core, scheduler provides concurrency

Multicore Programming (Cont.)

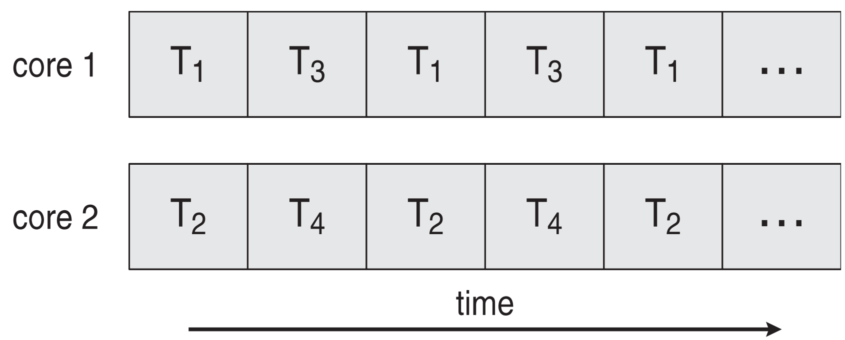
- Types of parallelism for multiple cores
  - **Data parallelism** – distributes subsets of the same data across multiple cores, applies same type operation on each subset
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as hardware thread support
  - Oracle T4 (a server) supports eight threads per core and there are 8 cores
  - Intel Core-i7 (8th gen): 6 cores with hyper-threading (SMT, Simultaneous Multi-Threading) = total 12 threads

Concurrency vs. Parallelism

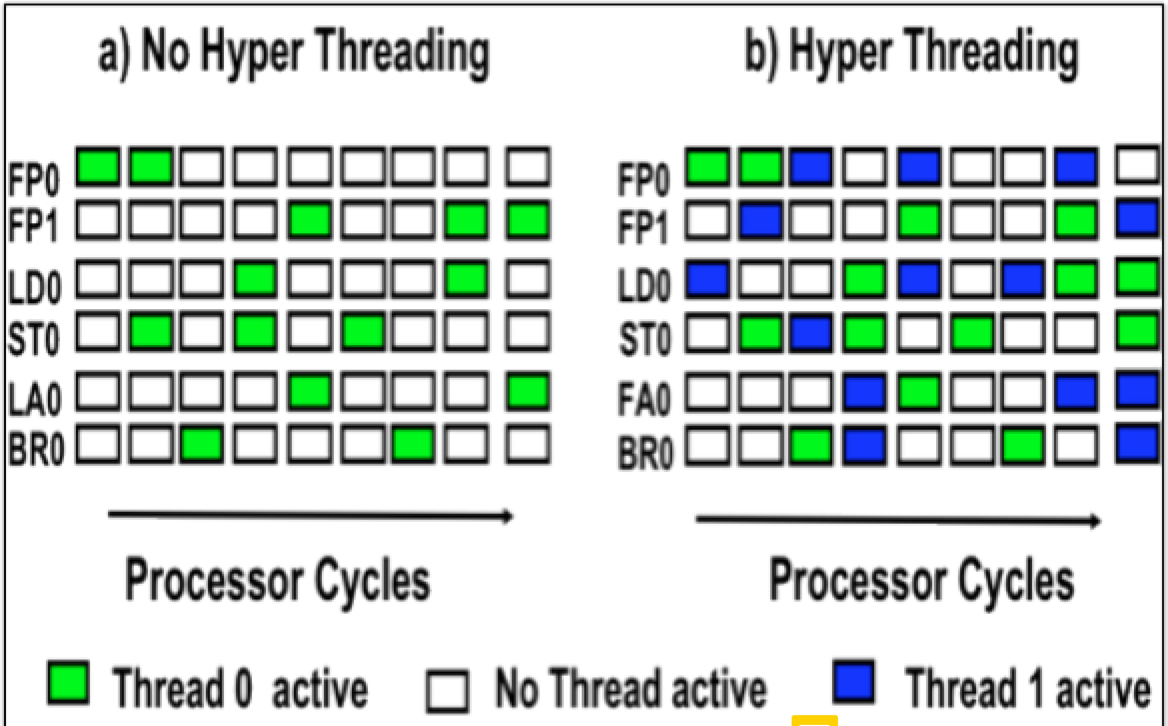
- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



Hyper Threading



- FP0/1 – floating-point unit
- LD0 – load unit / ST0 – store unit
  - Loads data from memory to registers and stores from registers to memory
- LA0 – load address unit
- BR0 – branch unit
  - Changes the program counter

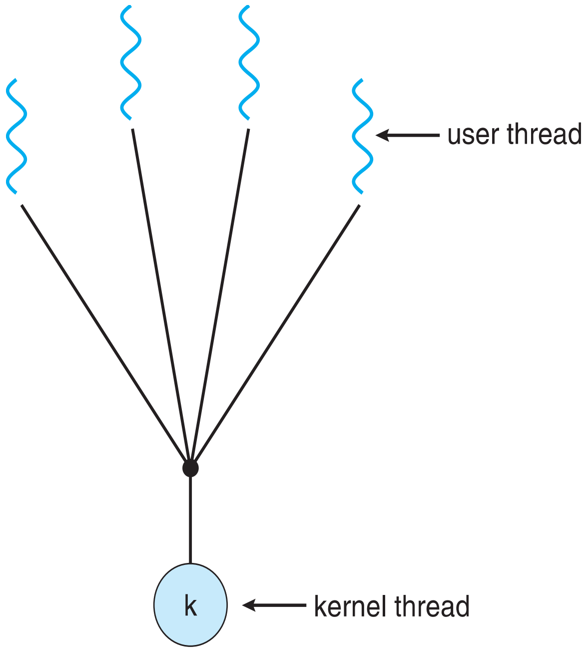
Subhash Saini, Haoqiang Jin, Robert Hood, David Barker, Piyush Mehrotra and Rupak Biswas, The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications, In Proceedings of HPC, 2012.

User Threads and Kernel Threads

- **User threads** are supported above the kernel, without kernel support.
  - Application programmers would put them into their programs
- **Kernel threads** are supported within the kernel of the OS itself
  - All modern OSes support kernel level threads
  - Kernel performs multiple simultaneous tasks
  - Kernel services multiple kernel system calls simultaneously
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies:
  - Many-to-One Model
  - One-to-One Model
  - Many-to-Many Model

Many-to-One

- Many user-level threads are mapped to a single kernel thread
- Thread is managed by the **thread library** in user space, which is very efficient
- However, if a **blocking system call** is made, then **the entire process blocks**, even if the other user threads would otherwise be able to continue
- Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
- **Green threads** for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.

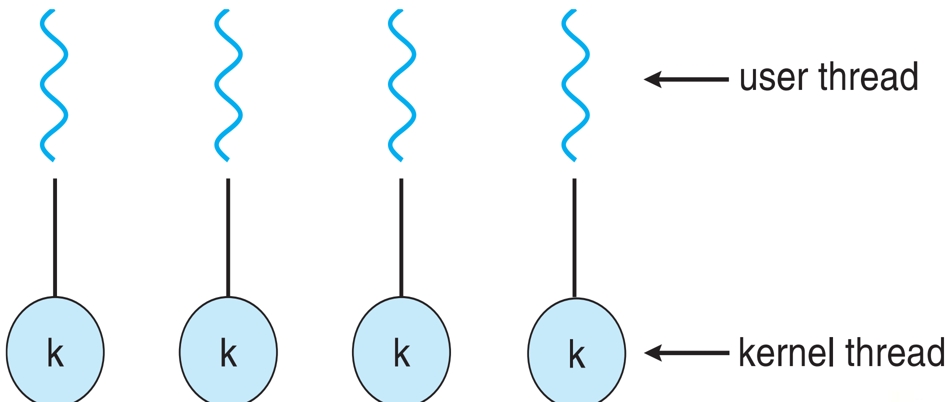


One-to-One

- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a corresponding kernel thread
- More concurrency than many-to-one

- Overhead of managing the one-to-one model increases so this overhead slows down the system.
- Most implementations of this model place a limit to the maximum number of threads

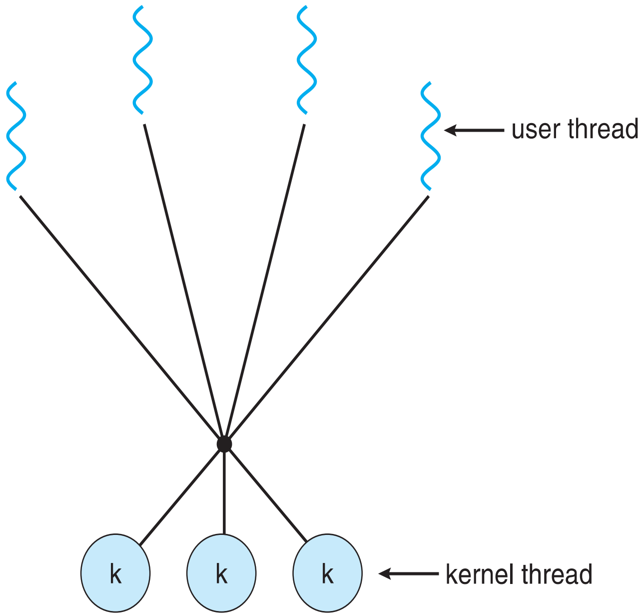
- Examples
  - Windows 95 - XP
  - Linux
  - Solaris 9 and later



Many-to-Many Model

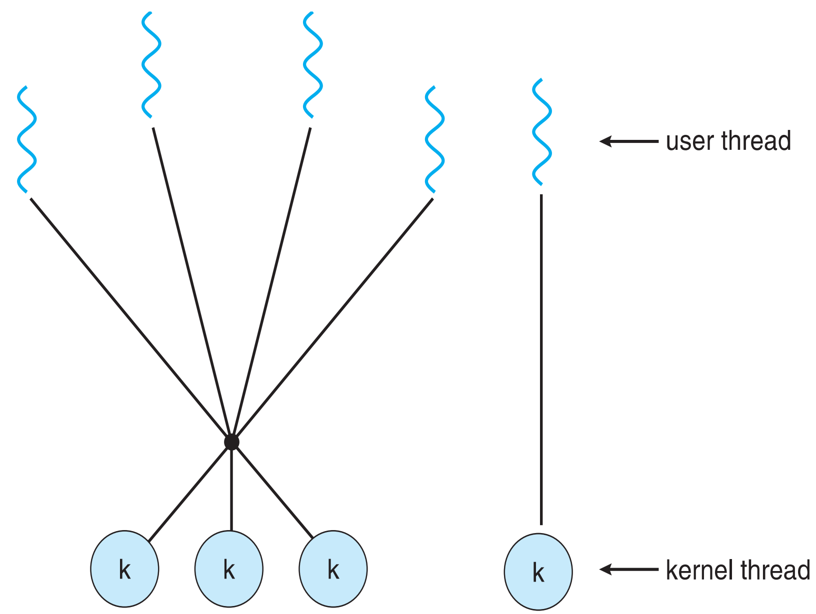
- Multiplexes many user level threads to an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Blocking kernel system calls do not block the entire process
- An application allocated many kernel threads can be split across multiple CPUs

- Windows with the ThreadFiber package



Two-level Model

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



Thread Libraries

- **Thread library** provides APIs to a programmer for creating and managing threads
- Two primary ways of implementation
  - Thread library exists entirely in user space (a local function call, not a system call)
  - Kernel-level library supported by the OS
- Three main thread libraries in use today:
  - POSIX pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
  - Win32 threads - provided as a kernel-level library on Windows systems
  - Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either pthreads or Win32 threads depending on the system.

pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- **Specification**, not implementation
- API only specifies behavior of the thread library, implementation is up to development of the library

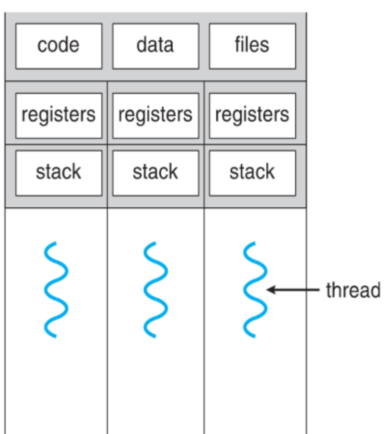
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- The following example will demonstrate the use of threads for calculating the sum of integers from 0 to N in a separate thread, and storing the result in a variable "sum"

Pthreads Example

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

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

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

Global variables are shared amongst all threads.

• • •




```c
   /* get the default attribute
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

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

pThreads begin execution in a specified function, in this example the runner( ) function

One thread can wait for the others to rejoin before continuing.

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

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Windows Multithreaded C Program

```c
#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 Thread Example

```
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 threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

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

```
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

- Growing in popularity as the number of threads increases, program correctness become more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch (GCD)
- Other methods include Microsoft Threading Building Blocks (TBB), ***java.util.concurrent*** package

Thread Pools

- Creating new threads every time when they are needed and then deleting it when it is done can be inefficient
- Create a number of threads when the process first starts, and put those threads into a **thread 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 (i.e., can place a bound on the number of threads concurrently active in the system)
- Windows API supports thread pools:

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

// to cause a thread from the thread pool to invoke PoolFunction()

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

OpenMP

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

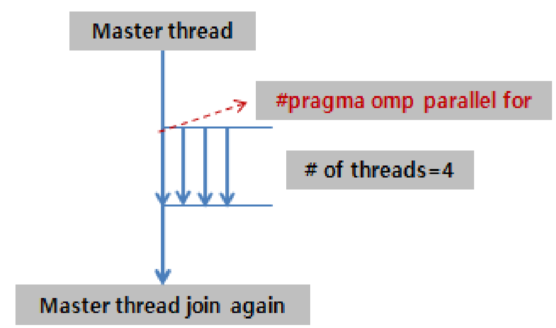
#pragma omp parallel

-> Create as many threads as the number of cores

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

-> Divide the work contained in the for loop among the threads

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```



Grand Central Dispatch (GCD)

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Internally, the thread pool of GCD is implemented the POSIX thread

- Allows identification of **parallel blocks** by placing a carat **^**
  -> E.g., Block is in “^{ }” - ˆ{ printf("I am a block"); }

- GCD schedules blocks by placing them on one of several dispatch queues
  - Assigned to available thread in thread pool when removed from queue

- Like OpenMP, GCD manages most of the details of threading

Grand Central Dispatch (cont.)

- Two types of dispatch queues:
  - Serial queue

    -> Blocks are removed in FIFO order. Each block must complete its execution before another block is removed. Each process has its own serial queue (known as its **main queue**).

    -> Programmers can create additional serial queues that are local to particular processes
  - Concurrent queue

    -> Blocks are removed in FIFO order but several blocks may be removed at a time

    -> Three system wide queues with priorities low, default, high

    -> The following code illustrates obtaining the default-priority queue and submitting a block to the queue using the *dispatch_async()* function:

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

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

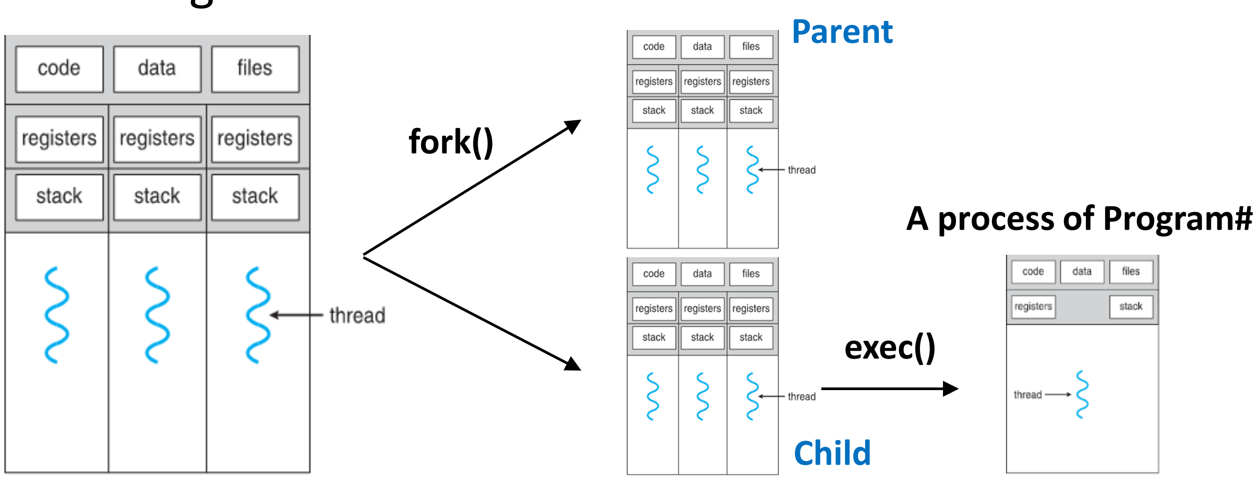
Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads of the new process?
  - *exec()* loads another process to the forked process (child process)
  - If the new process execs right away, there is no need to copy all the other threads. If it doesn't, then the entire process should be copied
  - Some UNIXes have two versions of fork

- **exec()** usually works as normal – replace the running process including all threads



A process of Program0

Signal Handling

- **Signals** are used in UNIX systems to notify a process (inter-process or kernel-to-process) that a particular event has occurred.
- A **signal handler** is used to send signals to process as follows:

  1) Signal is generated by particular events

  2) Signal is delivered to a process

  3) Signal is handled by one of two signal handlers:

  - Default and user-defined

- Every signal has **default handler** that kernel runs when handling signal
  - ***User-defined signal handler*** *can override default*
  - *For single-threaded, signal delivered to process*

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded process?
  - *Deliver the signal to the thread to which the signal applies*
  - *Deliver the signal to every thread in the process*
  - *Deliver the signal to certain threads in the process*
  - *Assign a specific thread to receive all signals for the process*

- The standard UNIX function for delivering a signal is

  kill(pid, signal), pthread_kill(tid, signal)

  - *These system calls specify the process/thread (pid/tid) to which a particular signal (signal) is to be delivered.*
  - *Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, in some cases, an asynchronous (external) signal may be delivered only to the first thread found that is not blocking it.*

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is a **target thread**
- Two general approaches:
    - ***Asynchronous cancellation*** *terminates the target thread immediately*
    - ***Deferred cancellation*** *allows the target thread to periodically check if it should be cancelled*

- Pthread code to create and cancel a thread:

```
pthread_t tid;

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

. . .

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

Thread Cancellation (Cont.)

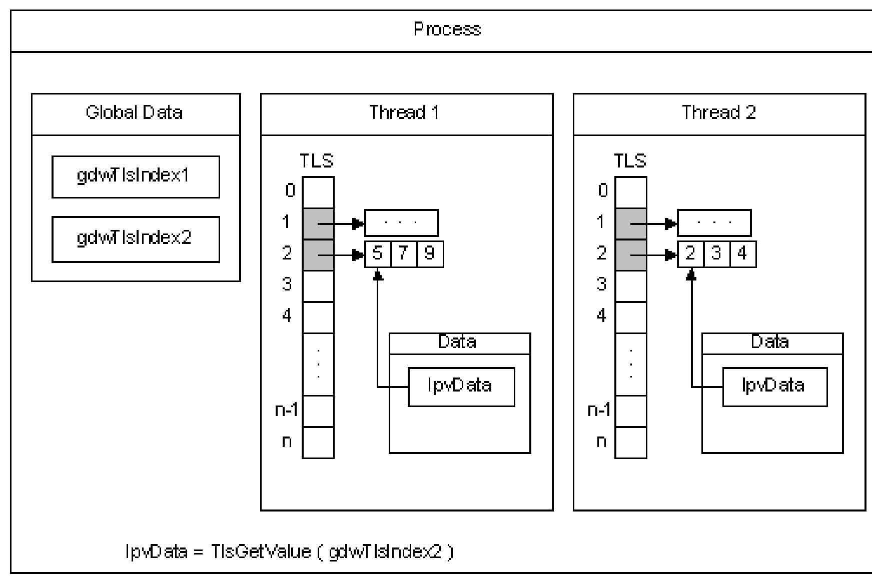
- Invoking thread cancellation requests cancellation, but actual cancellation occurs depending on thread state

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

- If the target thread disabled cancellation, cancellation signal remains pending until the thread enables it
- Default type is deferred
  - *Cancellation only occurs when thread reaches* ***cancellation point***
    - E.g., *pthread_testcancel()*, then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

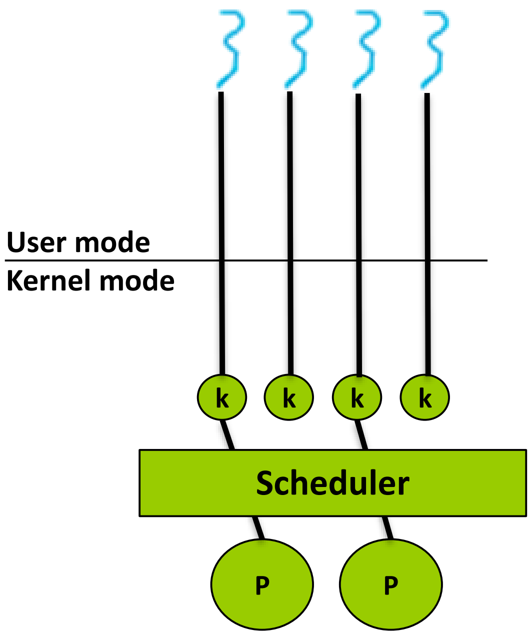
Thread-Local Storage

- **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)
  - *E.g., save unique transaction id in TLS*
- 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*

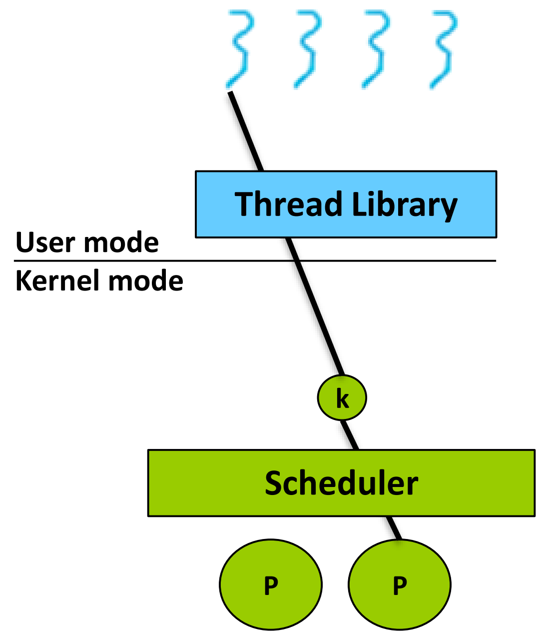


Scheduler Activations (1)

- First, imagine the case of 1:1 and M:1
  - 1:1 model does not require additional notification between a user thread and a kernel thread
  - M:1 model also does not require additional notification because when a user thread invokes a system call that blocks, then a kernel thread blocks and threads library has to wait until the system call ends



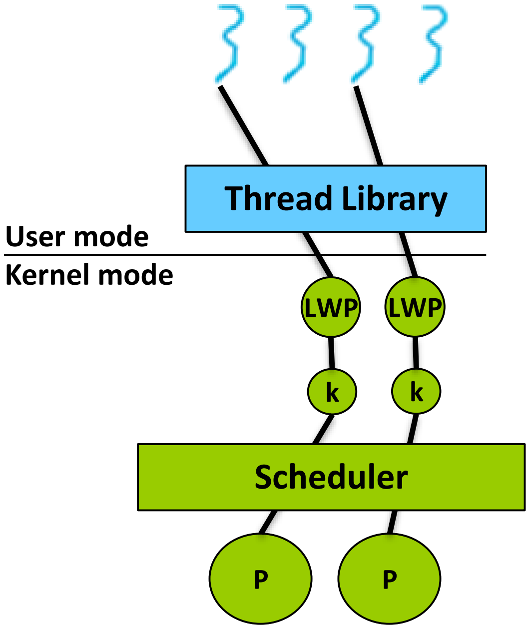
1:1 model



M:1 model

Scheduler Activations (2)

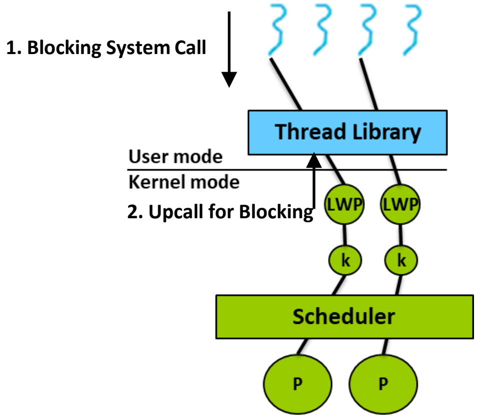
- Both M:M and two-level models typically place an intermediate data structure, **lightweight process** (LWP), between the user (thread library) and kernel threads for communication to maintain the appropriate number of kernel threads allocated to the application
  - To the user-thread library, LWP appears to be a virtual processor on which the application can schedule user thread to run
  - Each LWP is attached to kernel thread
  - How many LWPs to create?

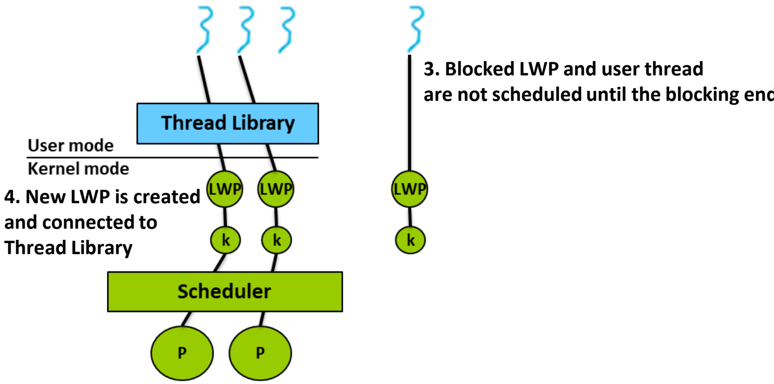


M:M model

Scheduler Activations (3)

- **Scheduler activation** scheme enables **upcall** procedure (occurs when an application thread is about to block) to communicate between the kernel and the **upcall handler** (user-side scheduler) in the thread library
  - A new LWP is allocated for the upcall handler to run on, then the upcall handler reschedules the user threads. OS will also issue upcalls when a thread becomes unblocked, so the thread library can make appropriate adjustments.





You can read a paper in this link for better understanding

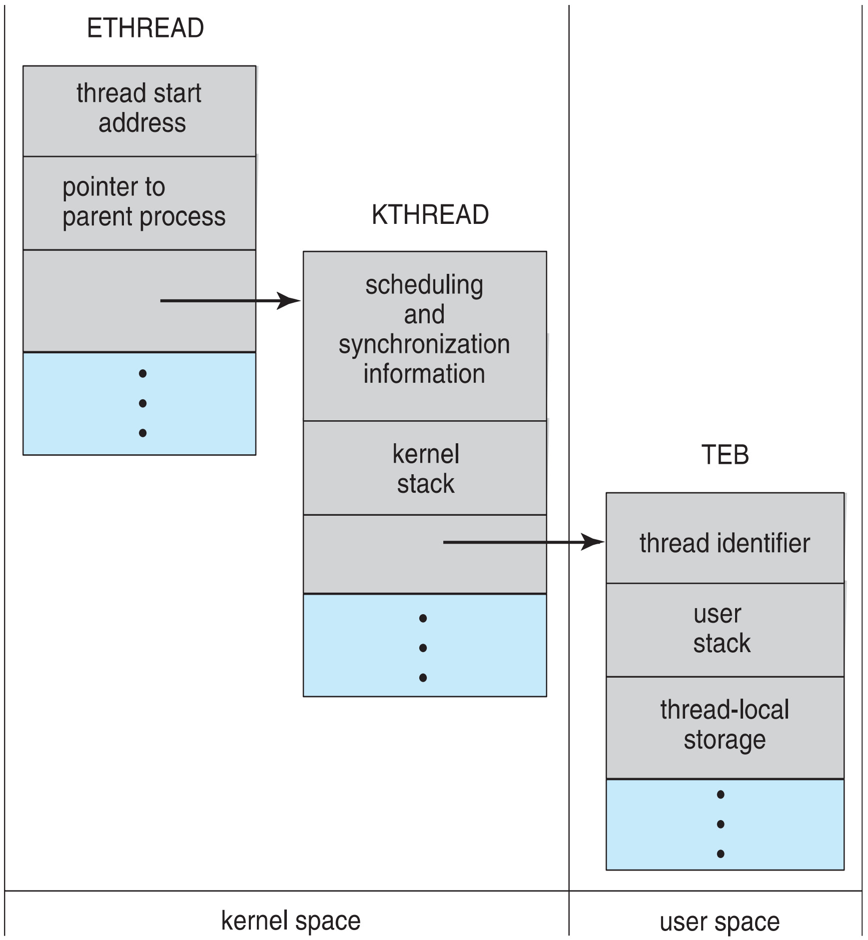
http://web.mit.edu/nathanw/www/usenix/freenix-sa/freenix-sa.html

Example (1) - Windows Threads

- The Win32 API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping model
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space



Windows Threads Data Structures

Linux Threads

- Linux refers to them as **tasks** rather than **process** or **thread**
- Thread creation is done through *clone()* system call
- *clone()* passes a set of flags that determine how much sharing is to take place between the parent and child tasks
  - Flags control behavior

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.

- The child task points to process data structures of the parent task (shared or unique)
  - When *fork()* is invoked, a new task is created, along with a copy of all the associated data

End of Chapter 4