Chapter 4: Threads

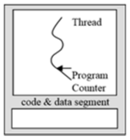
Motivation

- Generally, a process is defined by two characteristics:
  1. An execution state (running, ready, etc.) and the location at which it is executing
  2. Resources it uses such as the memory, I/O channels, I/O devices, files, etc.
- A typical process has:
  - An address space to hold the process image
  - Protected access to CPU, other processes (for inter-process communication), files and I/O resources.
- In many instances, it is useful for resources to be shared and accessed concurrently. For example, a browser may have a thread to display text/image, receiving another request for a page, another thread to retrieve data from network yet another one to print a web page.

Motivation (contd.)

- **Traditional approach:**

  Single process (thread of execution) in which concept of thread is not recognized and the process is called a **heavyweight process.**

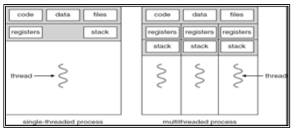


- **In modern operating Systems,** the Unit of CPU utilisation or dispatching is usually referred as **thread.**
  - Generally consists of program counter, a register set, and a stack space.
  - In a multiple process model, each process operates independent of others → useful if jobs performed are unrelated.
  - Multiple processes can also perform the same task → less efficient.

Motivation (contd.)

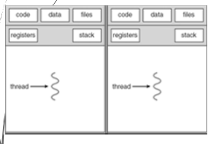
- **Multithreading** is the ability of an OS to support multiple threads of execution within a single process.
- Shares with other peer threads its code section, data section, and other OS resources. Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the 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

Threading

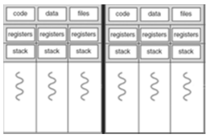


One process one thread
e.g. MS-DOS

One process. multiple threads
e.g. Java run-time engine

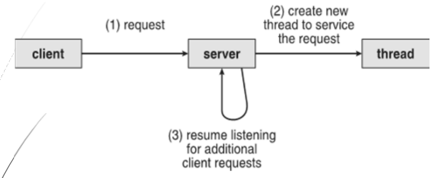


Multiple processes
one thread / process, e.g. UNIX



Multiple processes, multiple threads / process
e.g., Solaris, Linux, NT, Mach, etc.

Multithreaded Server Architecture


client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** due to extensive resource sharing
  - Less time to create a new thread in an existing process than to create a brand new process. In Solaris, it is 30 times faster to create a new thread than creating a new process. Context switching is 5-times faster.
  - Less time to terminate a thread
  - Less time to switch between two threads within the same process (a thread context switch still requires a register set switch but no memory-management-related work need to be done).
  - Enhance efficiency of communication between different executing programs.
    - No intervention of kernel to provide protection / communication.
- **Scalability –** process can take advantage of multiprocessor architectures
  - Multithreading on a multi-CPU machine or multi-CORE processor increases parallelism → threads may be running in parallel on different processors.

Multicore VS Multi-Processor Systems

- A multi-core architecture is more efficient than a multi-processor system because of:
  - faster on-chip communication
  - Uses less power.
  - Each core has its own set of registers as well as own cache (or can also use shared cache). These N-cores appear to the OS as N standard processors.
  - Economy

Thread Model

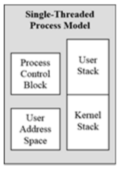
Within a process, there may be one or more threads, each with the following:

- A thread execution state (Running, ready, blocked, terminated) - much like the states of a process
- A saved thread context when not running;

  one-way to view a thread is as an independent program counter operating within a process.
- An execution stack.
- Some per-thread static storage for local variables
- Access to memory and resources of its process, shared with all other threads in that process.

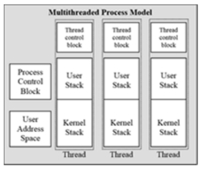
Thread Model (contd.)

In a single threaded process model, the representation includes:



- Process control block (PCB)
- User address space
- User and kernel stacks to manage the call/return behaviour of the executing process.
- While process is running, processor registers are controlled by that process
- Contents of these registers are saved when the process is not running

Thread Model (contd.)



In a multithreaded environment:

- Single process control block
- User address space
- Separate stacks for each thread
- Separate control block for each thread containing register image, priority, and other thread related state information.

All threads of a process:

- Share the state and resources of the process
- Reside in the same address space and have access to the same code/data.
- When one thread alters an item of data in memory, other threads see the results if and when they access that item.
- If a thread opens a file with read privileges, other threads in same process can also read from that file.

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**

    Find areas that can be divided into separate concurrent tasks and thus can run in parallel on individual cores
  - **Balance**

    While identifying tasks that can run in parallel need to make sure that the tasks perform equal work
  - **Data splitting**

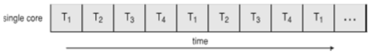
    Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate tasks
  - **Data dependency**

    Data accessed by tasks must be examined for dependencies between two or more tasks. For cases of dependencies, the execution of tasks must be synchronized to accommodate data dependency
  - **Testing and debugging**

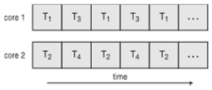
    When programming is running in parallel on multiple cores, there are many paths. Testing and debugging of these paths is inherently more difficult

Multicore Programming

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency
- On a single core system, concurrency means inter-leaving of threads execution.
- On a multi-core system, concurrency → threads can run in parallel.
- **Concurrent execution on single-core system:**



Parallelism on a multi-core system



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

Thread Functionality

A Thread has:

- Execution State
- May synchronise with one another

Thread Functionality (contd.)

Thread States:

- **Spawn:**
  - Typically when a new process is spawned, a thread for that process is also spawned.
  - A thread within a process may spawn another thread within the same process, providing an instruction pointer and arguments for the new thread.
  - New thread is provided with its own register context and stack space. Placed on the ready list.
- **Block:**
  - When a thread needs to wait for an event, it will block (saving its user registers, PC and SP).
  - Processor may now turn to the execution of another ready thread.
- **Unblock:**
  - When the event for which a thread is blocked occurs, the thread is moved to the ready state.
- **Finish:**
  - When a thread completes, its register context and stacks are de-allocated.

Thread Functionality (contd.)

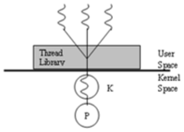
Thread synchronisation:

- All threads of a process share the same address space and other resources, such as open files.
  - Alteration of a resource by one thread affects the environment of the other threads in the same process.
- Necessary to synchronise the activities of various threads so that they do not interfere with each other or corrupt data structures.
  - **For example,** two threads try to add an element to a double linked list, one element may be lost or the list may end up malformed.
- **Many different methods** and techniques - same as for processes. We will talk about these later on in next chapters.

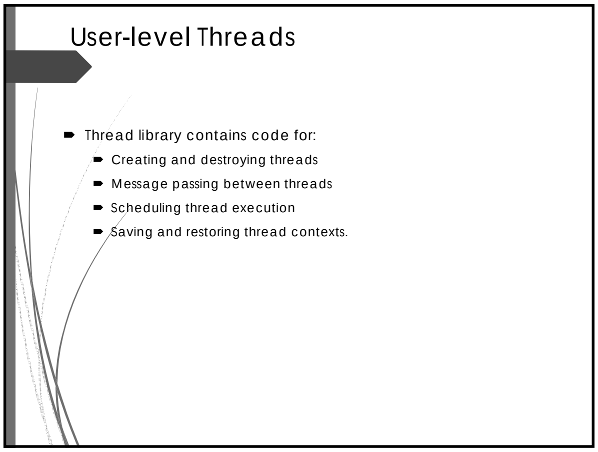
User-level Threads & Kernel-level Threads

Two broad categories of implementation:

- **User-level threads** - management done by user-level threads library
  - Thread management is done by the application ➔ kernel is not aware of the existence of the threads

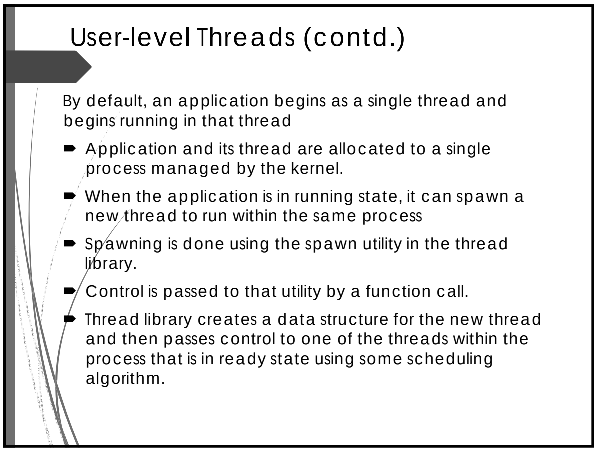


  - Any application can be programmed to be multithreaded by using a thread library.
  - Three primary thread libraries:
    - POSIX **Pthreads**
    - Windows threads
    - Java threads



User-level Threads

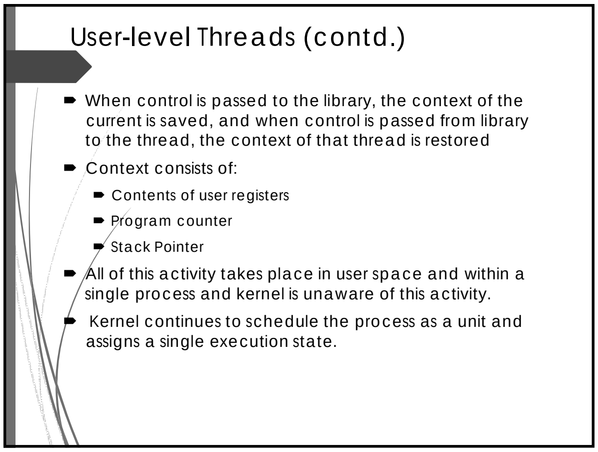
- Thread library contains code for:
  - Creating and destroying threads
  - Message passing between threads
  - Scheduling thread execution
  - Saving and restoring thread contexts.



User-level Threads (contd.)

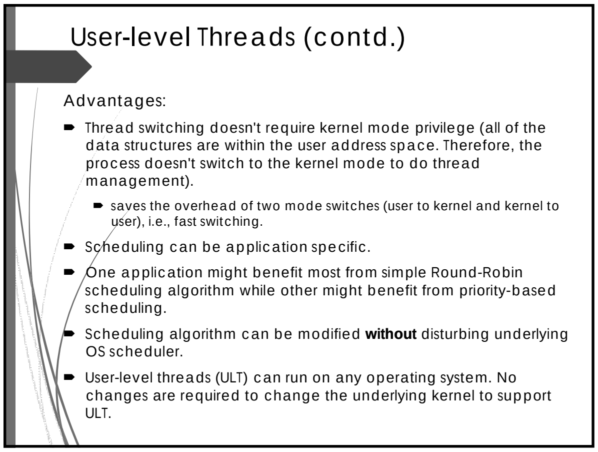
By default, an application begins as a single thread and begins running in that thread

- Application and its thread are allocated to a single process managed by the kernel.
- When the application is in running state, it can spawn a new thread to run within the same process
- Spawning is done using the spawn utility in the thread library.
- Control is passed to that utility by a function call.
- Thread library creates a data structure for the new thread and then passes control to one of the threads within the process that is in ready state using some scheduling algorithm.



User-level Threads (contd.)

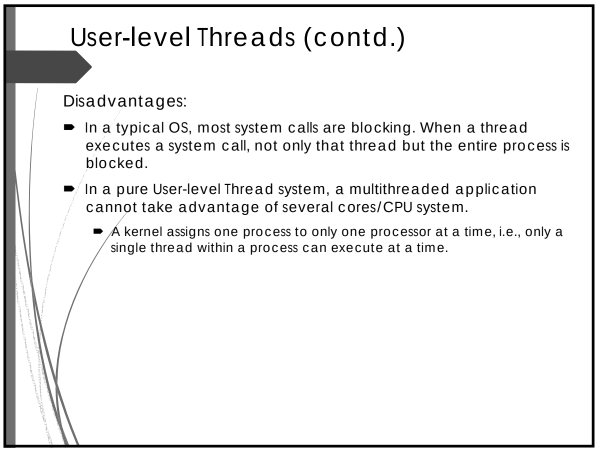
- When control is passed to the library, the context of the current is saved, and when control is passed from library to the thread, the context of that thread is restored
- Context consists of:
  - Contents of user registers
  - Program counter
  - Stack Pointer
- All of this activity takes place in user space and within a single process and kernel is unaware of this activity.
- Kernel continues to schedule the process as a unit and assigns a single execution state.



User-level Threads (contd.)

Advantages:

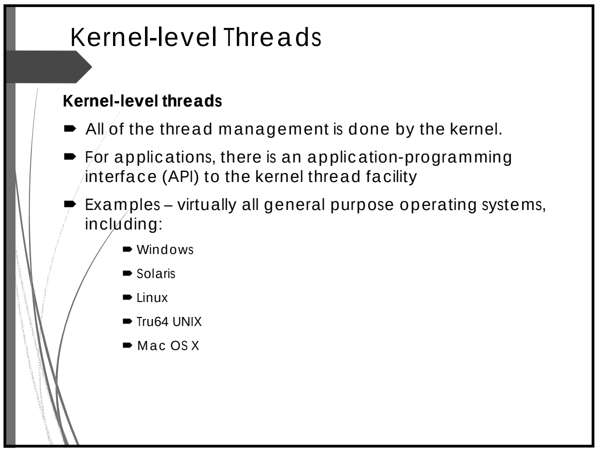
- Thread switching doesn't require kernel mode privilege (all of the data structures are within the user address space. Therefore, the process doesn't switch to the kernel mode to do thread management).
  - saves the overhead of two mode switches (user to kernel and kernel to user), i.e., fast switching.
- Scheduling can be application specific.
- One application might benefit most from simple Round-Robin scheduling algorithm while other might benefit from priority-based scheduling.
- Scheduling algorithm can be modified **without** disturbing underlying OS scheduler.
- User-level threads (ULT) can run on any operating system. No changes are required to change the underlying kernel to support ULT.



User-level Threads (contd.)

Disadvantages:

- In a typical OS, most system calls are blocking. When a thread executes a system call, not only that thread but the entire process is blocked.
- In a pure User-level Thread system, a multithreaded application cannot take advantage of several cores/CPU system.
  - A kernel assigns one process to only one processor at a time, i.e., only a single thread within a process can execute at a time.



Kernel-level Threads

Kernel-level threads

- All of the thread management is done by the kernel.
- For applications, there is an application-programming interface (API) to the kernel thread facility
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Kernel-level Threads (contd.)

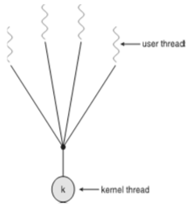
- An application can be programmed to be multithreaded.
- All of the threads within an application are supported within a process.
- Kernel maintains the context information for the process as a whole and that of the individual threads.
- Scheduling by the kernel is done on a thread basis.
- Overcomes two main drawbacks for the User-Level Threads:
  1. Kernel can simultaneously schedule multiple threads from the same process on multiple processors.
  2. If one thread in a process is blocked, the kernel can schedule another thread of the same process.

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

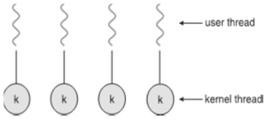
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



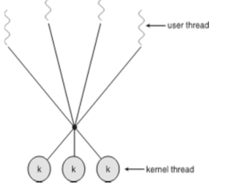
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



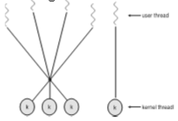
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Two-level Model

- One variation of many-to-many model is to multiplex many user-level threads to a smaller or equal number of kernel threads but also allow a user-level thread to be bound to a kernel thread, for example: Solaris
- Thread creation, scheduling and synchronisation of threads within an application are done completely in the user space.
- The programmer has the liberty of adjusting the number of Kernel-Level Threads for a particular application and machine to achieve the best overall results.
- Multiple threads within the same application can run in parallel on multiple processors.
- Therefore, a blocking system call need not block the entire process.
- A proper design can combine the advantages of pure User or Kernel-Level Threads while minimising the disadvantages.
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Two-level Model

Important Questions:

1. *Does the blocking of thread results in the blocking of the entire process?*
2. *Does this prevent execution of any other thread in the same process even if that other thread is in a ready state?*

- Answer depends on whether it is a User-Level Threading or the Kernel-Level Threading. More effect in User-level threading.
- **Example:** Assume 2 processes, P0 and P1 such that process P1 is executing in its thread 3. Possible scenarios:
  1. Application executing in Thread 3 makes a system call (e.g., I/O call is made) ➔ P1 blocks
     - Control is transferred to the kernel.
     - Kernel invokes the I/O action and places process P1 in **wait state**.
     - Transfers control to process P0
     - According to the data structure maintained by the thread library, Thread 3 of process P1 is still in **running** state.
     - This process is not actually in running state but is perceived in the running state.
  2. A clock interrupt passes control to the kernel.
     - Kernel determines that process P1 has exhausted its time quantum
     - Kernel places process P1 in the **ready** state and switches to process P0
     - According to the data structure maintained by the thread library, thread 3 of process P1 is still in **running** state.

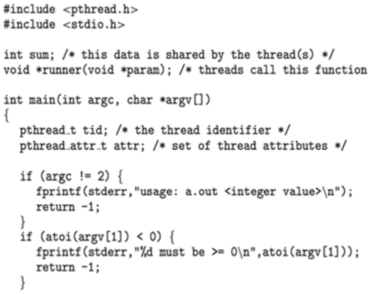
Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

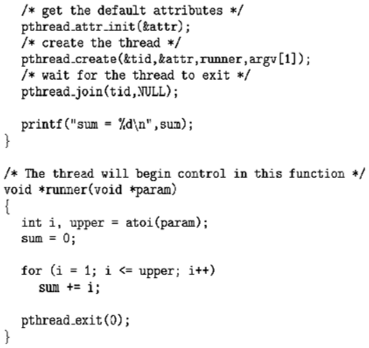
- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example



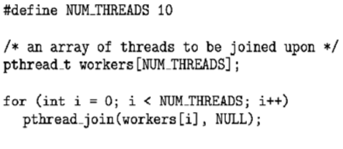
```
#include <pthread.h>
#include <stdio.h>

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;
  }
```

Pthreads Example (Cont.)

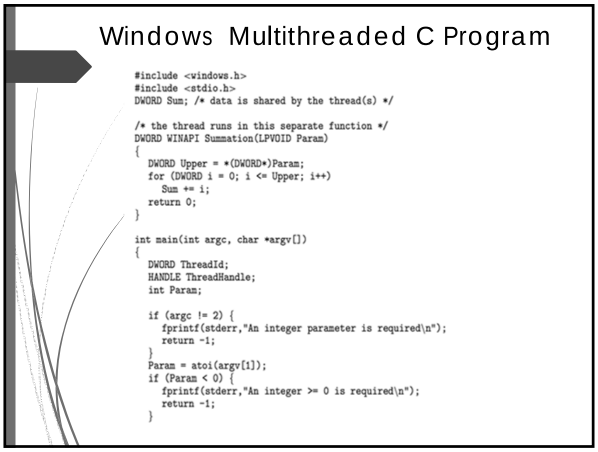


```
  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

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

/* 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

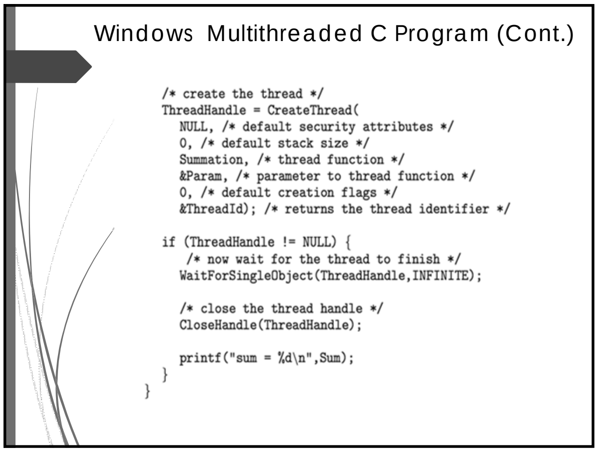


```
#define NUM_THREADS 10

/* 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);
```



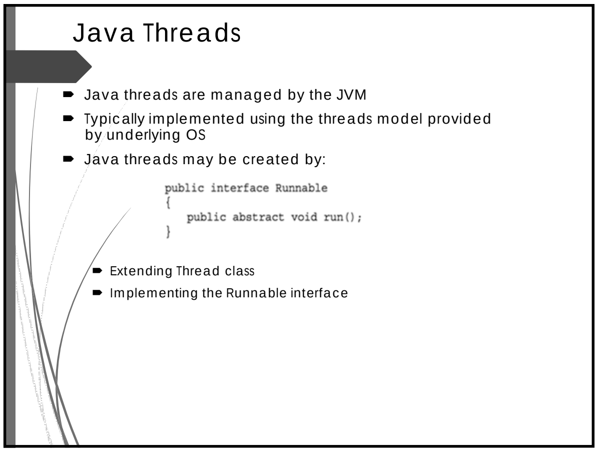
Windows Multithreaded C Program

```
#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;
  }
```



Windows Multithreaded C Program (Cont.)

```
  /* 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);
  }
}
```

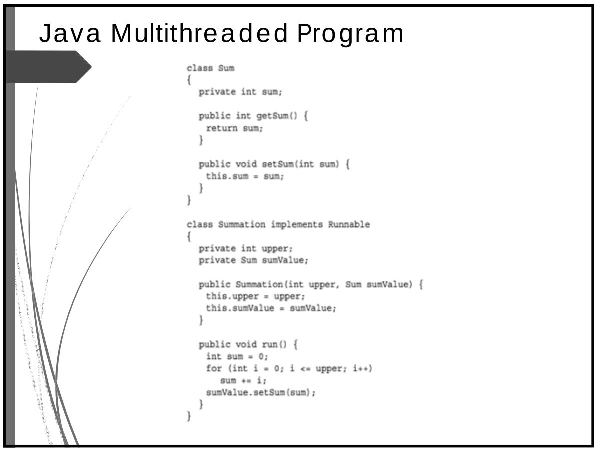


Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

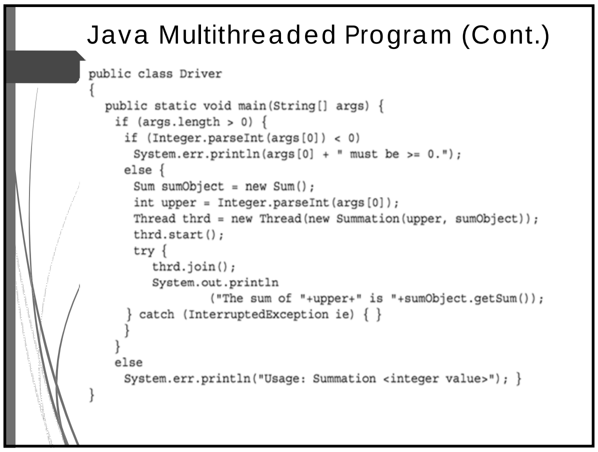
```
public interface Runnable
{
    public abstract void run();
}
```

  - Extending Thread class
  - Implementing the Runnable interface



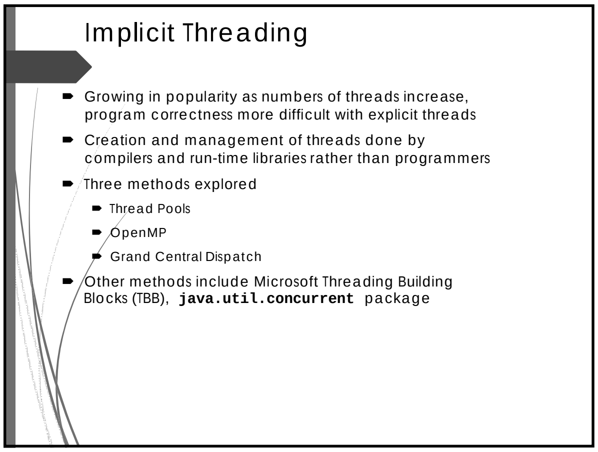
Java Multithreaded Program

```
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);
  }
}
```



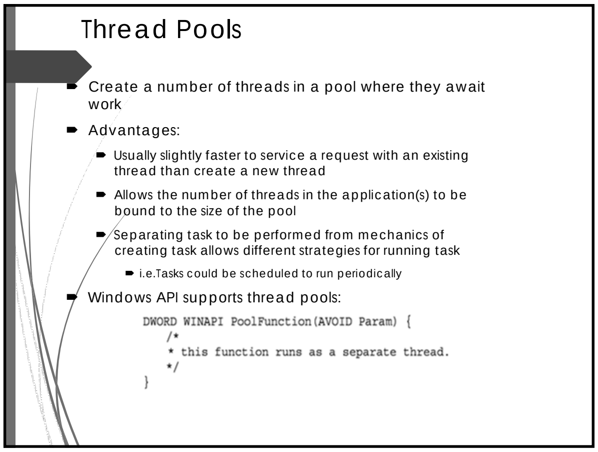
Java Multithreaded Program (Cont.)

```
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

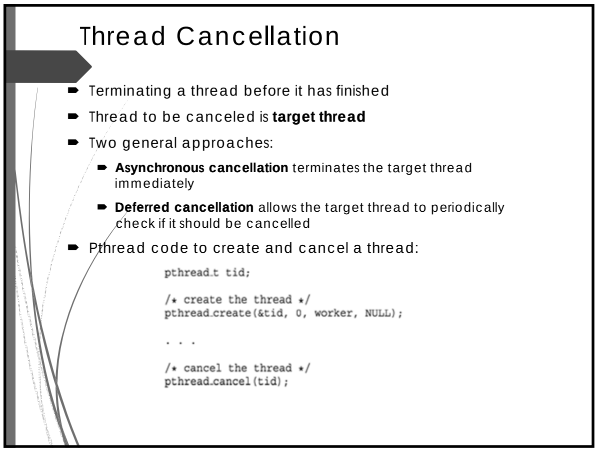
- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other 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:

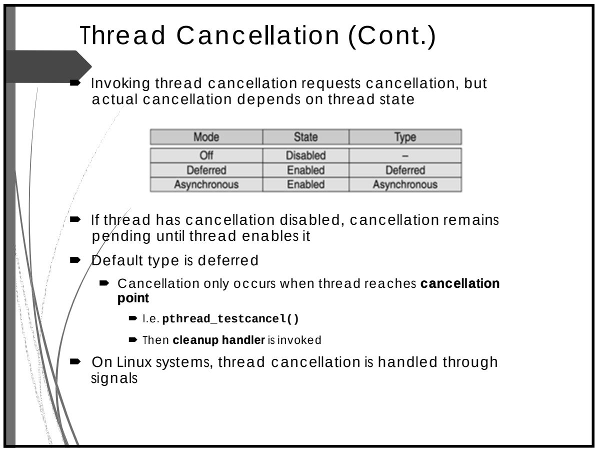
```
DWORD WINAPI PoolFunction(AVOID Param) {
   /*
    * this function runs as a separate thread.
    */
}
```



Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **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);
```

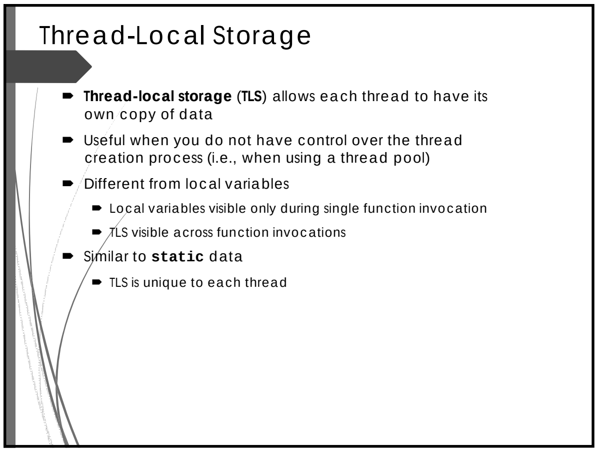


Thread Cancellation (Cont.)

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

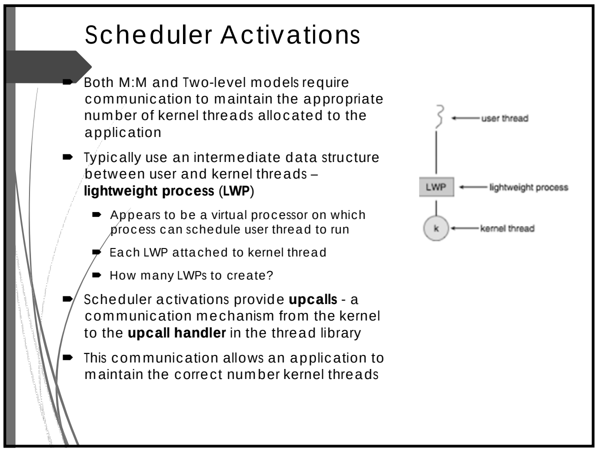
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- 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



Thread-Local Storage

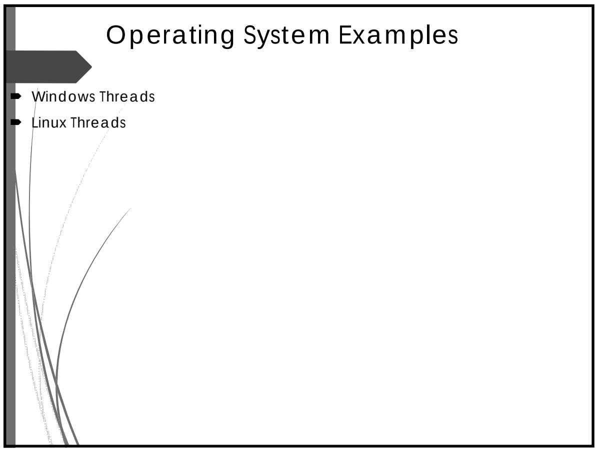
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible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



Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads

user thread
LWP
lightweight process
k
kernel thread



Operating System Examples

- Windows Threads
- Linux Threads

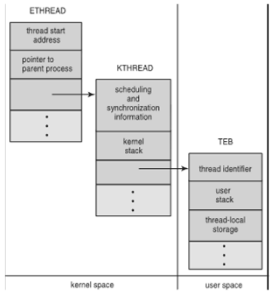
Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- 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.)

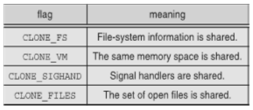
- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - 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 ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - 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.

- **`struct task_struct`** points to process data structures (shared or unique)