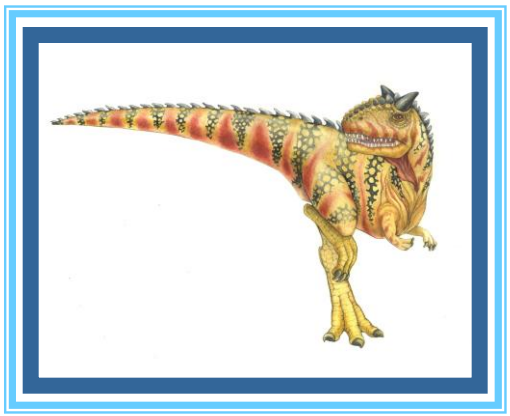
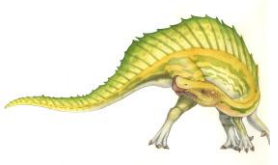
Chapter 4: Threads & Concurrency





Outline

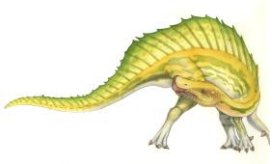
- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples





Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Design multithreaded applications using the Pthreads, Java and Windows threading APIs

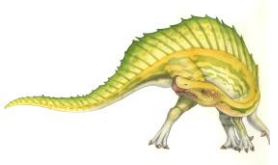


Troy University Resources



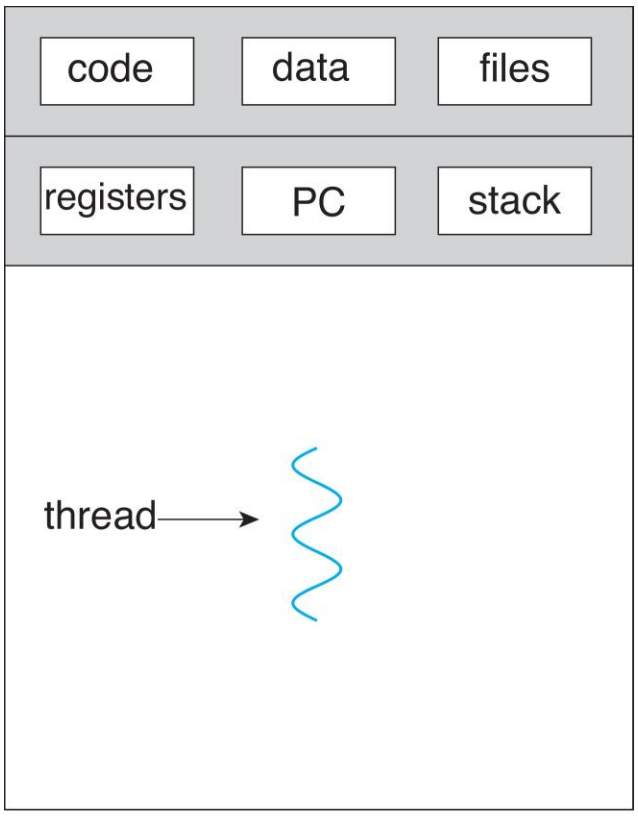
Motivation

- 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

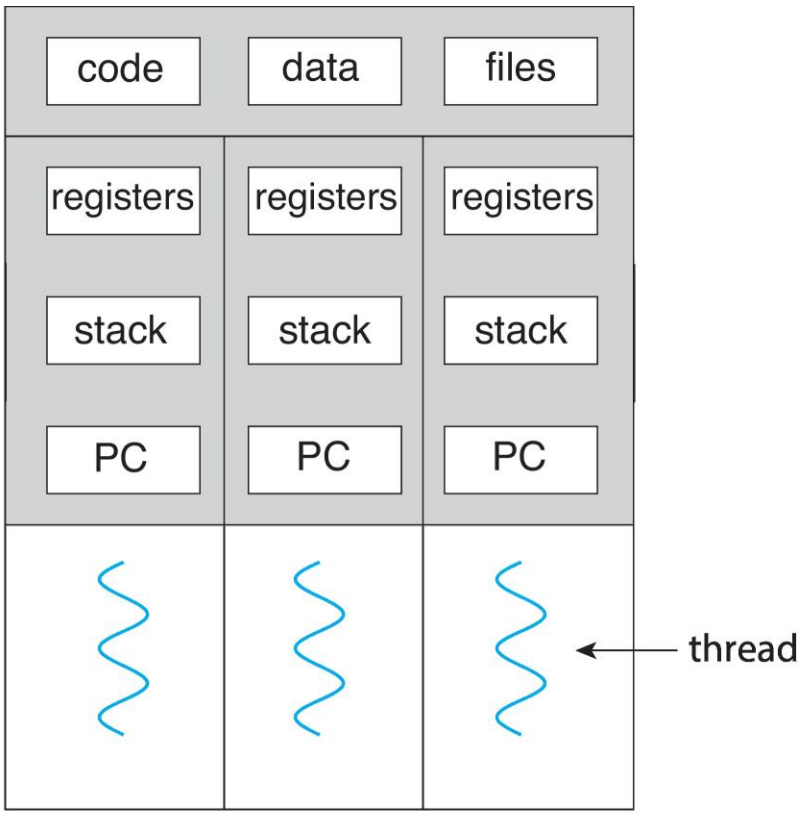




Single and Multithreaded Processes



single-threaded process



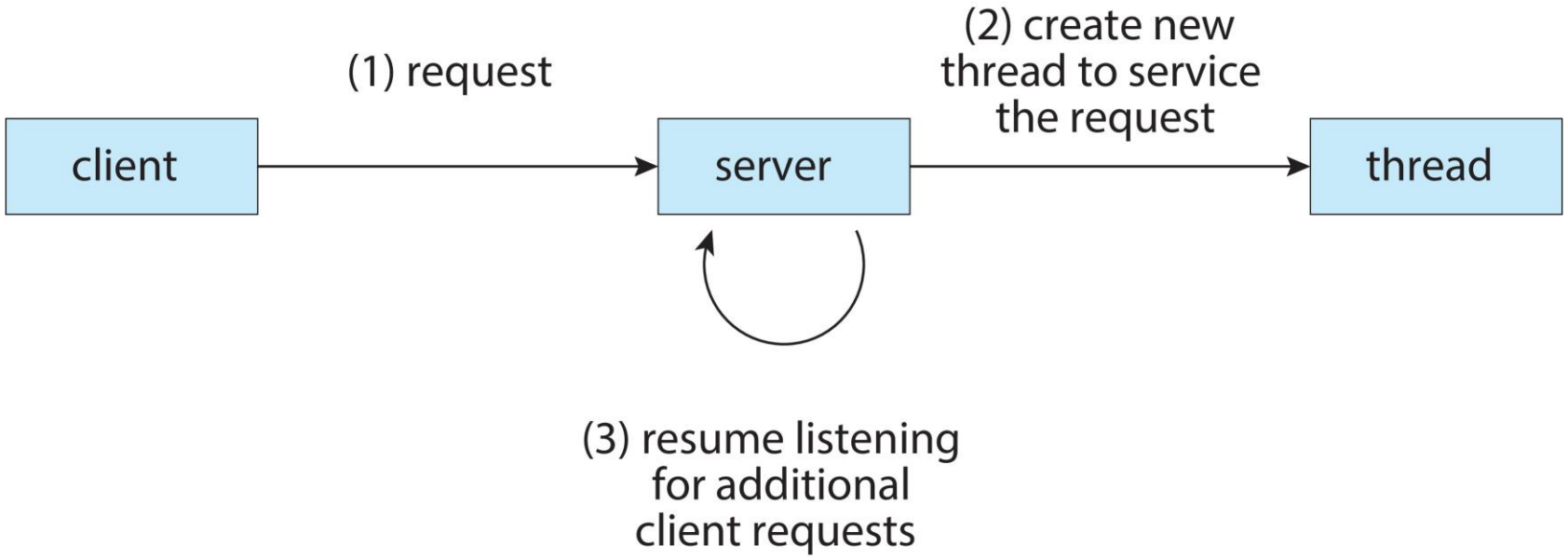
multithreaded process



Troy University Resources



Multithreaded Server Architecture

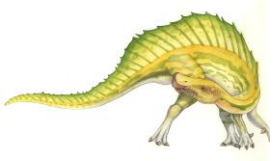






Benefits

- **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: creating threads and switching between threads in one process often costs less than creating or switching between separate processes**
- **Scalability –** process can take advantage of multicore architectures

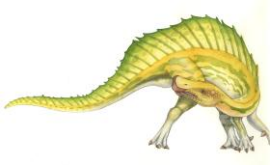


Troy University Resources



Multicore Programming

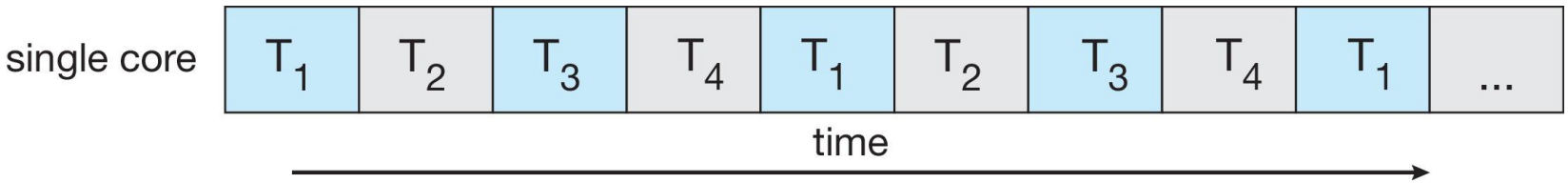
- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency



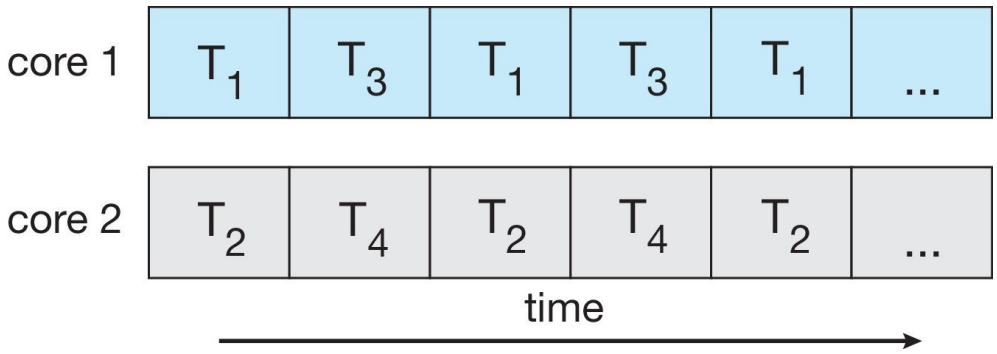


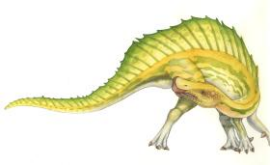
Concurrency vs. Parallelism

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



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



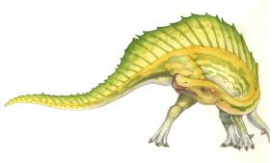


Troy University Resources



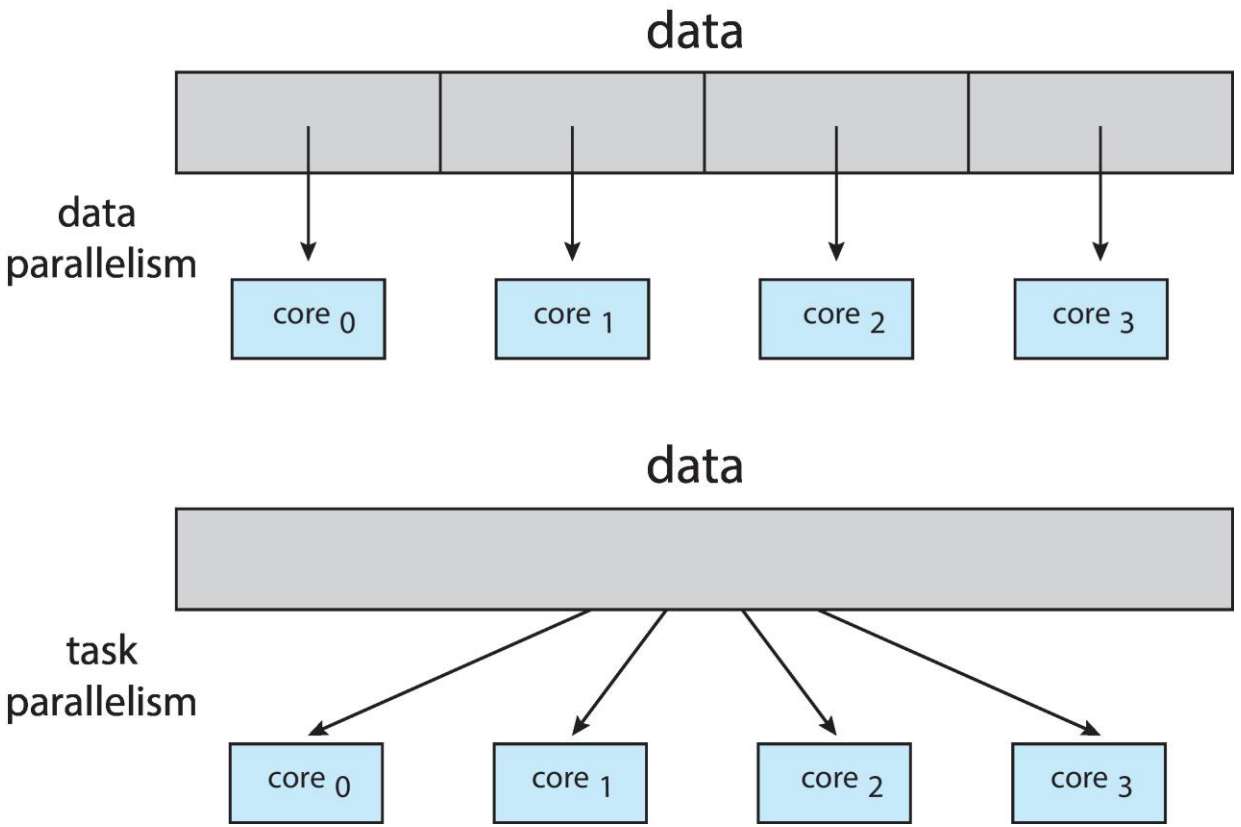
Multicore Programming

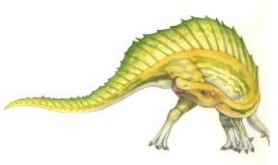
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation





Data and Task Parallelism







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

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

- That is, if 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*

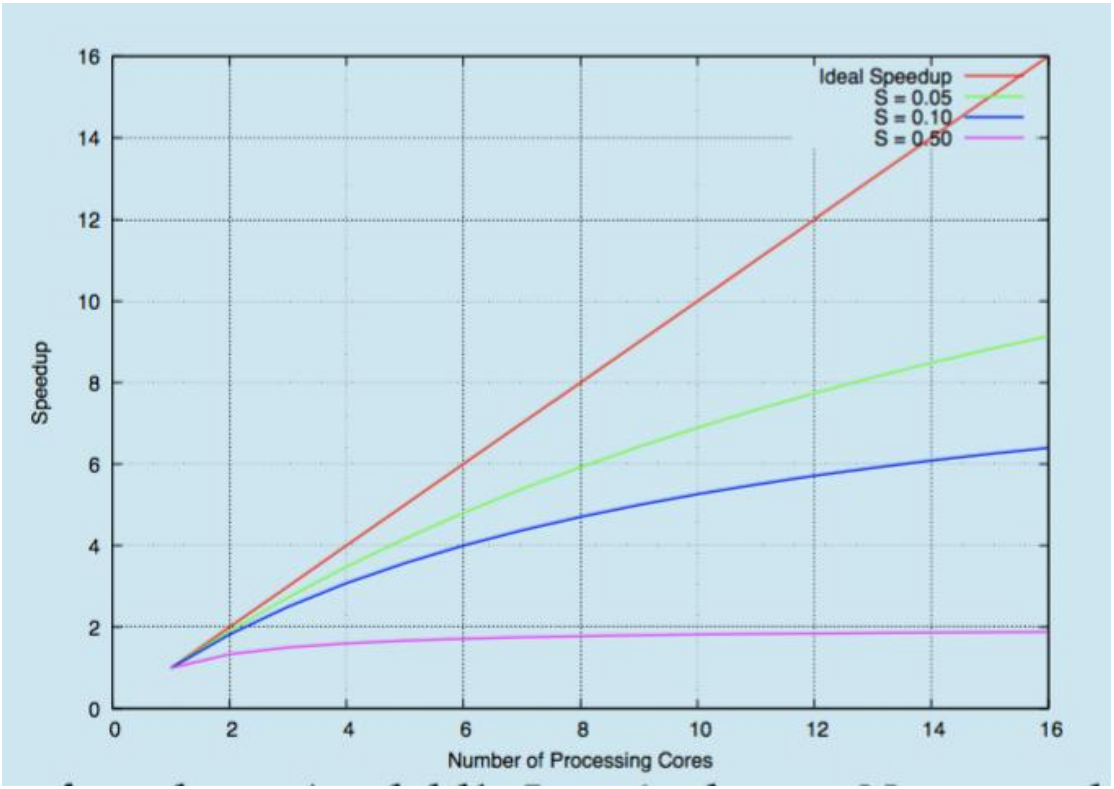
Serial portion of an application has disproportionate effect on performance gained by adding additional cores

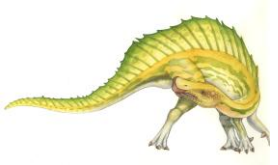
- Assumes a fixed workload and ideal parallel execution. Communication, synchronization and other overheads can reduce the actual speedup.





Amdahl's Law



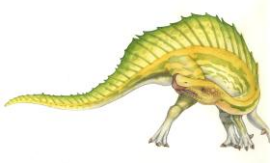


Troy University Resources



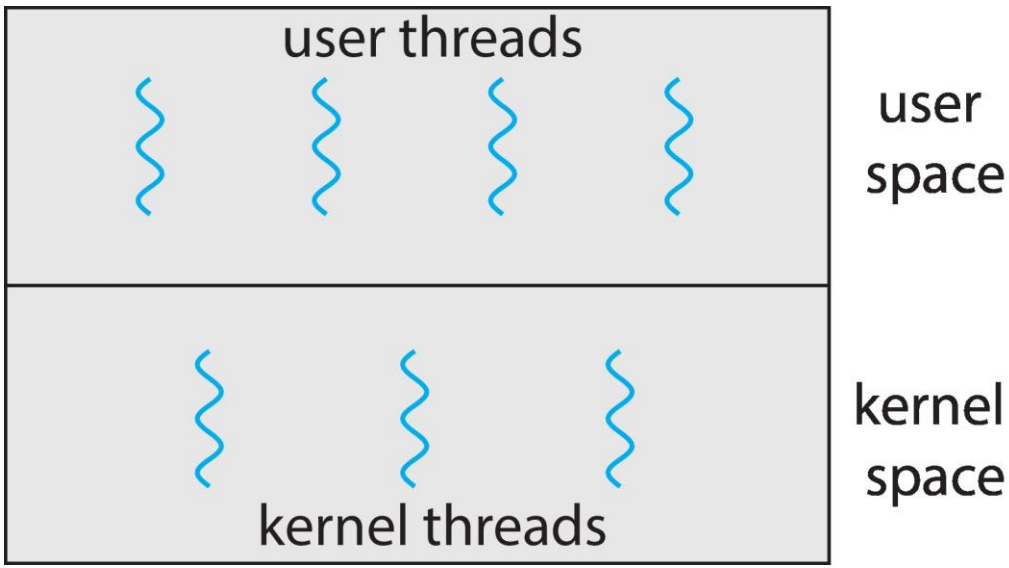
User Threads and Kernel Threads

- User-level thread abstractions come from a runtime or thread library.
- The kernel schedules kernel-supported tasks onto logical CPUs.
- An API name alone does not determine the mapping between user and kernel threads.
- For example, Linux Pthreads normally use a one-to-one mapping through NPTL.
- A runtime may instead multiplex many user threads over a smaller set of kernel threads.
- The next slides compare many-to-one, one-to-one and many-to-many mappings.





User and Kernel Threads

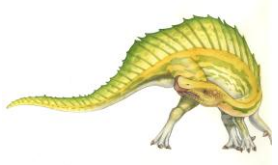






Multithreading Models

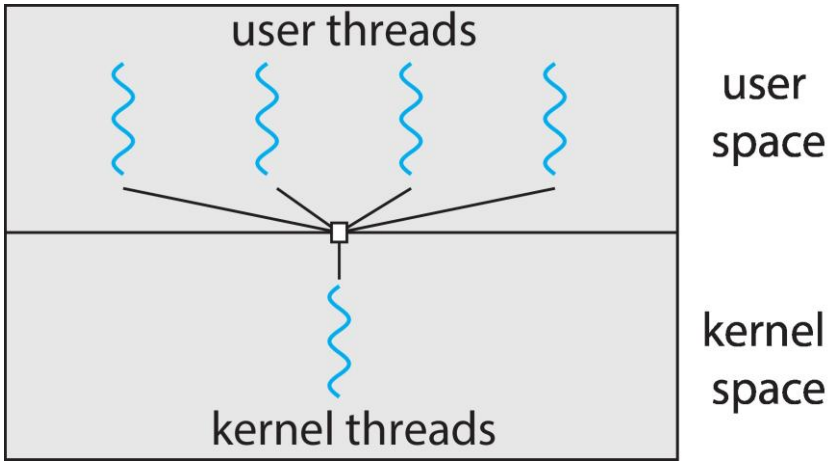
- Many-to-One
- One-to-One
- Many-to-Many

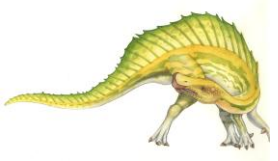




Many-to-One

- Many user-level threads mapped to single kernel thread
- A blocking operation on the sole kernel thread can stop progress for every user thread mapped to it
- Only one mapped user thread can execute at a time because the model has one kernel scheduling context
- Historical examples of this mapping:
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



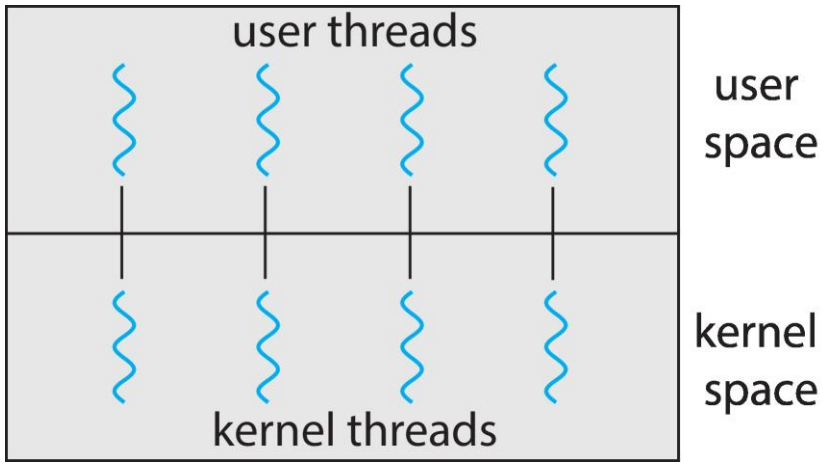


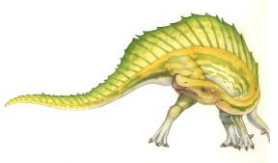
Troy University Resources



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

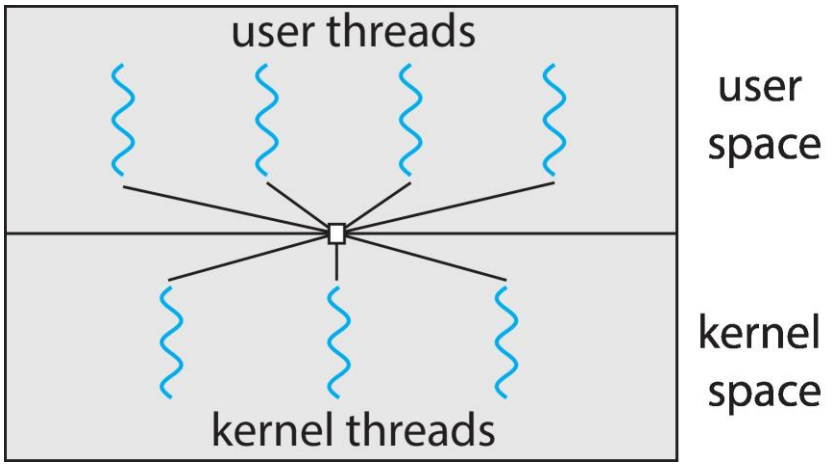


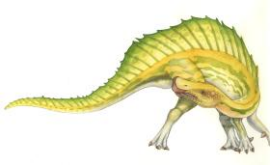




Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- The runtime schedules user threads over a set of kernel-supported threads
- Historical example: Solaris before version 9
- Modern runtimes can also multiplex user threads over carrier threads



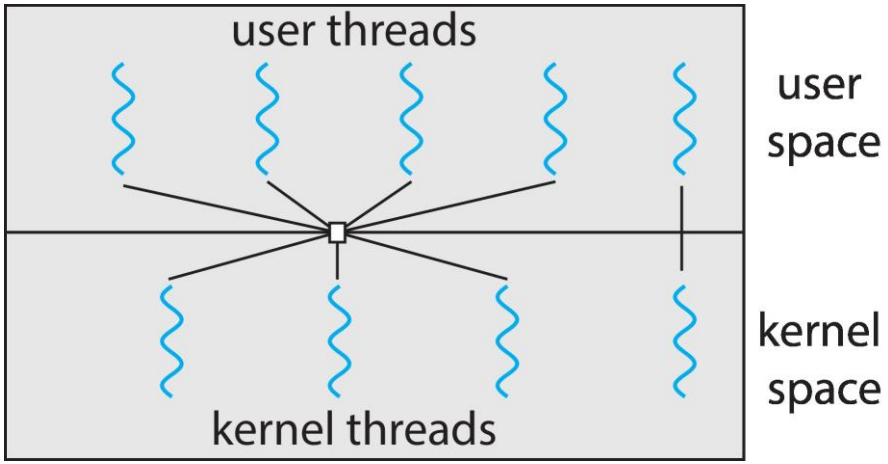


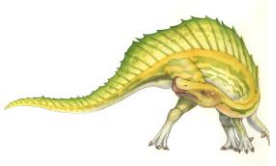
Troy University Resources



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread







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

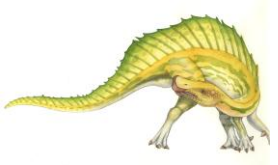


Troy University Resources



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 (Linux & Mac OS X)





Pthreads Example

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

#include <stdlib.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 */

   /* set the default attributes of the thread */
   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);
}
```



Troy University Resources



Pthreads Example (Cont.)

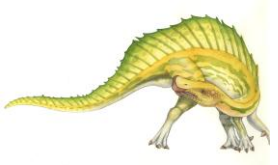
Check argc before argv[1], parse with strtol(), and check the thread API return codes.

```
/* The thread will execute 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);
}
```

With a 32-bit signed int sum, inputs from 0 to 65535 avoid overflow. Larger sums need a wider type. A successful join makes the completed sum visible to the main thread.





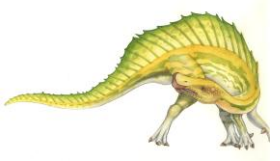
Pthreads Code for Joining 10 Threads

```
#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);
```

- workers[] must contain valid joinable IDs from successful pthread_create() calls.
- Join each successfully created thread exactly once and check the return code.
- Do not join uninitialized array entries.





Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

Use a DWORD parameter and keep it alive until the worker finishes. For a 32-bit DWORD sum, accept 0 to 92681. Check CreateThread() and wait results.





Windows Multithreaded C Program (Cont.)

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[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 */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

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

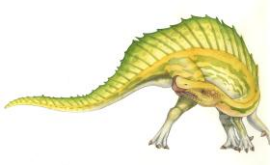




Java Threads

- The JVM manages Java threads. These examples use platform threads backed by OS threads.
- Runnable describes a task. A Thread or executor can execute it.
- Since Java 21, the JVM schedules virtual threads over carrier threads. They support many blocking tasks; CPU parallelism is still limited by available cores.

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





Java Threads

Implementing Runnable interface:

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

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```





Java Executor Framework

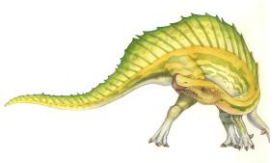
Executor is an interface. Its implementation decides how to execute submitted Runnable tasks.

```
public interface Executor
{
   void execute(Runnable command);
}
```

execute() does not itself require a new thread or asynchronous execution.

```
Executor service = new Executor;
service.execute(new Task());
```

```
ExecutorService service = Executors.newSingleThreadExecutor();
service.execute(new Task());
service.shutdown();
```





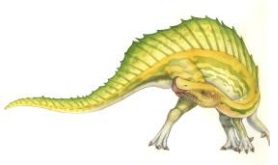
Java Executor Framework

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
   private int upper;
   public Summation(int upper) {
      this.upper = upper;
   }

   /* The thread will execute in this method */
   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;

      return new Integer(sum);
   }
}
```



Troy University Resources

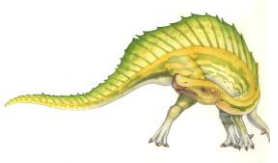


Java Executor Framework (Cont.)

```
public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

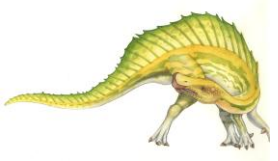
    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```





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
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks



Troy University Resources

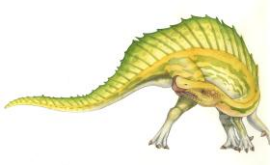


Thread Pools

- A thread pool reuses workers for submitted tasks.
- Reuse avoids creating a thread for each request.
- Pool limits and queue policies control concurrent work.
- Task submission separates the work from thread creation and scheduling.

Historical Windows QueueUserWorkItem() API excerpt:

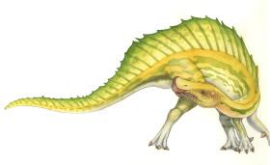
```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```





Java Thread Pools

- Three factory methods for creating thread pools in Executors class:
  - `static ExecutorService newSingleThreadExecutor()`
  - `static ExecutorService newFixedThreadPool(int size)`
  - `static ExecutorService newCachedThreadPool()`





Java Thread Pools (Cont.)

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
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 < numTasks; i++)
      pool.execute(new Task());

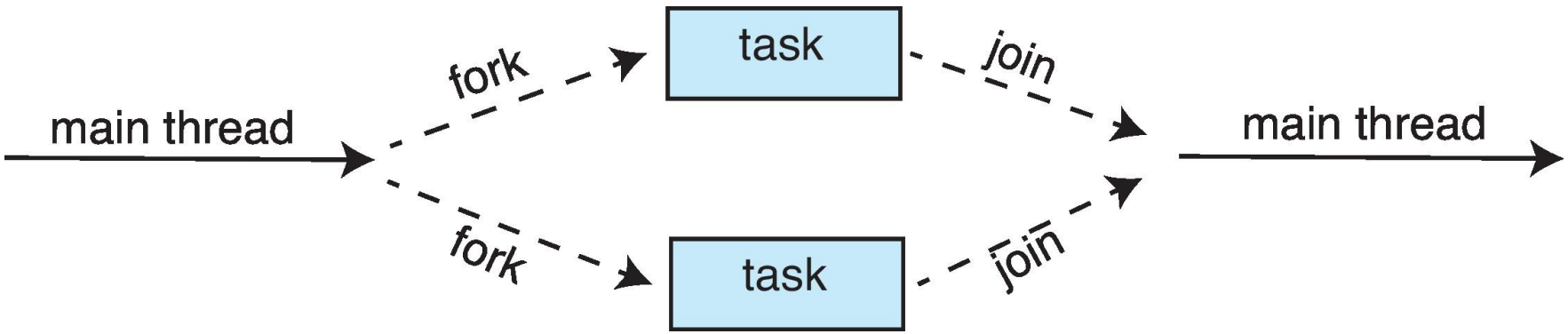
   /* Shut down the pool once all threads have completed */
   pool.shutdown();
}
```

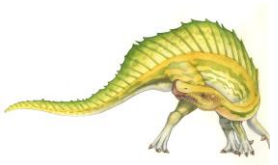




Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.







Fork-Join Parallelism

- General algorithm for fork-join strategy:

```
Task(problem)
   if problem is small enough
     solve the problem directly
   else
     subtask1 = fork(new Task(subset of problem)
     subtask2 = fork(new Task(subset of problem)

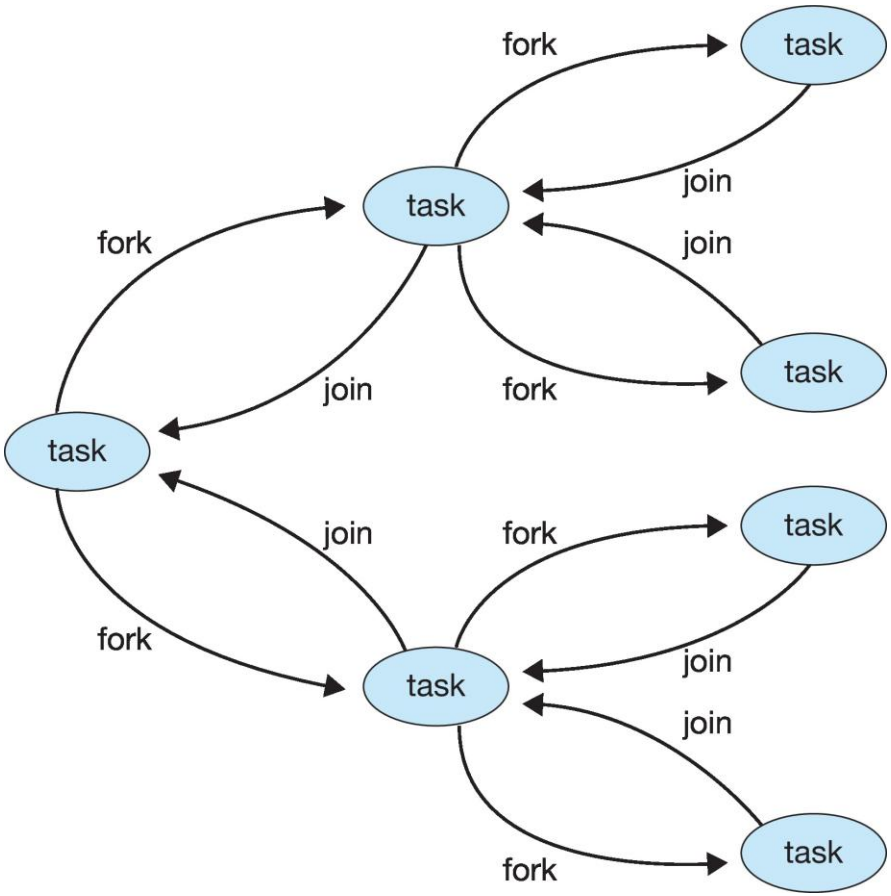
     result1 = join(subtask1)
     result2 = join(subtask2)

     return combined results
```





Fork-Join Parallelism





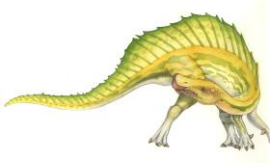
Troy University Resources



Fork-Join Parallelism in Java

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```





Fork-Join Parallelism in Java

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

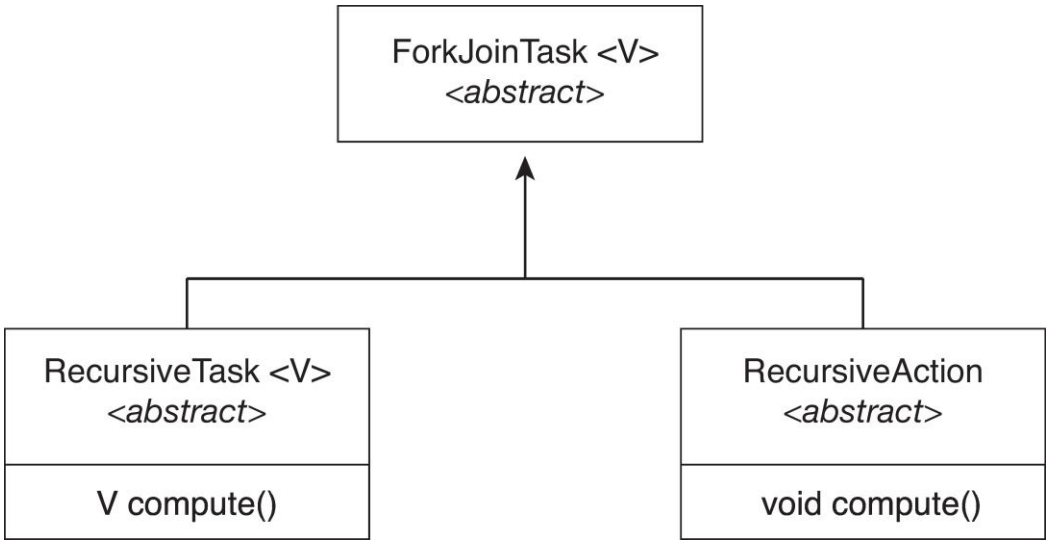


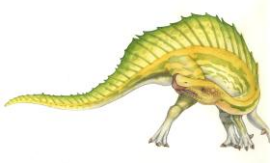
Troy University Resources



Fork-Join Parallelism in Java

- The **`ForkJoinTask`** is an abstract base class
- **`RecursiveTask`** and **`RecursiveAction`** classes extend **`ForkJoinTask`**
- **`RecursiveTask`** returns a result (via the return value from the **`compute()`** method)
- **`RecursiveAction`** does not return a result







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`

The runtime chooses the team size from settings such as OMP_NUM_THREADS and num_threads, subject to its limits

```
#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;
}
```

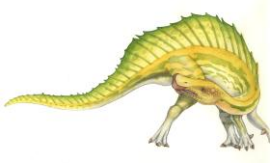


Troy University Resources



- Run the for loop in parallel

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



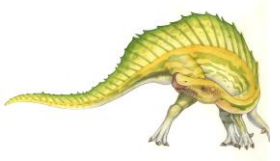


Grand Central Dispatch

- Apple technology for macOS and iOS operating systems
- Extensions to C, C++ and Objective-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

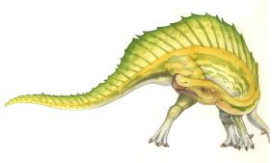


Troy University Resources



Grand Central Dispatch

- A serial dispatch queue executes one submitted task at a time, in FIFO order.
- The main queue is a special serial queue associated with the main thread.
- Applications may create additional serial or concurrent queues.
- A concurrent queue can begin multiple tasks without waiting for each earlier task to finish.
- Global queues include QoS classes USER_INTERACTIVE, USER_INITIATED, DEFAULT, UTILITY and BACKGROUND.
- C API names include QOS_CLASS_UTILITY and QOS_CLASS_BACKGROUND.





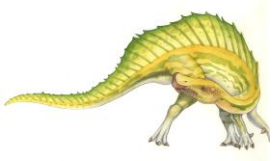
Grand Central Dispatch

- Swift expresses a submitted task as a closure.
- The image uses an older Swift dispatch API. Current Swift uses DispatchQueue and its async method.

```
let queue = dispatch_get_global_queue
   (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```

```
let queue = DispatchQueue.global(qos: .userInitiated)
queue.async {
    print("I am a closure.")
}
```





Intel Threading Building Blocks (TBB)

- Template library for designing parallel C++ programs
- A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

- The same for loop written using TBB with **`parallel_for`** statement:

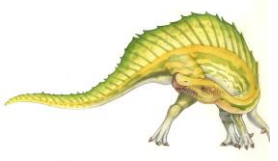
```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```





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

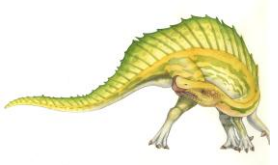


Troy University Resources



Semantics of fork() and exec()

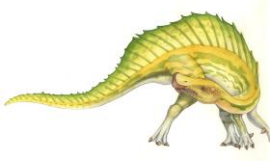
- POSIX fork() creates a child containing only the thread that called fork().
- The child inherits a copy of the process address space, including synchronization state.
- A lock held by another thread may remain locked even though that thread is absent in the child.
- After fork() in a multithreaded process, the child must restrict itself to async-signal-safe operations until exec().
- A successful exec() replaces the process image. The other threads of that process cease to exist.





Signal Handling

- A signal notifies a process or thread of an event.
- Each signal has a disposition: a default action, an installed handler or, when allowed, ignore.
- Default actions can terminate, stop or continue the process, or ignore the signal.
- A user-installed handler executes in the receiving thread's user context.
- SIGKILL and SIGSTOP cannot be caught, blocked or ignored.
- The delivery rules depend on whether a signal is directed to a process or a particular thread.

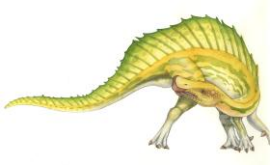


Troy University Resources



Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - 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





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);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```



Troy University Resources

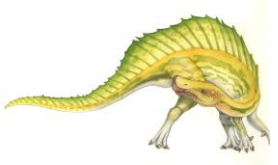


Thread Cancellation (Cont.)

- 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 Cancellation in Java

Interruption supports cooperative cancellation. Invoke worker.interrupt(); to request it.

```
Thread worker;

 . . .

/* set the interruption status of the thread */
worker.interrupt()
```

The worker checks its own interrupted status:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```

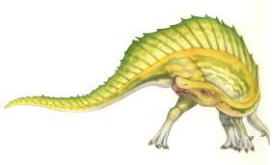
isInterrupted() leaves the flag set. Blocking calls may throw InterruptedException and clear it. Handle cancellation and release resources.





Thread-Local Storage

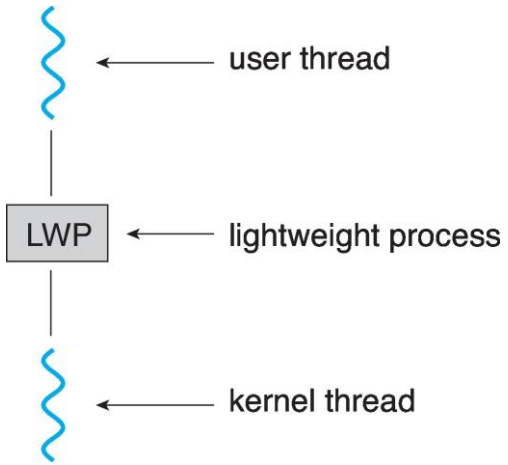
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible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

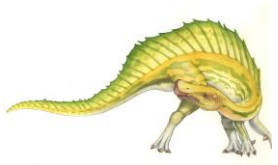




Scheduler Activations

- Scheduler activations are one historical mechanism for coordinating an M:M runtime with the kernel
- 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
- The thread library uses these notifications to adjust its scheduling decisions



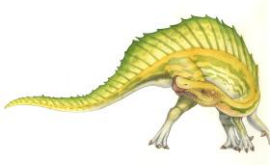


Troy University Resources



Operating System Examples

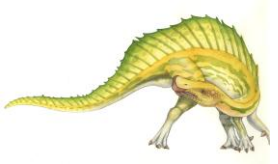
- Windows Threads
- Linux Threads





Windows Threads

- Windows API – primary API for Windows applications
- 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

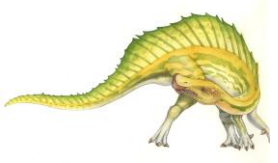


Troy University Resources



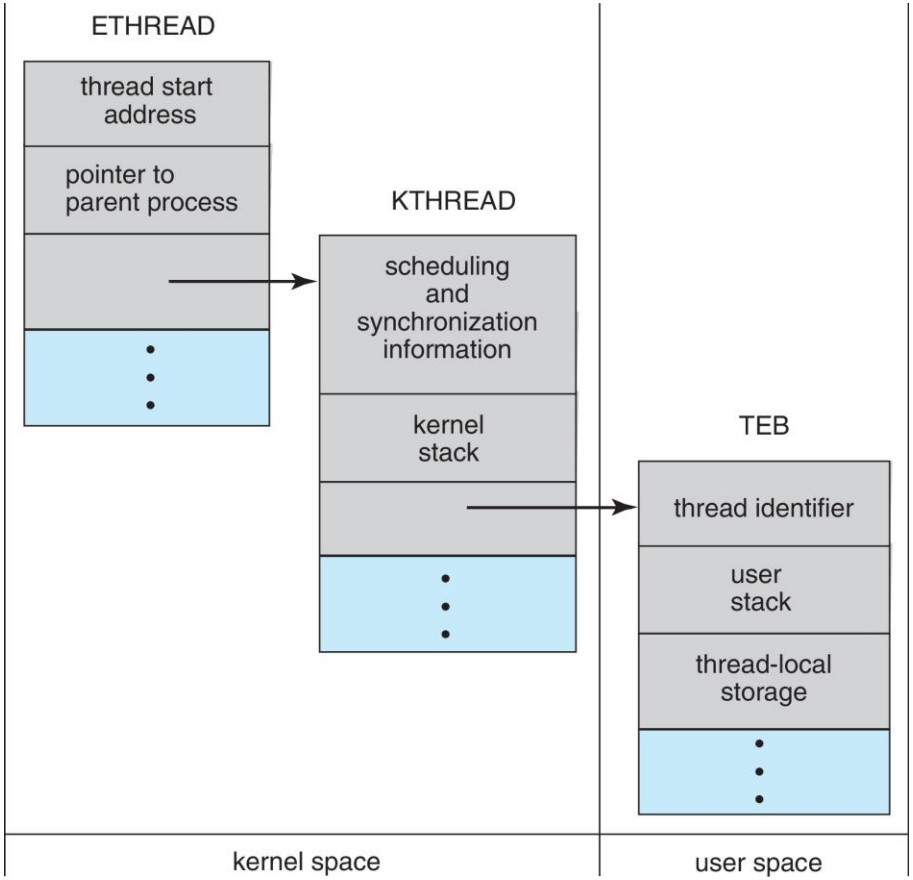
Windows Threads (Cont.)

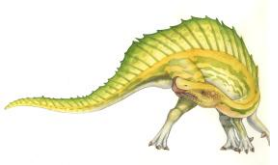
- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  -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





Windows Threads Data Structures





Troy University Resources



Linux Threads

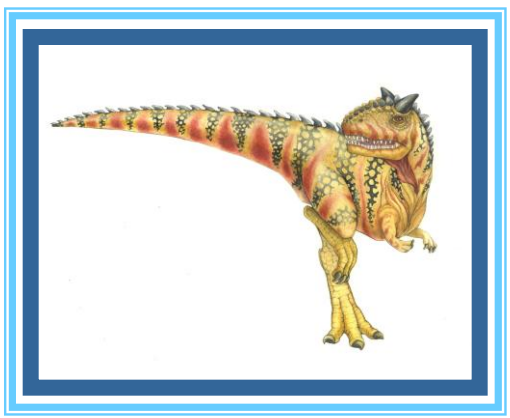
- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

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)



End of Chapter 4



Silberschatz, Galvin and Gagne ©2018