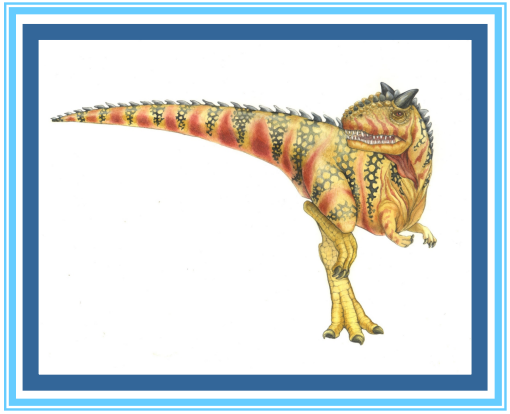
Chapter 4: Threads & Concurrency



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 designng 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

Threads

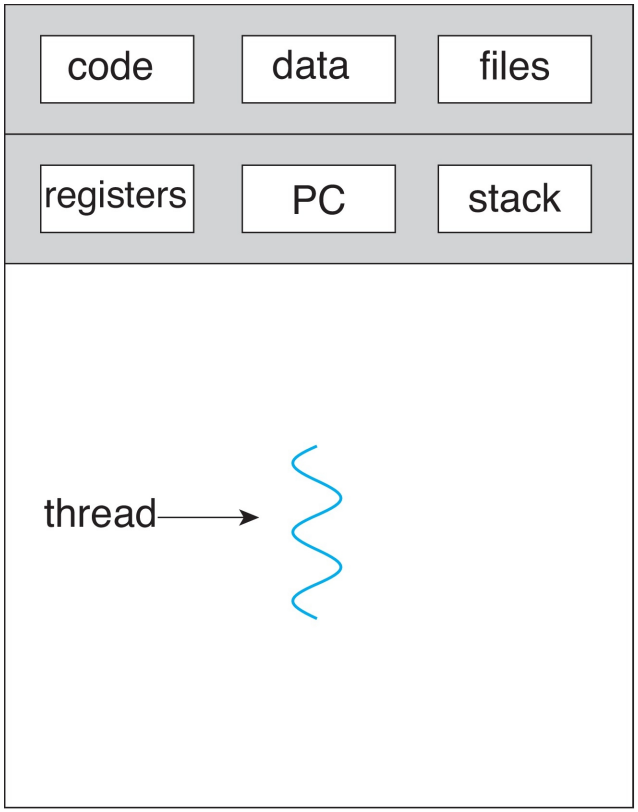
- A thread is a basic unit of CPU utilization. It consists of a thread ID, PC, a stack, and a set of registers.
- Traditional processes have a single thread of control. It is also called as **heavyweight** process. There is one program counter, and one sequence of instructions that can be carried out at any given time.
- A multi-threaded application 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 open files. Such processes are also called as **lightweight** processes.
- Process with multiple threads of control can do more than one task at a time.

Motivation

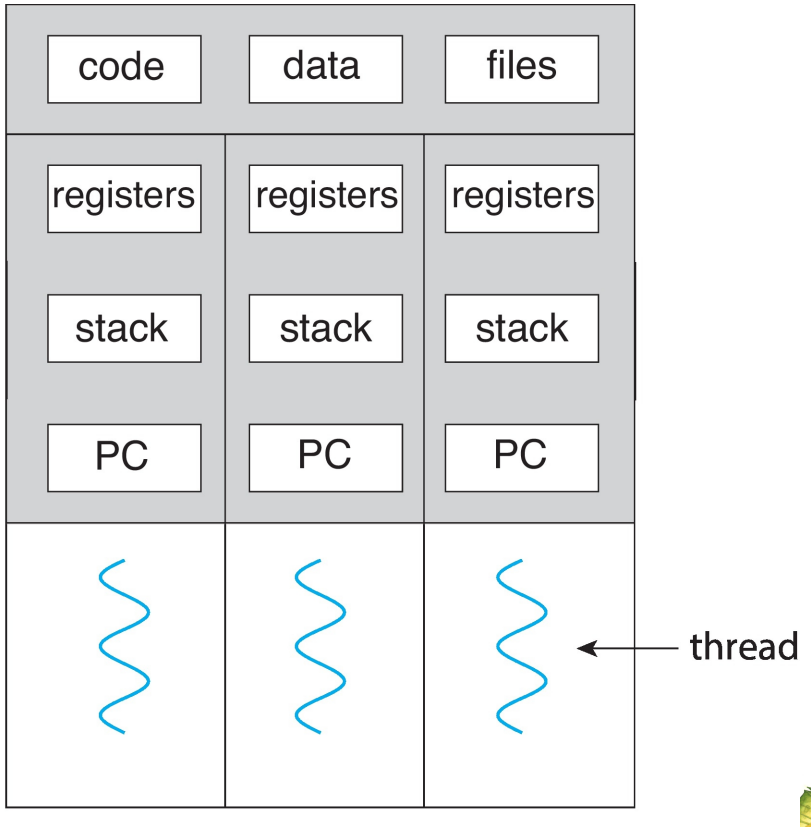
- Most modern applications are multithreaded
- If one of the tasks may block, the other tasks can proceed without blocking.
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads Eg. Word processor
  - Spell checking (background process)
  - process user input (foreground)
  - displaying graphics (another thread)
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Single and Multithreaded Processes

- In a heavyweight process, new processes are created to perform the work in parallel (multiple processes running). Each heavyweight process contains its own address space. Communication between these processes would involve additional communications mechanisms such as IPC, RPC, sockets or pipes.



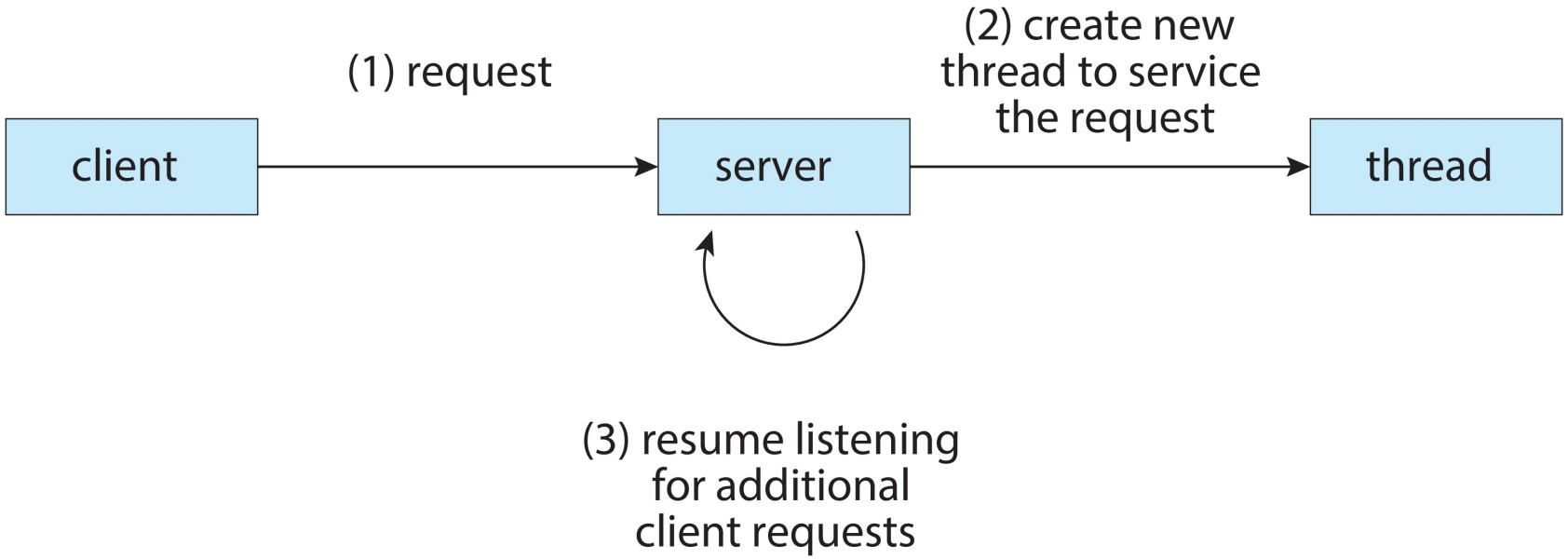
single-threaded process



multithreaded process

Multithreaded Server Architecture

Eg: Web server



Benefits

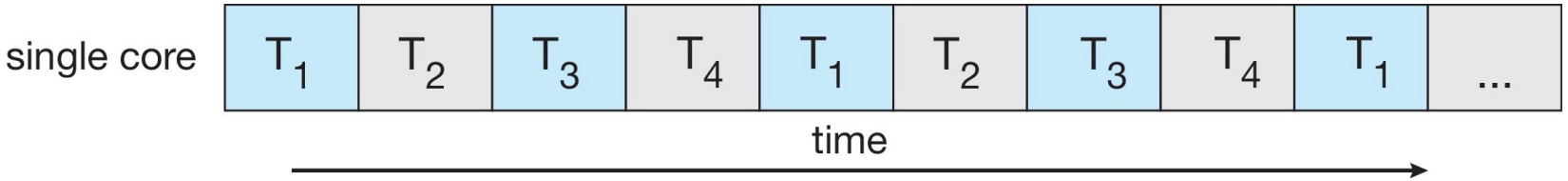
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- Multi threading allows a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- allows multiple tasks to be performed simultaneously in a single address space.
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- Creating and managing threads is much faster
- Context switching between threads takes less time.
- **Scalability / utilization of multiprocessor architecture –** with multitreading a process can take advantage of multicore architectures

Multicore Programming

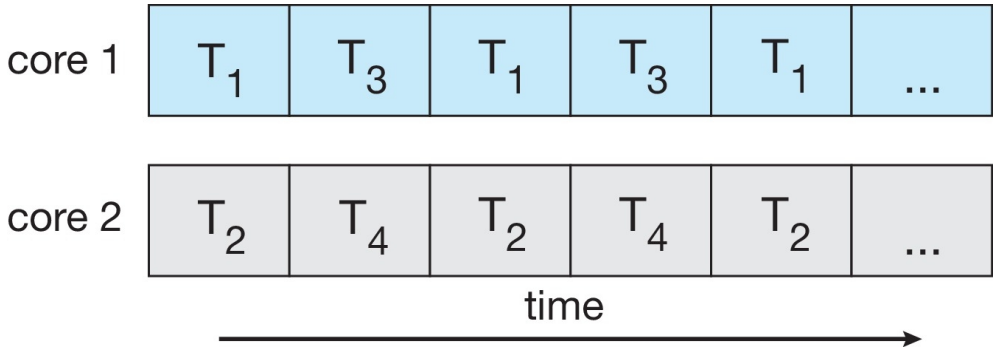
- A recent trend in computer architecture is to produce chips with multiple cores, or CPUs on a single chip.
- A multi-threaded application running on a traditional **single-core** chip, would have to execute the threads one after another.
- On a **multi-core chip**, the threads could be spread across the available cores, allowing true parallel processing.
- For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.

Concurrency vs. Parallelism

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



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



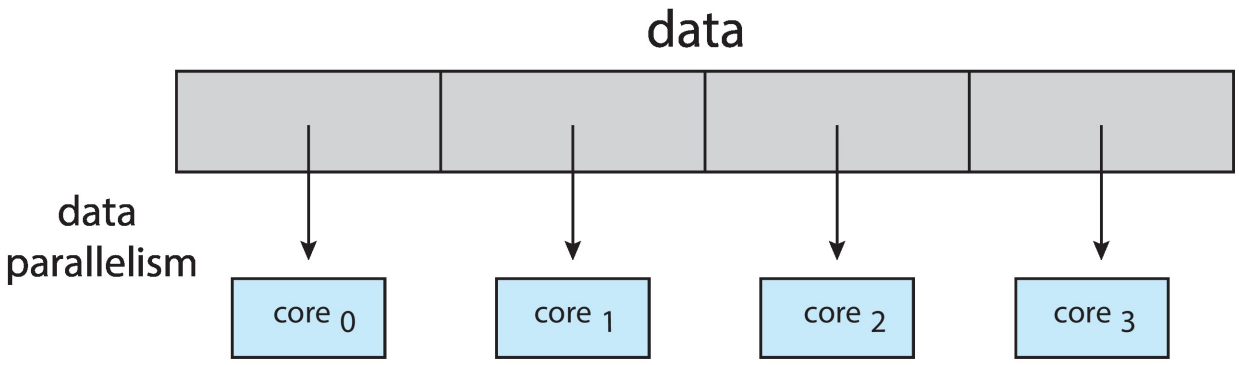
Multicore Programming

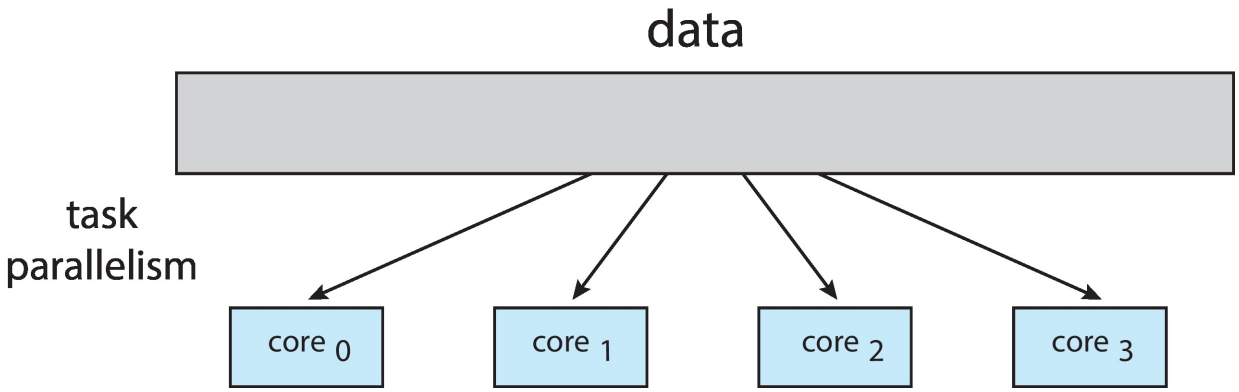
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities:** Examining applications to find activities that can be performed concurrently.
  - **Balance:** All threads should have same level of computation; do not create thread for trivial tasks
  - **Data splitting:** To prevent the threads from interfering with one another.
  - **Data dependency:** If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
  - **Testing and debugging:** Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

Data and Task Parallelism

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

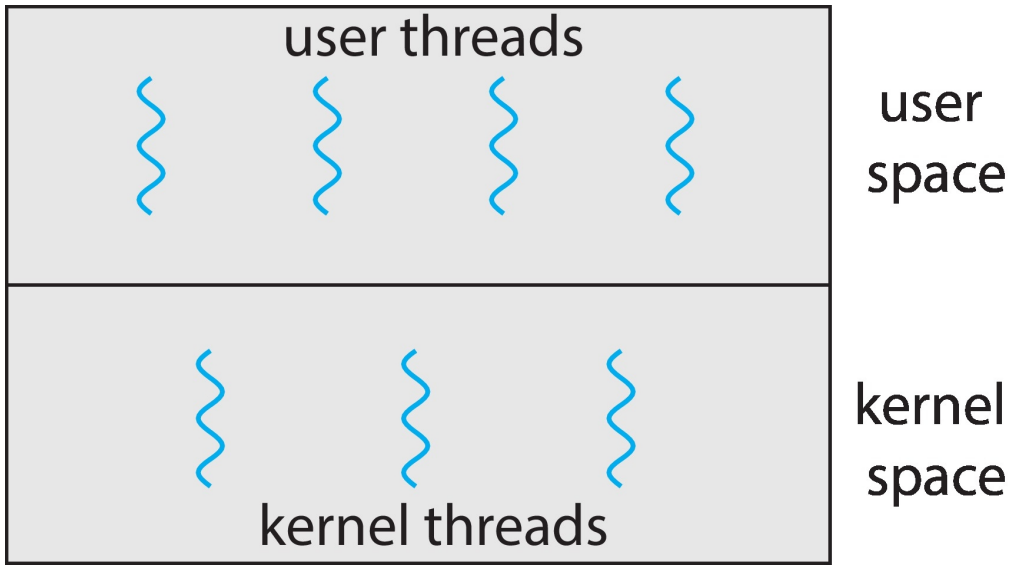




User Threads and Kernel Threads

- Two types of threads are managed in modern systems: **User and Kernel Threads**
- **User threads** - management done by user-level threads library. They are supported above the kernel.
- creation, scheduling and management is done by library without kernel support.
- Problem: If kernel is single threaded, any user level thread performing a blocking system call will block the entire process.
- Three primary thread libraries:
  - POSIX **Pthreads,** Windows threads, Java threads
- **Kernel threads** - Supported directly by the OS/Kernel allowing the kernel to perform multiple tasks simultaneously.
- Generally slower to create and manage than user threads.
- Examples – virtually all general -purpose operating systems, including:
  - Windows ,Linux, Mac OS X, iOS, Android

User and Kernel Threads



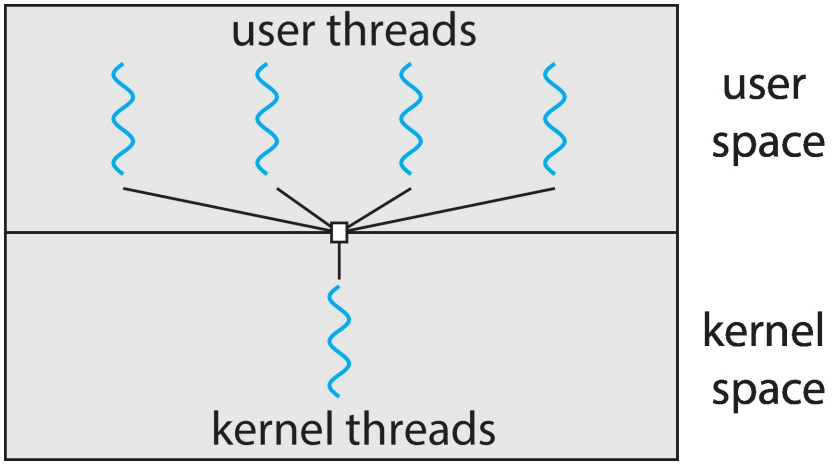
Multithreading Models

In a specific implementation, the user threads must be mapped to kernel threads, using one of the following models.

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

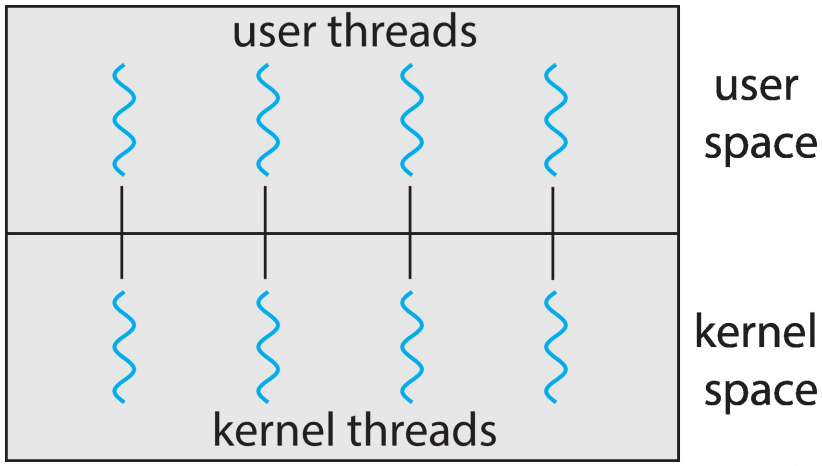
Many-to-One

- Thread management is handled by the thread library in user space, which is very efficient.
- 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**



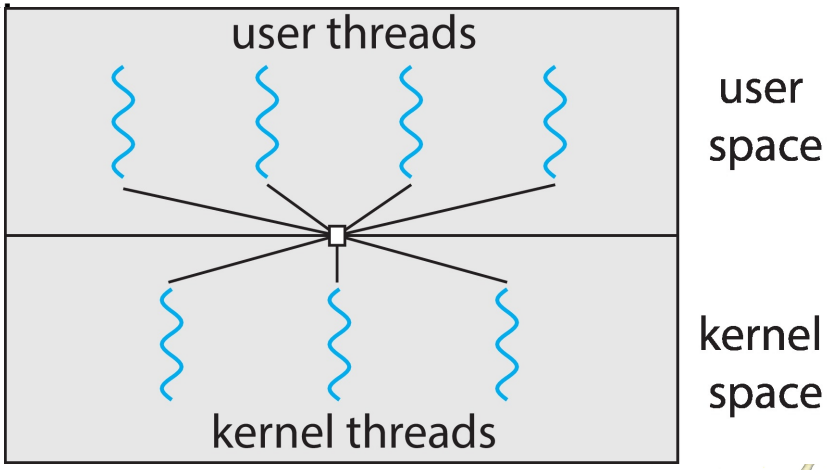
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- Number of threads per process sometimes restricted due to overhead of creating kernel threads. Hence places a limit on the number of threads created.
- Examples: Windows 95 to XP, Linux



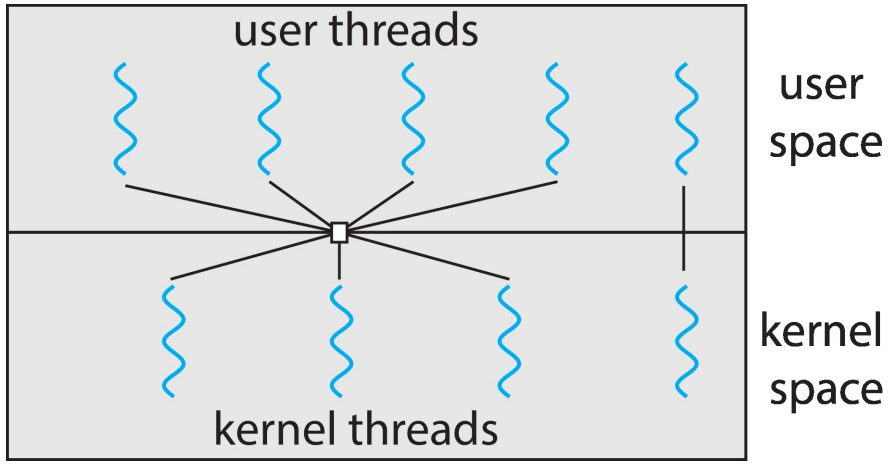
Many-to-Many Model

- Multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.
- This model is also called as two-tier model.
- Eg: IRIX, HP-UX, and Tru64 UNIX.



Two-level Model

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



Threading Issues

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

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
  - The new process can be a copy of the parent, with all the threads
  - The new process is a copy of the single thread only (that invoked the process)
- `exec()` usually works as normal – replace the running process including all threads i.e., if the thread invokes the exec( ) system call, the program specified in the parameter to exec( ) will be executed by the thread created.

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Example : Multiple threads required in loading a webpage is suddenly cancelled, if the browser window is closed.
- Two general approaches to cacel Threads that are no longer needed:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled thus giving an opportunity to the thread, to terminate itself in an orderly fashion.
  - ➢ In this method, the operating system will reclaim all the resources before cancellation.

Thread Cancellation (Cont.)

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

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

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

Signal Handling

- A signal is used to notify a process that a particular event has occurred.
- All signals follow same path-

1. A signal is generated by the occurrence of a particular event.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

- A signal can be invoked in 2 ways : synchronous or asynchronous.
  - **Synchronous** signal – signal delivered to the same program. Eg – illegal memory access, divide by zero error.
  - **Asynchronous** signal – signal is sent to another program. Eg – Ctrl C
- A signal can be handled by one of the two ways –
  - **Default** signal handler - signal is handled by OS.
  - **User-defined** signal handler - User overwrites the OS handler. It can override default

Signal Handling (Cont.)

- In a single-threaded program, the signal is sent to the same thread.
- But, in multithreaded environment, the signal is delivered in variety of ways, depending on the type of signal – 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 Pools

- In multithreading process, thread is created for every service. Eg – In web server, thread is created to service every client request.
- Creating new threads every time, when thread is needed and then deleting it when it is done can be inefficient, as –
- Time is consumed in creation of the thread.
- A limit has to be placed on the number of active threads in the system. Unlimited thread creation may exhaust system resources.
- Solution: Create a number of threads at process startup and place them in a pool where they await work
- Threads are allocated from the pool when a request comes, and returned to the pool when no longer needed(after the completion of request).
- When no threads are available in the pool, the process may have to wait until one becomes available.

Thread Pools

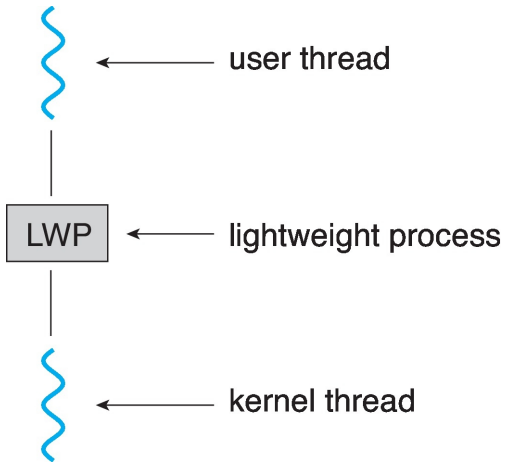
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.,Tasks could be scheduled to run periodically
- Windows API supports thread pools:

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)
- Data of a thread, which is not shared with other threads is called thread specific data.
- Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data.
- Example – if threads are used for transactions and each transaction has an ID. This unique ID is a specific data of the thread.
- 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
- 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



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 (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);
}
```

Pthreads Example (Cont.)

```
/* 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);
}
```

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);
```

End of Chapter 4