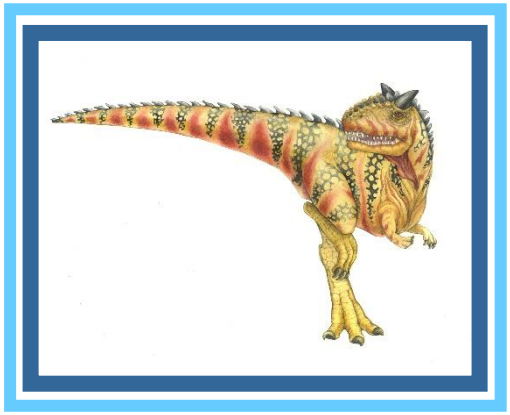
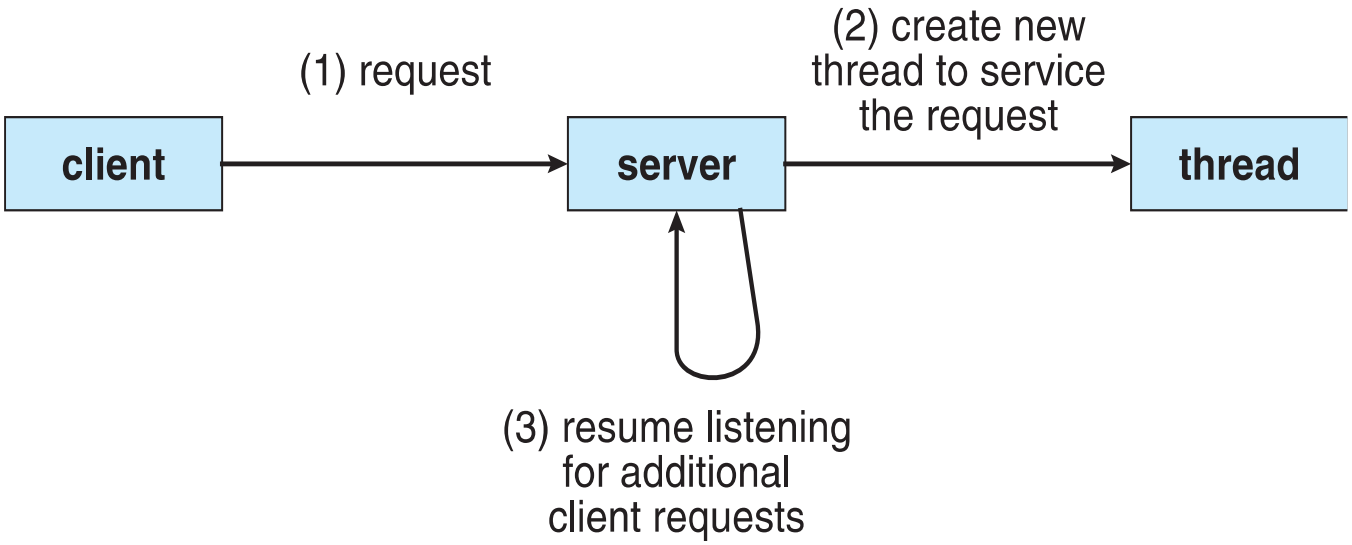
Chapter 4: Threads



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

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 –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

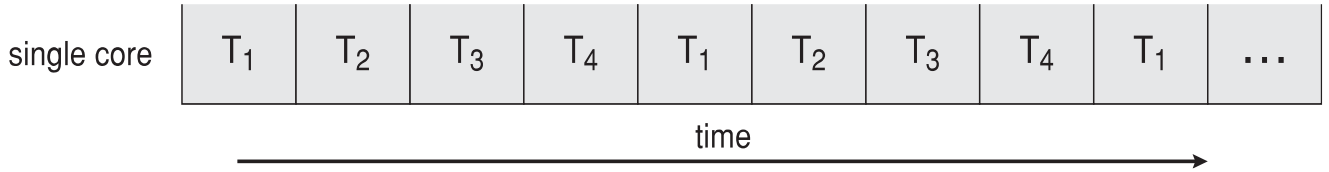
Each processor can execute one thread . Many processor can execute one process only.

Multicore Programming

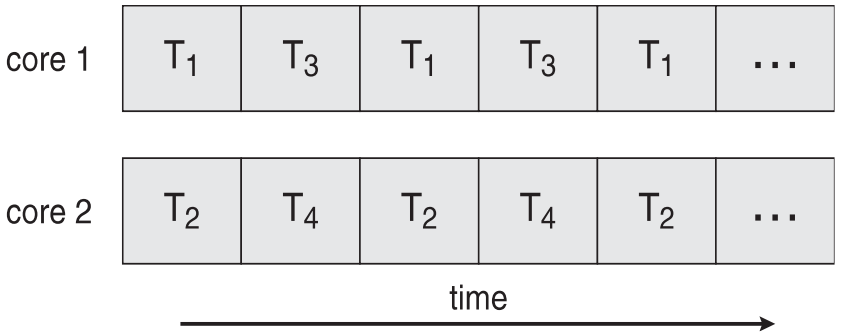
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously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

Concurrency vs. Parallelism

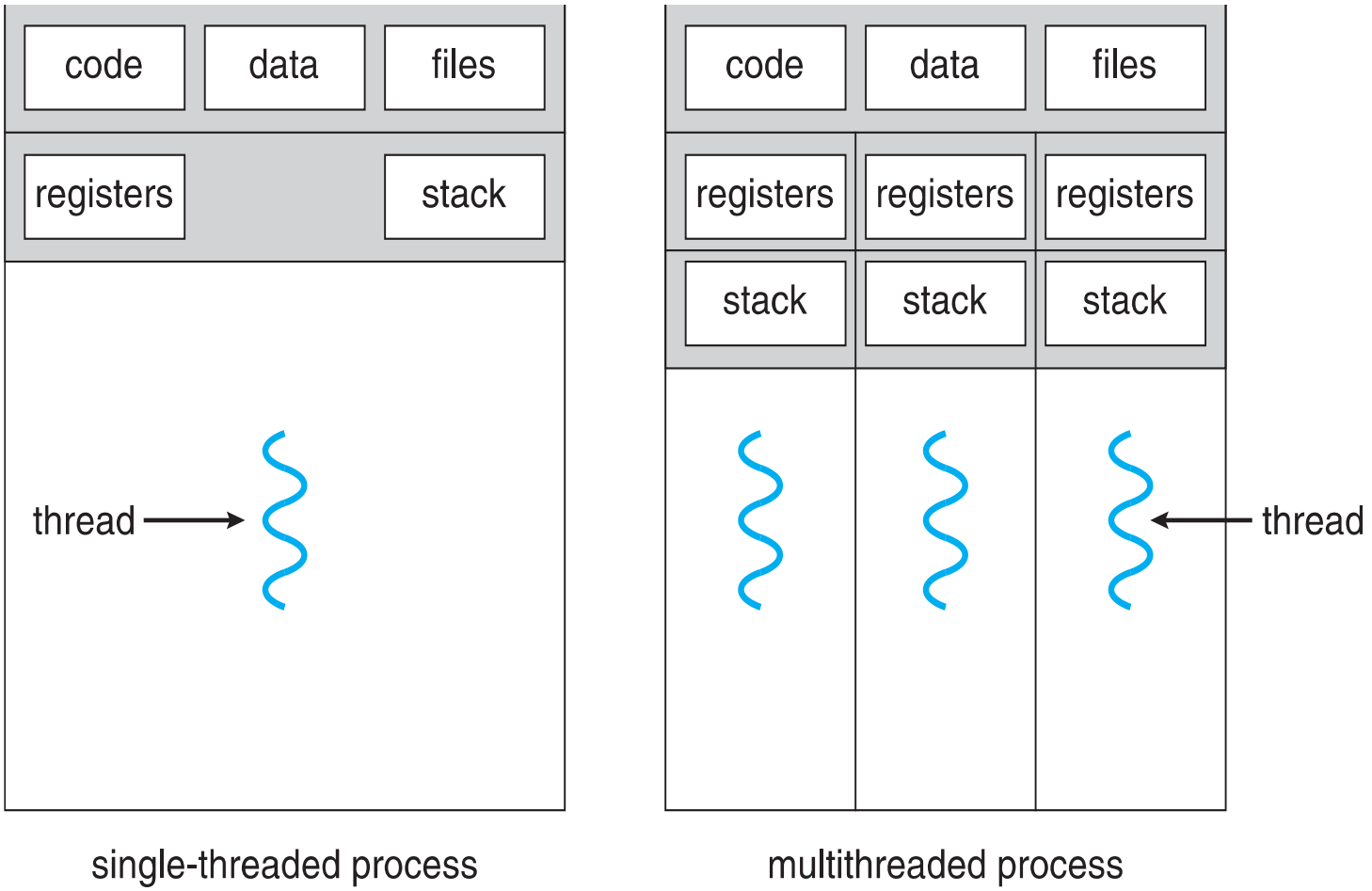
- **Concurrent execution on single-core system:**



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



Single and Multithreaded Processes



User Threads and Kernel Threads

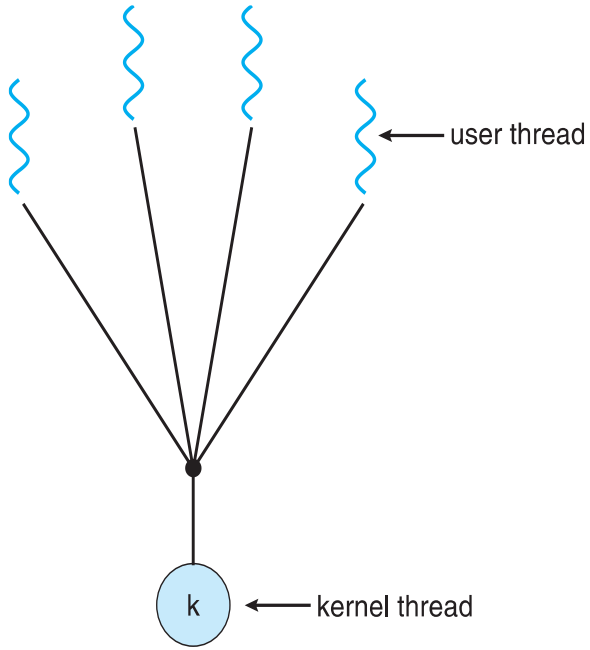
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

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

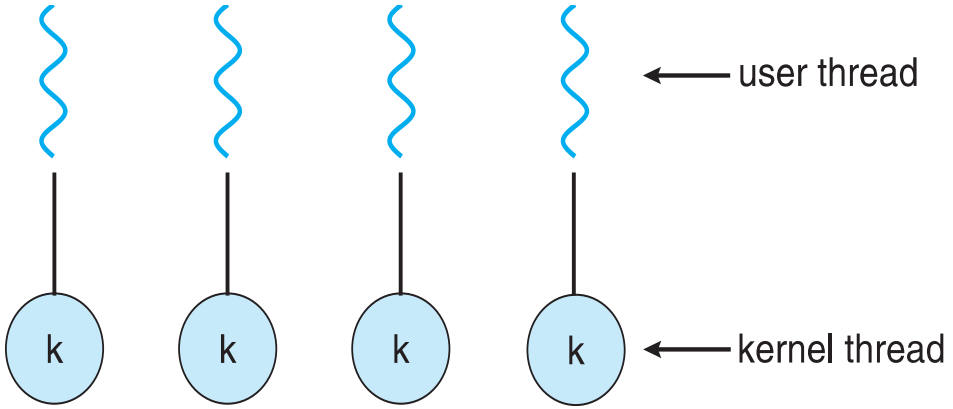
Many-to-One

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



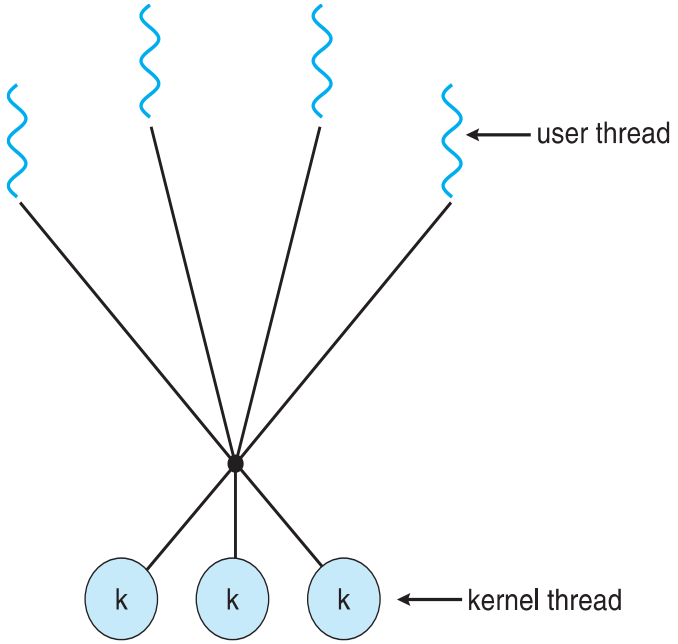
One-to-One

- 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 based on number of cpu
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



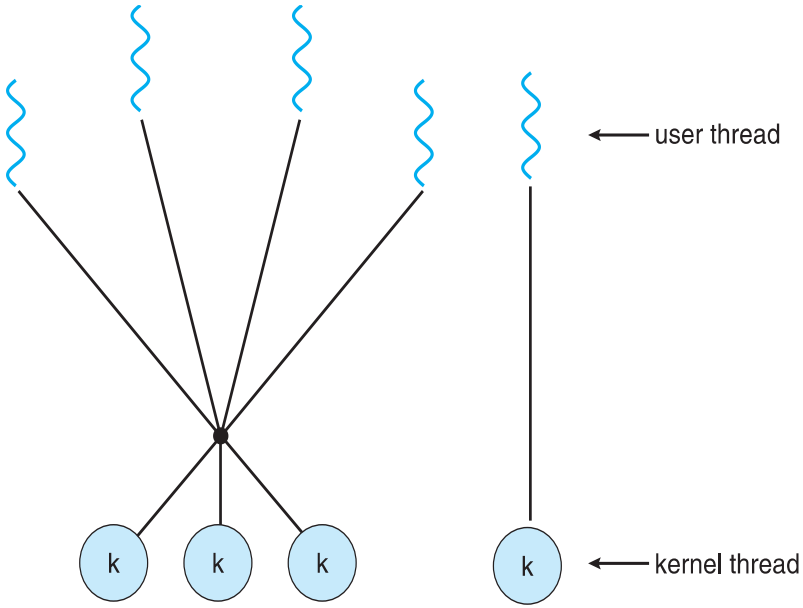
Many-to-Many Model

- 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

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



Multithreading :it create logical core from physical one

```
C:\Users\JAISON>wmic
wmic:root\cli>CPU Get NumberOfCores
NumberOfCores
2

wmic:root\cli>CPU Get NumberOfCores,NumberOfLogicalProcessors
NumberOfCores  NumberOfLogicalProcessors
2              4

wmic:root\cli>
```

Semantics of fork() and exec()

- Does **`fork()` duplicate only the calling thread** or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – **replace the running process** including all threads

Threading Issues (Part-1)

The fork() and exec() System Calls

The semantics of the fork() and exec() system calls change in a multithreaded program.

Issue

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?

Solution

Some UNIX systems have chosen to have two versions of fork (), one that duplicates all threads and another that duplicates only the thread that invoked the fork () system call.

But which version of fork () to use and when ?

Also, if a thread invokes the exec () system call, the program specified in the parameter to exec () will replace the entire process —including all threads.

But which version of fork () to use and when ?

Also, if a thread invokes the exec () system call, the program specified in the parameter to exec () will replace the entire process —including all threads.

Which of the two versions of fork () to use depends on the application.

If exec() is called immediately after forking

Then duplicating all threads is unnecessary, as the program specified in the parameters to exec () will replace the process.
In this instance, duplicating only the calling thread is appropriate.

If the separate process does not call exec () after forking

Then the separate process should duplicate all threads.

Thread Cancellation

- Terminating a thread before it has finished like to stop download or opening a web site. Or one thread got the results all others threads will be canceled .
- Thread to be canceled is called **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately by other thread.
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled it self. but not by others.

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
  - Asynchronous or deferred

  It may reclaim the resource or may not reclaim all the resource

  Others.

  Some threads my still needed by others .

  The solution is that the thread should checked its self before it get cancel .