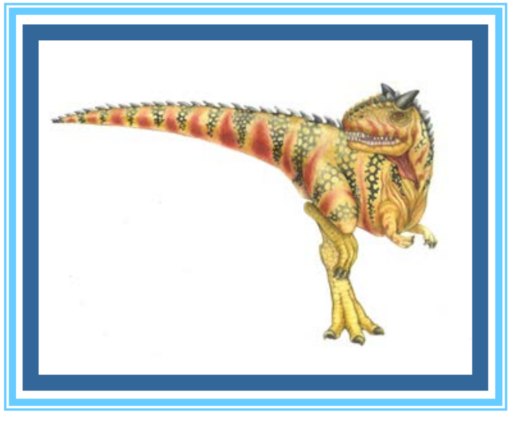
Chapter 4: Threads



Chapter 4: Threads

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

Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and and Java thread libraries

Processes

- Recall that a process includes many things
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)
- Creating a new process is costly because of all of the data structures that must be allocated and initialized
  - Recall struct proc in Linux
  - …which does not even include page tables, perhaps TLB flushing, etc.
- Communicating between processes is costly because most communication goes through the OS
  - Overhead of system calls and copying data

Parallel Programs

- To execute these programs we need to
  - Create several processes that execute in parallel
  - Cause each to map to the same address space to share data
    - They are all part of the same computation
  - Have the OS schedule these processes in parallel
- This situation is very inefficient
  - Space: PCB, page tables, etc.
  - Time: create data structures, fork and copy addr space, etc.
- Solutions: possible to have more efficient, yet cooperative "processes"?

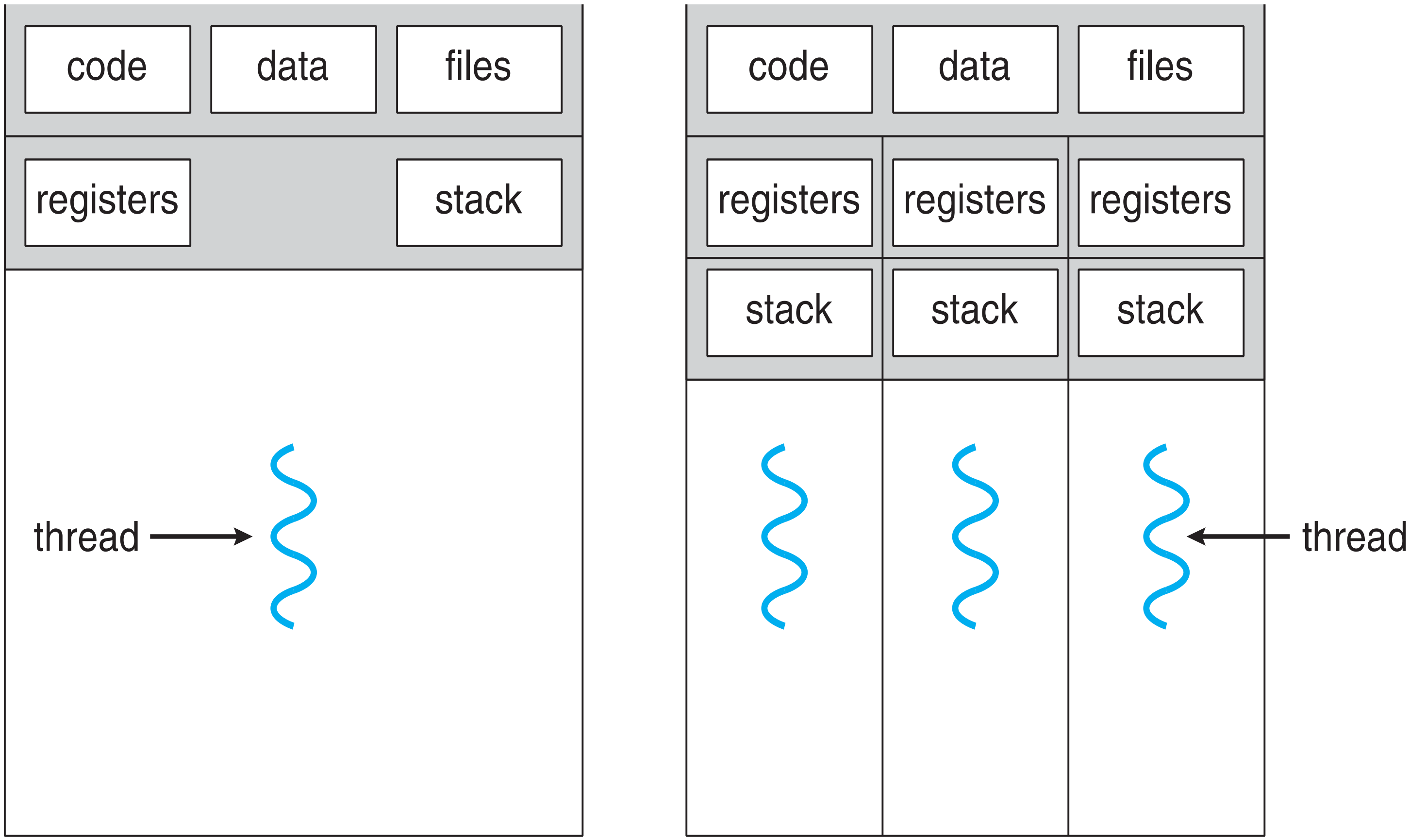
Rethinking Processes

- What is similar in these cooperating processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They all share the same resources (files, sockets, etc.)
- What don't they share?
  - Each has its own execution state: PC, SP, and registers
- Key idea: Why don't we separate the concept of a process from its execution state?
  - Process: address space, privileges, resources, etc.
  - Execution state: PC, SP, registers
- Exec state also called thread of control, or thread

Threads

- Modern OSes (Mac, Windows, modern Unix) separate the concepts of processes and threads
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the unit of scheduling
  - Processes are now the containers in which threads execute

Threads: lightweight processes



single-threaded process

multithreaded process

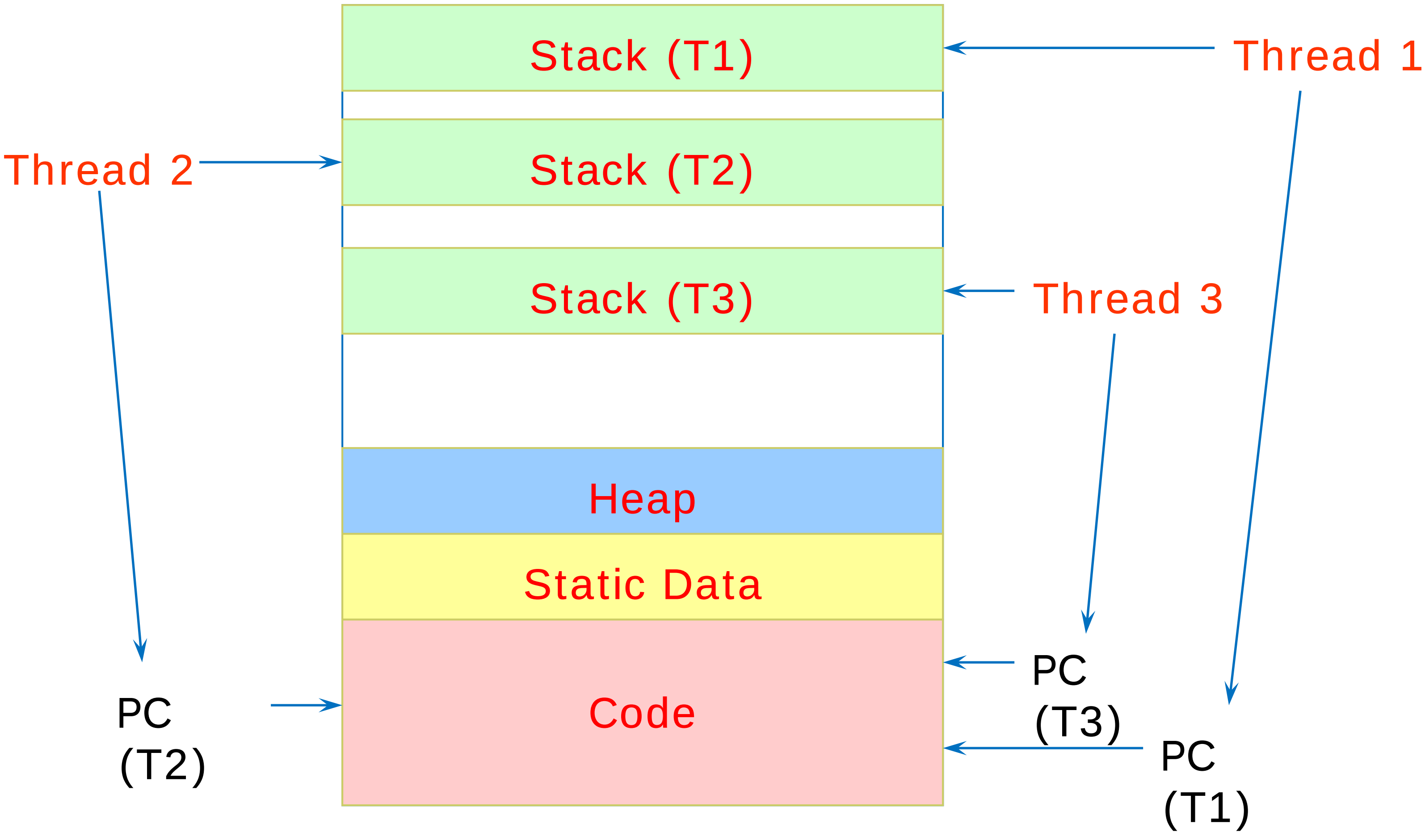
Benefits

- **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

The thread model

- Shared information
  - Processor info: parent process, time, etc
  - Memory: segments, page table, and stats, etc
  - I/O and file: communication ports, directories and file descriptors, etc
- Private state
  - Registers
  - Program counter
  - Execution stack
  - State (ready, running and blocked)
  - Why?
- Each thread execute separately

Threads in a Process



Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is overkill for such a simple task
- Recall our forking Web server:

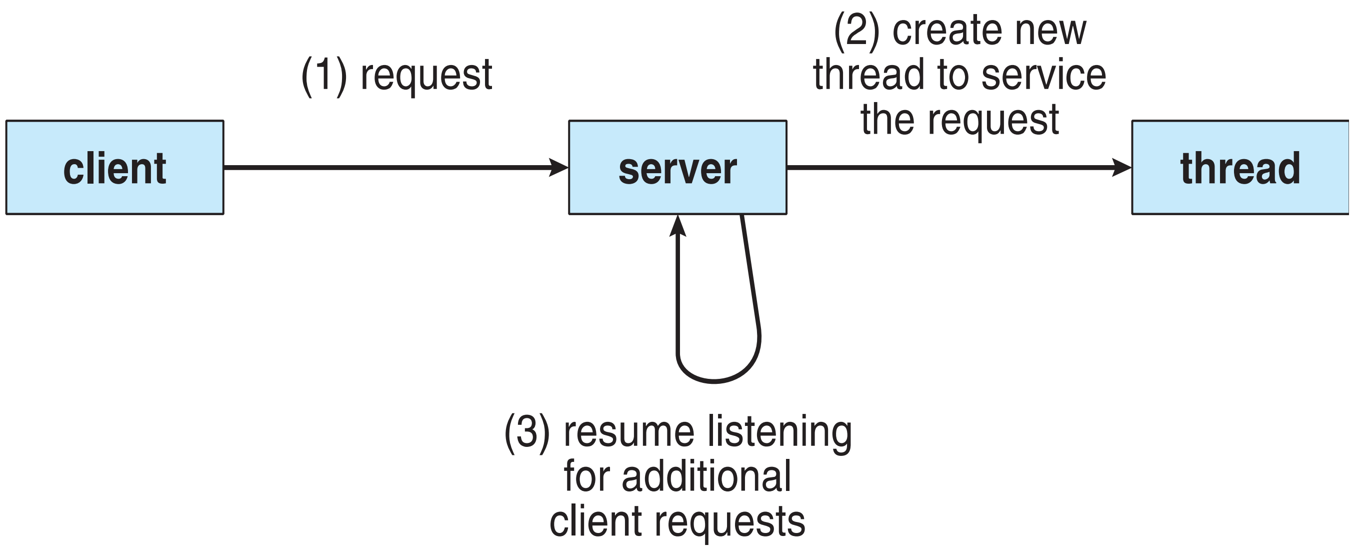
```
while (1) {
   int sock = accept();
   if ((child_pid = fork()) == 0) {
     Handle client request
     Exit
   } else {
     Close socket
   }
}
```

Threads: Concurrent Servers

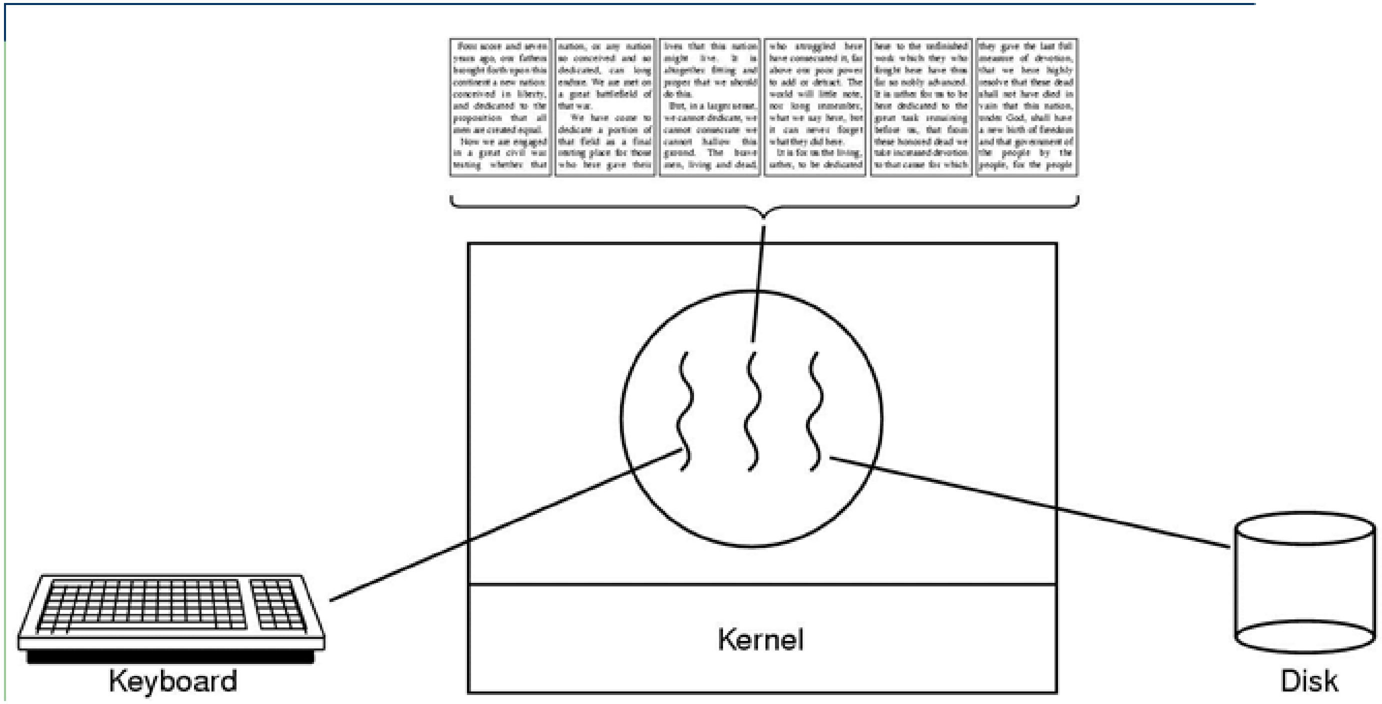
- Instead, we can create a new thread for each request

```
web_server() {
   while (1) {
    int sock = accept();
     thread_create(handle_request, sock);
   }
}
handle_request(int sock) {
    Process request
    close(sock);
}
```

Thread usage: web server



Thread usage: word processor



- A thread can wait for I/O, while the other threads can still running.
- What if it is single-threaded?

Drawbacks

- Make the programming more complicated
- Make the debugging harder
- Possible error when threads concurrently access the shared resources
- Poorly divided jobs can cause even worse system performance
- ……

1. A traditional (or heavyweight) process has a single thread of control.

A) True

B) False

2. A thread is composed of a thread ID, program counter, register set, and heap.

A) True

B) False

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

Kernel-Level Threads

- We have taken the execution aspect of a process and separated it out into threads
  - To make concurrency cheaper
- As such, the OS now manages threads *and* processes
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
- OS-managed threads are called kernel-level threads or lightweight processes
  - Windows: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads (pthreads): PTHREAD_SCOPE_SYSTEM

Kernel-level Thread Limitations

- Kernel-level threads make concurrency much cheaper than processes
  - Much less state to allocate and initialize
- However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead
  - Thread operations still require system calls
    - Ideally, want thread operations to be as fast as a procedure call
- For such fine-grained concurrency, need even “cheaper” threads

User-Level Threads

- To make threads cheap and fast, they need to be implemented at user level
  - Kernel-level threads are managed by the OS
  - User-level threads are managed entirely by the run-time system (user-level library)
- User-level threads are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call
    - No kernel involvement
  - User-level thread operations 100x faster than kernel threads
  - pthreads: PTHREAD_SCOPE_PROCESS

User-level Thread Limitations

- But, user-level threads are not a perfect solution
  - As with everything else, they are a tradeoff
- User-level threads are invisible to the OS
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
- Solving this requires communication between the kernel and the user-level thread manager

Kernel- vs. User-level Threads

- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slow to create, manipulate, synchronize
- User-level threads
  - Fast to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel- and user-level threads is important
  - For programming (correctness, performance)
  - For test-taking

Multiplexing User-Level Threads

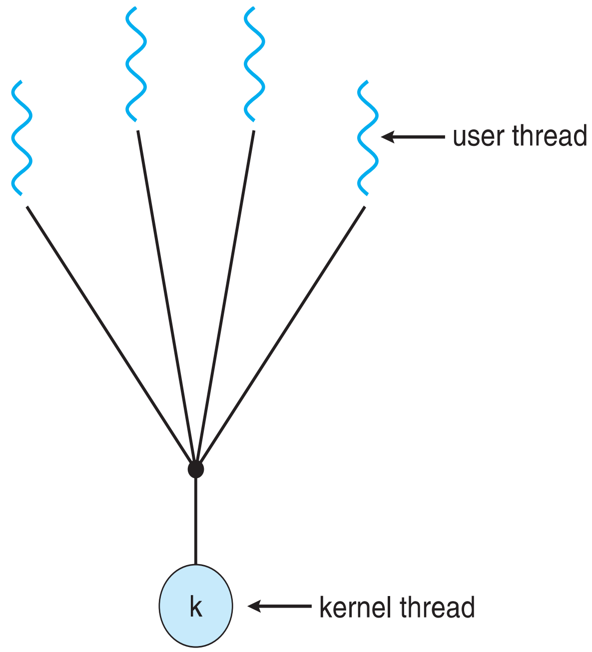
- Or use both kernel- and user-level threads
  - Can associate a user-level thread with a kernel-level thread
  - Or, multiplex user-level threads on top of kernel-level threads
  - A thread library map user threads to kernel threads
- Java Virtual Machine (JVM)
  - Java threads are user-level threads
  - On older Unix, only one “kernel thread” per process
    - Multiplex all Java threads on this one kernel thread
  - On Windows NT, modern Unix
    - Can multiplex Java threads on multiple kernel threads
    - Can have more Java threads than kernel threads

Multithreading Models

- Different mappings exist, representing different tradeoffs
  - Many-to-One: many user threads map to one kernel thread, i.e. kernel sees a single process
  - One-to-One: one user thread maps to one kernel thread
  - Many-to-Many: many user threads map to many kernel threads

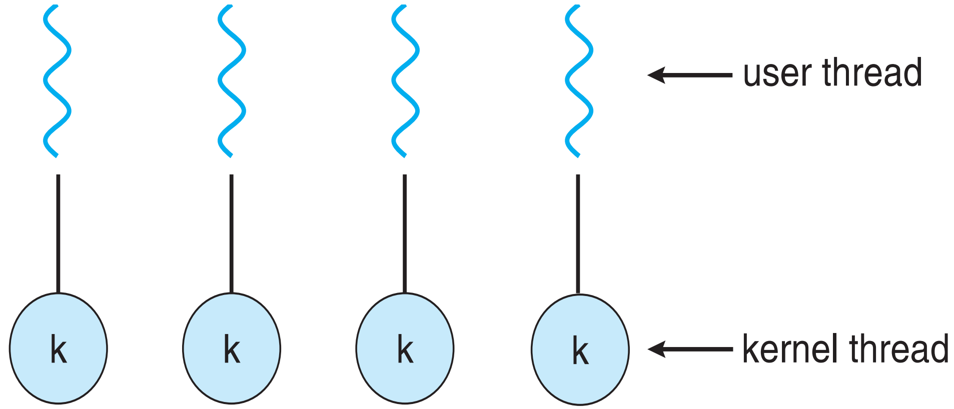
Many-to-One

- Many user-level threads mapped to single kernel thread
- Pros
  - Fast: no system calls required
  - Portable: few system dependencies
- Cons
  - No parallel execution of threads
    - All thread block when one waits for I/O
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



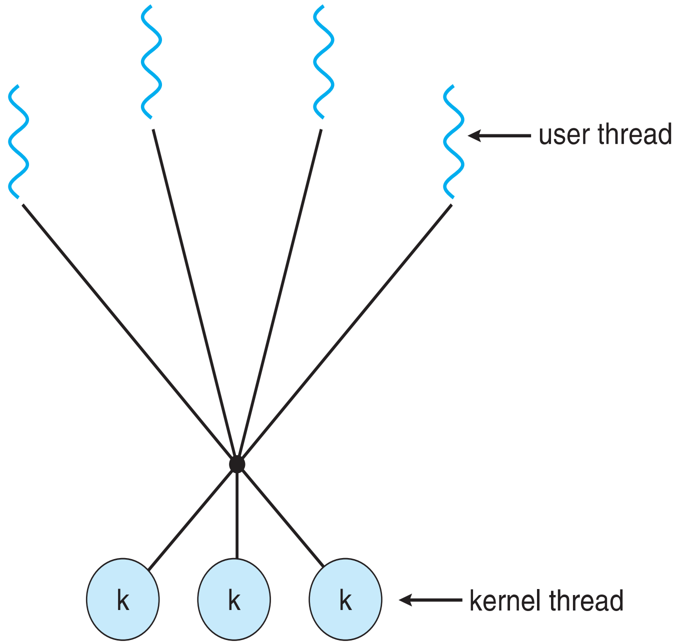
One-to-One

- Each user-level thread maps to kernel thread
- Pros: more concurrency
  - When one blocks, others can run
  - Better multicore or multiprocessor performance
- Cons: expensive
  - Thread operations involve kernel
  - Thread need kernel resources
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



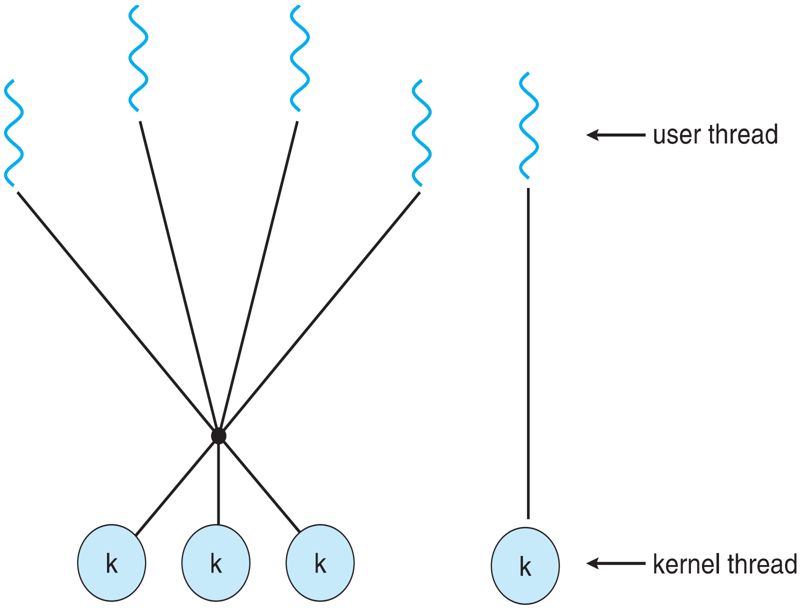
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Pros: flexible
  - OS creates kernel threads for physical concurrency
  - Applications creates user threads for application concurrency
- Cons: complex
  - Most programs use 1:1 mapping anyway
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Variation（M：M）：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



1. The _____ model multiplexes many user-level threads to a smaller or equal number of kernel threads.

A) many-to-many

B) two-level

C) one-to-one

D) many-to-one

2.The _____ model maps many user-level threads to one kernel thread.

A) many-to-many

B) two-level

C) one-to-one

D) many-to-one

3.The ______ model allows a user-level thread to be bound to one kernel thread.

A) many-to-many

B) two-level

C) one-to-one

D) many-to-one

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:

  - Implementing the Runnable interface

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

  - Extending Thread class

Java Multithreaded Program (method 1)

```
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>"); }
}
```

Creating Threads (method 2)

- extending the Thread class
  - must implement the *run()* method
  - thread ends when *run()* method finishes
  - call *.start()* to get the thread ready to run

Creating Threads Example

```
class Output extends Thread {
    private String toSay;
    public Output(String st) {
        toSay = st;
    }
    public void run() {
        try {
            for(;;) {
                System.out.println(toSay);
                sleep(1000);
            }
        } catch(InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

(continued)

```
class Program {
    public static void main(String [] args) {
        Output thr1 = new Output("Hello");
        Output thr2 = new Output("There");
        thr1.start();
        thr2.start();
    }
}
```

- main thread is just another thread (happens to start first)
- main thread can end before the others do
- any thread can spawn more threads

Controlling Java Threads

- _.*start()*: begins a thread running
- *wait()* and *notify()*: for synchronization
- _.*stop()*: kills a specific thread (deprecated)
- _.*suspend()* and *resume()*: deprecated
- _.*join()*: wait for specific thread to finish
- _.*setPriority*(): 0 to 10 (MIN_PRIORITY to MAX_PRIORITY); 5 is default (NORM_PRIORITY)

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

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

Operating System Examples

- Windows Threads
- Linux Threads

Process vs. threads

- Multithreading is only an option for “cooperative tasks”
  - Trust and sharing
- Process
  - Strong isolation but poor performance
- Thread
  - Good performance but share too much
- Example: web browsers
  - Safari: multithreading (no longer the case in the latest version)
    - one webpage can crash entire Safari
  - Google Chrome: each tab has its own process

Threads Summary

- *The operating system as a large multithreaded program*
  - Each process executes as a thread within the OS
- Multithreading is also very useful for applications
  - Efficient multithreading requires fast primitives
  - Processes are too heavyweight
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization…

End of Chapter 4