Lecture 4: Threads

Hossein Asadi (asadi@sharif.edu)

Rasool Jalili (jalili@sharif.edu)

Fall 2024

Silberschatz, Galvin and Gagne ©2013, Edited by H. Asadi, Fall 2024

Lecture 4: Threads

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

Objectives

- To Introduce Notion of a Thread—a Fundamental Unit of CPU Utilization that Forms Basis of Multithreaded Computer Systems
- To Discuss APIs for Pthreads, Windows, and Java Thread Libraries
- To Explore Strategies that Provide Implicit Threading
- To Examine Issues Related to Multithreaded Programming
- To Cover OS Support for Threads in Windows and Linux

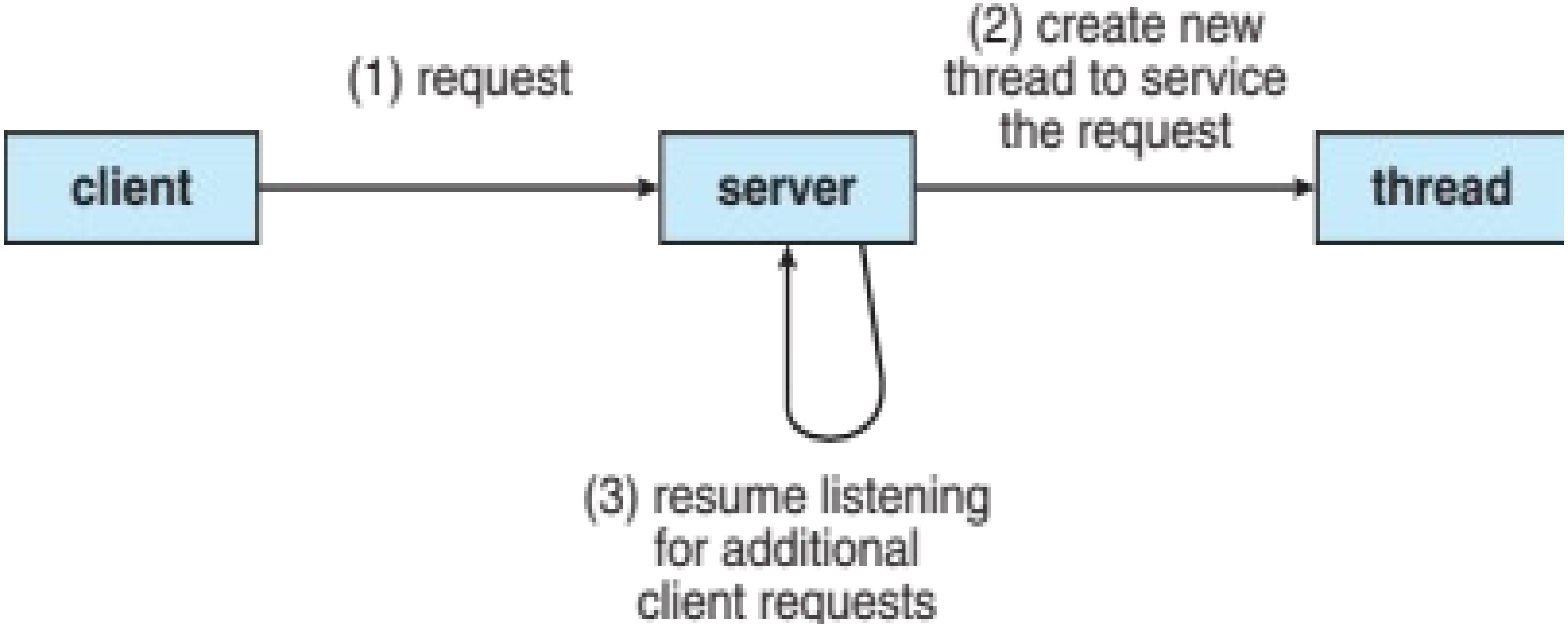
Motivation

- Many Parts of a Typical Application can be Run in Parallel (in Different Threads)
- Example 1: Web Browser
  - Thread A: display images
  - Thread B: retrieve data from network
- Example 2: Word Processor
  - Thread A: displaying graphics
  - Thread B: responding to keystrokes by user
  - Thread C: spell & grammar check in background

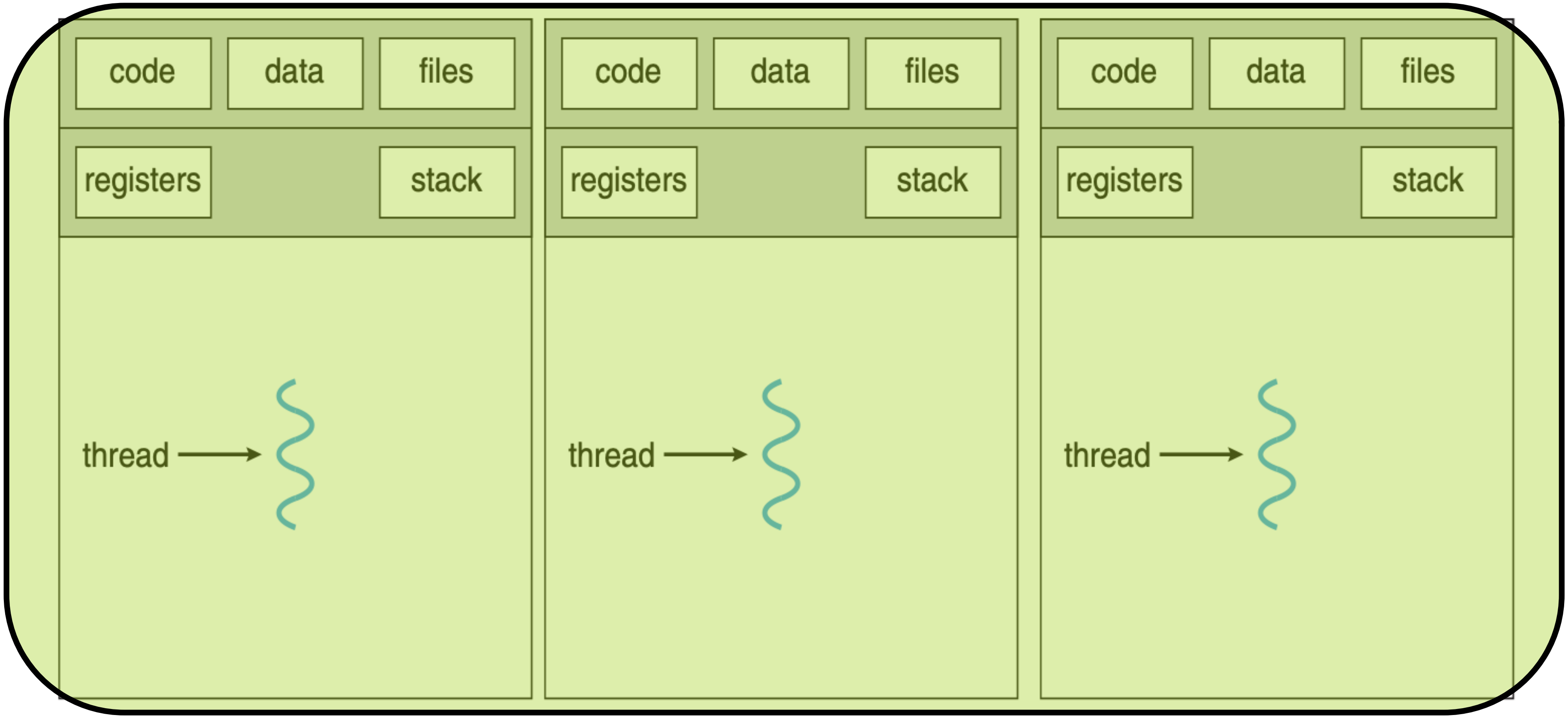
Motivation (cont.)

- Example 3: Matrix Multiplication (1000x1000)
  - Can have up to 1M parallel threads of execution
- Example 4: Remote Procedure Calls (RPCs)
- Example 5: when a process need an I/O
- Possible Solution: Creating Child Processes
  - Allows multiple flows of execution ☺
  - Time-consuming ☹
  - Resource consumptive ☹
- Fact:
  - Child Process will Perform Same Task as Parent Process → Why to Incur such Overheads?

Multithreaded Server Architecture



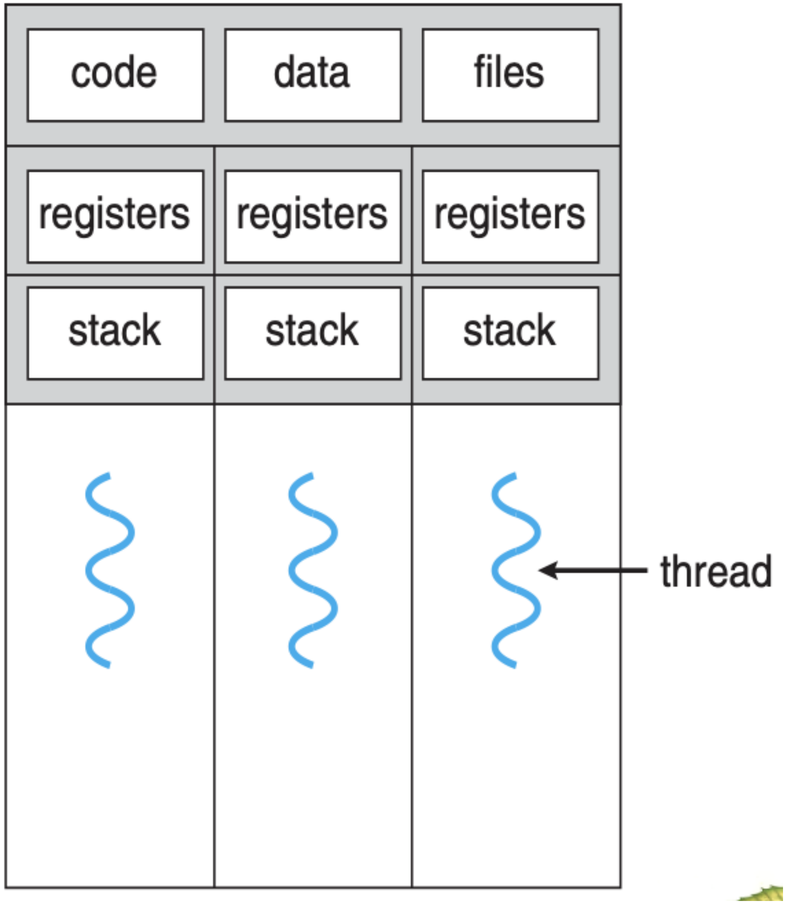
Single and Multithreaded Processes



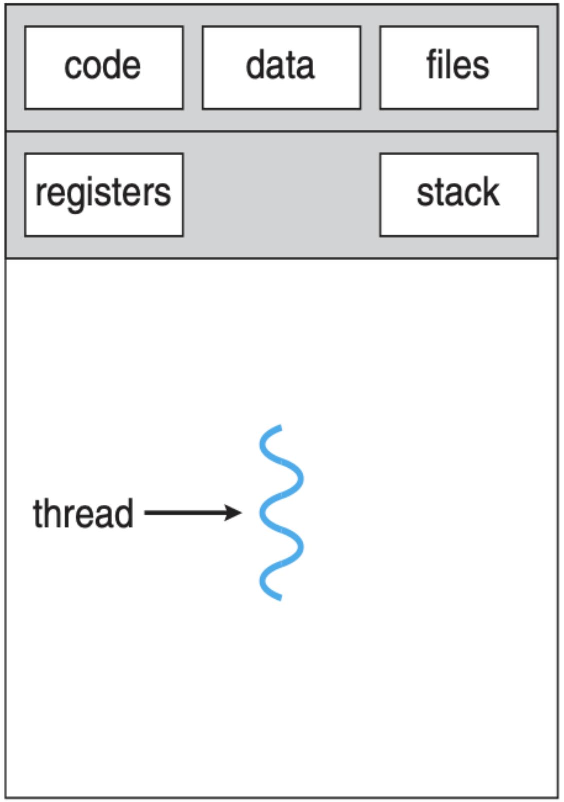
Single-Threaded Process

Multiple Processes

Single and Multithreaded Processes



Multithreaded Process



Single-Threaded Process

Multi-Threading in OS Kernels

- Most OSes are Multi-Threaded
  - Each thread performs a specific task
- Example
  - A thread to manage I/O type A
  - A thread to manage memory
  - A thread for interrupt handling

Benefits of Multi-Threading

- **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
  - Solaris: process creation 30X slower, context switching 5X slower
- **Scalability** – process can take advantage of multiprocessor architectures

Threads vs. Processes

- Processes typically Independent
  - Threads exist as subsets of a process
- Processes Carry Much more State Info
  - Threads of a process share process state as well as memory and other resources
- Processes have Separate Address Spaces
  - Threads share their address space
- Processes Interact Only through IPC
- Processes Much Slower Context Switch Time
- Processes Much Slower Creation Time

Multicore Programming

- **Multicore** or **Multiprocessor** Systems Putting Pressure on Programmers
- Programming Challenges Include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

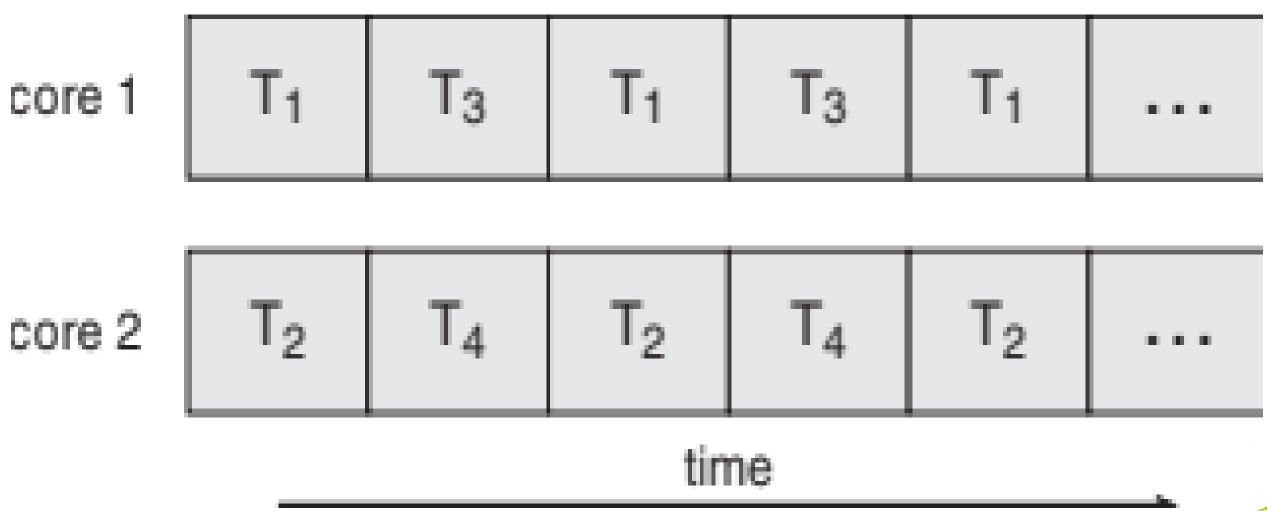
Multicore Programming (cont.)

- ***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
  - Possible to have concurrency without parallelism

Concurrency vs. Parallelism

- **Concurrent Exec. on a Single-Core System:**

- **Parallelism on a Multi-Core System:**



Amdahl's Law

- Imagine an Application has both Serial and Parallel Components
- *S* is Serial Portion
- *N* Processing Cores

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

- Sample Program: 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*

Multicore Programming (cont.)

- Before Advent of Multi-core Systems
  - CPU schedulers designed to provide illusion of parallelisms (not actual parallelism)
  - Processes were running concurrently (but not in parallel)
- As # of Threads Grows, so Does Architectural Support for Threading
  - CPUs have cores as well as ***HW threads***
    - HW support such as multiple sets of RFs and PCs
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

User Threads and Kernel Threads

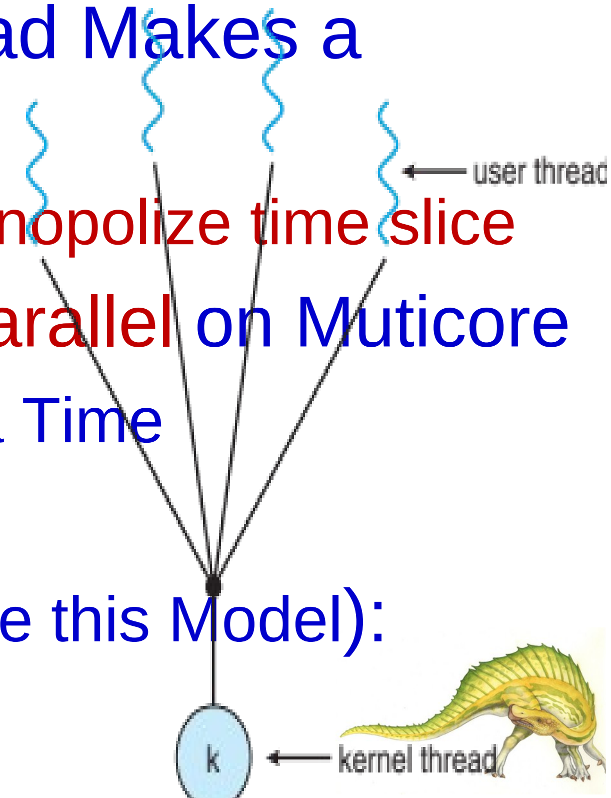
- **Kernel Threads**
  - Supported and managed directly by OS
- **User Threads**
  - Management done by user-level threads library
  - Supported above kernel and managed without kernel support
    - Kernel is unaware of them unless they are mapped to kernel threads
  - Context switching very fast (no interaction with kernel)
- **Note:** Ultimately, a Relationship Must Exist between User threads and Kernel threads

Multithreading Models

- Many-to-One (N:1)
  - All application-level threads maps to a single kernel-level scheduled entity
- One-to-One (1:1)
  - Threads created by user are in 1-1 correspondence with schedulable entities in kernel (Some say, no user-level thread)
- Many-to-Many (M:N)
  - Maps M number of application-level threads to N number of kernel entities

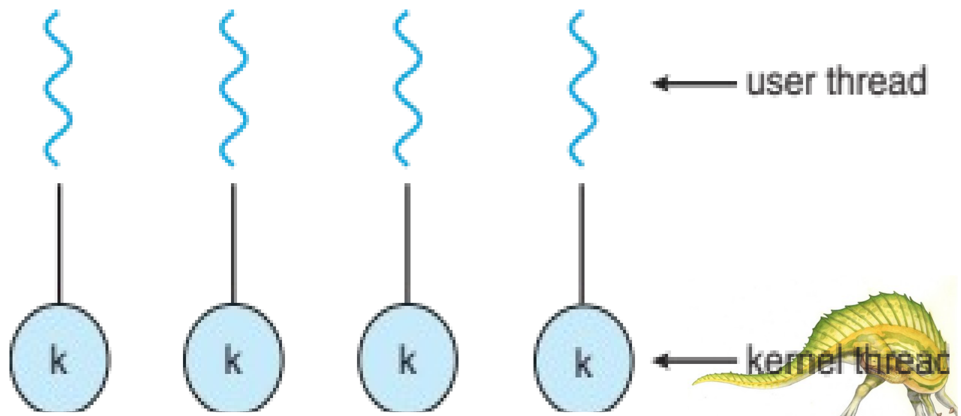
Many-to-One

- Many User-Level Threads Mapped to Single Kernel Thread
- Thread management Done by thread library in user space ➔ fast & efficient
- Entire Process Will Block if a thread Makes a Blocking System Call
  - Alternatively, a single thread can monopolize time slice
- Multiple Threads may not Run in Parallel on Muticore
  - Since only one may be in Kernel at a Time
  - Does not benefit from multi cores
- Examples (Few Systems Currently use this Model):
  - **Solaris Green Threads**
  - **GNU Portable Threads**



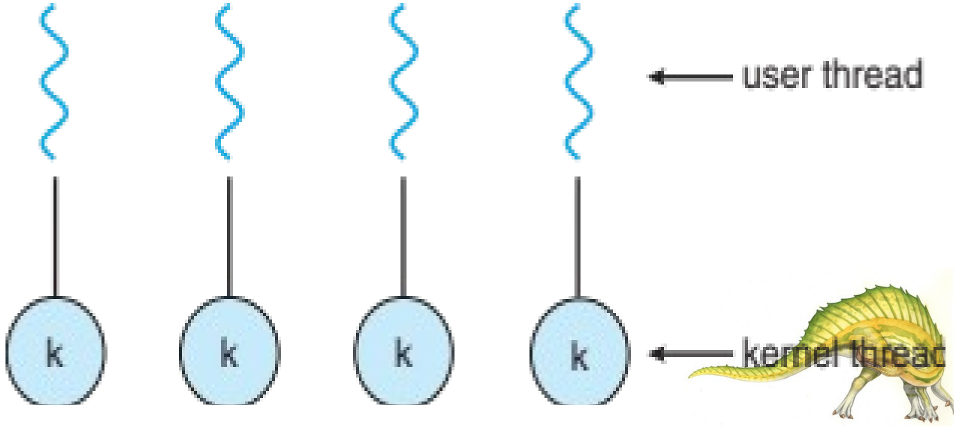
One-to-One

- Each User-Level Thread Maps to Kernel Thread
- Creating a User-Level Thread Creates a Kernel Thread
- More Concurrency than Many-to-One
  - Allows another thread to run when a thread makes a blocking system call
- Examples
  - Windows, Linux
  - Solaris 9 and later



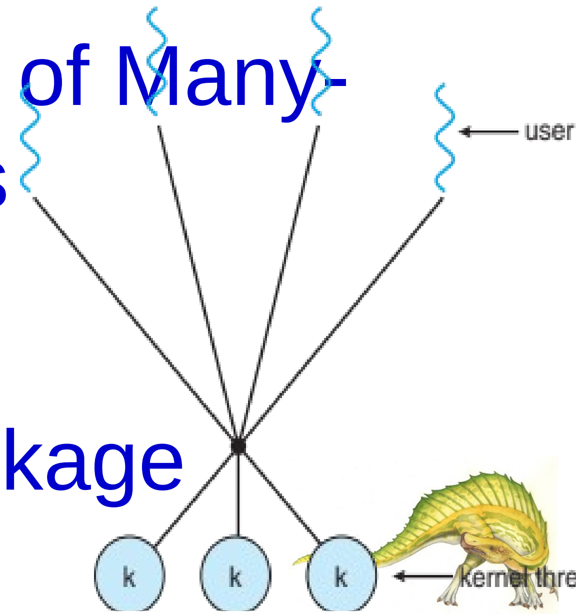
One-to-One (cont.)

- Main Drawback
  - Creating a user thread requires creating corresponding kernel thread
  - ➔ Overhead of creating kernel threads can burden performance of an application
  - Solution: most implementations of this model restrict number of threads supported by system



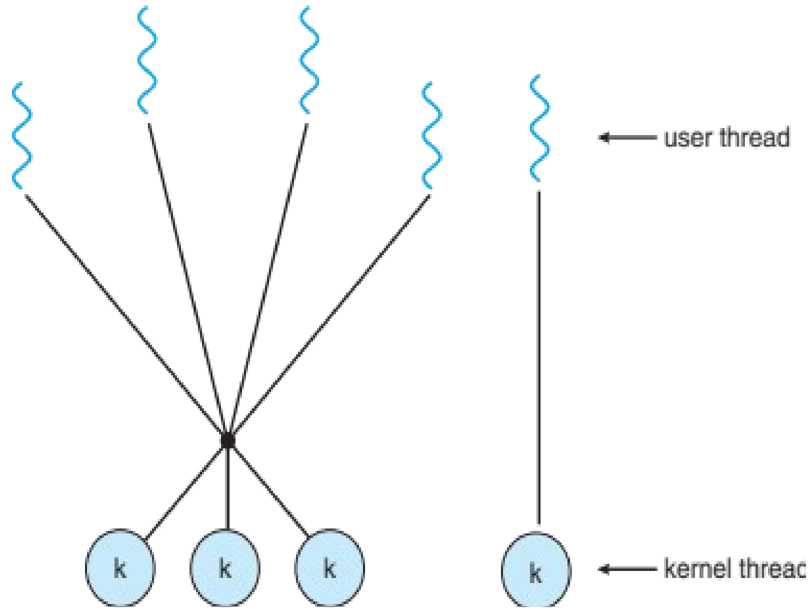
Many-to-Many Model

- Multiplexes Many User Level Threads to be Mapped to Smaller or Equal Number of Kernel Threads
- Allows OS to Create a Sufficient Number of Kernel Threads
- Tries to Address Shortcomings of Many-to-one and One-to-One Models
- Solaris prior to version 9
- Windows with *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



Processes, User Threads, Kernel Threads

- Process
  - Isolated with its own virtual address space
  - Contains process data like file handles
  - Lots of overhead
  - Every process has at least one kernel thread
- Kernel Threads
  - Shared virtual address space
  - Contains running state data
  - Less overhead
  - From OS's point of view, this is what is scheduled to run on a CPU

© Introduction to OS, University of Washington, by Zahorjan and Kim, Spring 2011.

Processes, User Threads, Kernel Threads

- User Threads
  - Shared virtual address space, contains running state data
  - Kernel unaware
  - Even less overhead

© Introduction to OS, University of Washington, by Zahorjan and Kim, Spring 2011.

Trade-Offs

- Processes
  - Secure and isolated ☺
  - Kernel aware ☺
  - Creating a new process (address space!) brings lots of overhead ☹

© Introduction to OS, University of Washington, by Zahorjan and Kim, Spring 2011.

Trade-Offs (cont.)

- Kernel Threads
  - No need to create a new address space ☺
  - No need to change address space in context switch ☺
  - Kernel aware ☺
  - Still need to enter kernel to context switch ☹
- User Threads
  - No new address space, no need to change address space ☺
  - No need to enter kernel to switch ☺
  - Kernel is unaware; No multiprocessing; I/O blocks all user threads; ☹

© Introduction to OS, University of Washington, by Zahorjan and Kim, Spring 2011.

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 OS

Thread Libraries (cont.)

- Library Entirely in User Space
  - With no kernel support
  - All code/data structures in user space
  - Invoking a function in API results in a local function call in user space not a system call
- Kernel-Level Library Supported by OS
  - Code and data structures for library exist in kernel space
  - Invoking a function in API ➔ system call

Thread Libraries (cont.)

- Three Main Thread Libraries in Use
- POSIX Pthreads Library
  - Thread extension of POSIX standard
  - May be provided as either a user- or kernel-level
- Win32 Thread Library
  - Kernel-level library
- Java Thread API
  - Allows threads to be created & managed directly in Java programs
  - Implemented using a thread library of host system since JVM runs on top of host system

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 Thread Library, Implementation is up to Development of Library
- Common in UNIX OS (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 Managed by JVM
- Typically Implemented using Threads Model provided by Underlying OS
- Each Java Program consists of at Least One Single Thread

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
- Two Methods explored (Reading Assignment)
  - OpenMP
  - Grand Central Dispatch

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal Handling
  - Synchronous and asynchronous
- Thread Cancellation of Target Thread
  - Asynchronous or deferred
- Scheduler Activations

Semantics of fork() and exec()

- Does **`fork()`** Duplicate only Calling Thread or all Threads?
  - Some UNIXes have two versions of fork
- What about **`exec?`**
  - Usually works as normal
  - Replaces running process including all threads

Signal Handling

- **Signals** Used in UNIX to Notify a Process that a Particular Event has Occurred
  - SW generated interrupts sent to a process when an event happens
- **Signal Handler** Used to Deal with Signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     - default
     - user-defined

Signal Handling (cont.)

- Every Signal has **Default Handler** that Kernel Runs when Handling Signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process
- Default Actions
  - Signal discarded after being received
  - Process terminated after signal received
  - A core file written, then process terminated
  - Stop process after signal received

Signal Handling (cont.)

- Example

NAME	Default Action	Description
SIGHUP	terminate process	terminal line hangup
SIGINT	terminate process	interrupt program
SIGQUIT	create core image	quit program
SIGILL	create core image	illegal instruction
SIGTRAP	create core image	trace trap
SIGABRT	create core image	abort(3) call (formerly SIGIOT)
SIGEMT	create core image	emulate instruction executed
SIGFPE	create core image	floating-point exception
SIGKILL	terminate process	kill program
SIGBUS	create core image	bus error
SIGSEGV	create core image	segmentation violation
SIGSYS	create core image	non-existent system call invoked
SIGPIPE	terminate process	write on a pipe with no reader
SIGALRM	terminate process	real-time timer expired
SIGTERM	terminate process	software termination signal
SIGURG	discard signal	urgent condition present on socket
SIGSTOP	stop process	stop (cannot be caught or ignored)
SIGTSTP	stop process	stop signal generated from keyboard
SIGCONT	discard signal	continue after stop

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
  printf("Caught signal!\n");
  exit(1);
}
int main() {
  struct sigaction sa;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sa.sa_handler = signal_callback_handler;
  sigaction(SIGINT, &sa, NULL);
  while (1) {}
}
```

Signal Handling (cont.)

- Where should a Signal be Delivered for Multi-Threaded?
  - Deliver signal to thread to which signal applies
  - Deliver signal to every thread in process
  - Deliver signal to certain threads in process
  - Assign a specific thread to receive all signals for process

Thread Pools

- Create a Number of Threads in a Pool where they Await Work

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

- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows number of threads in application(s) to be bound to size of pool
- Windows API supports thread pools

Thread Cancellation

- Terminating a Thread before it has finished
- Thread to be Canceled is **Target Thread**
- Two General Approaches
  - **Asynchronous cancellation** terminates target thread immediately
  - **Deferred can**
    periodically ch
- Pthread Code

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

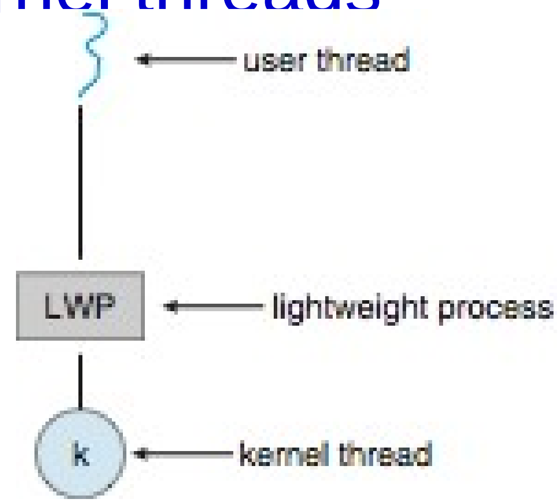
- Invoking Thread Cancellation Requests Cancellation, but Actual Cancellation Depends on Thread State

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, thread cancellation handled through signals

Scheduler Activations

- M:M/Two-Level Models Require communication to maintain appropriate number of kernel threads allocated to application



- **LightWeight Process (LWP)**
  - Typically use an intermediate data structure between user and kernel threads
  - 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?

Operating System Examples

- Reading Assignments
  - Windows threads
  - Linux threads
  - Upcalls

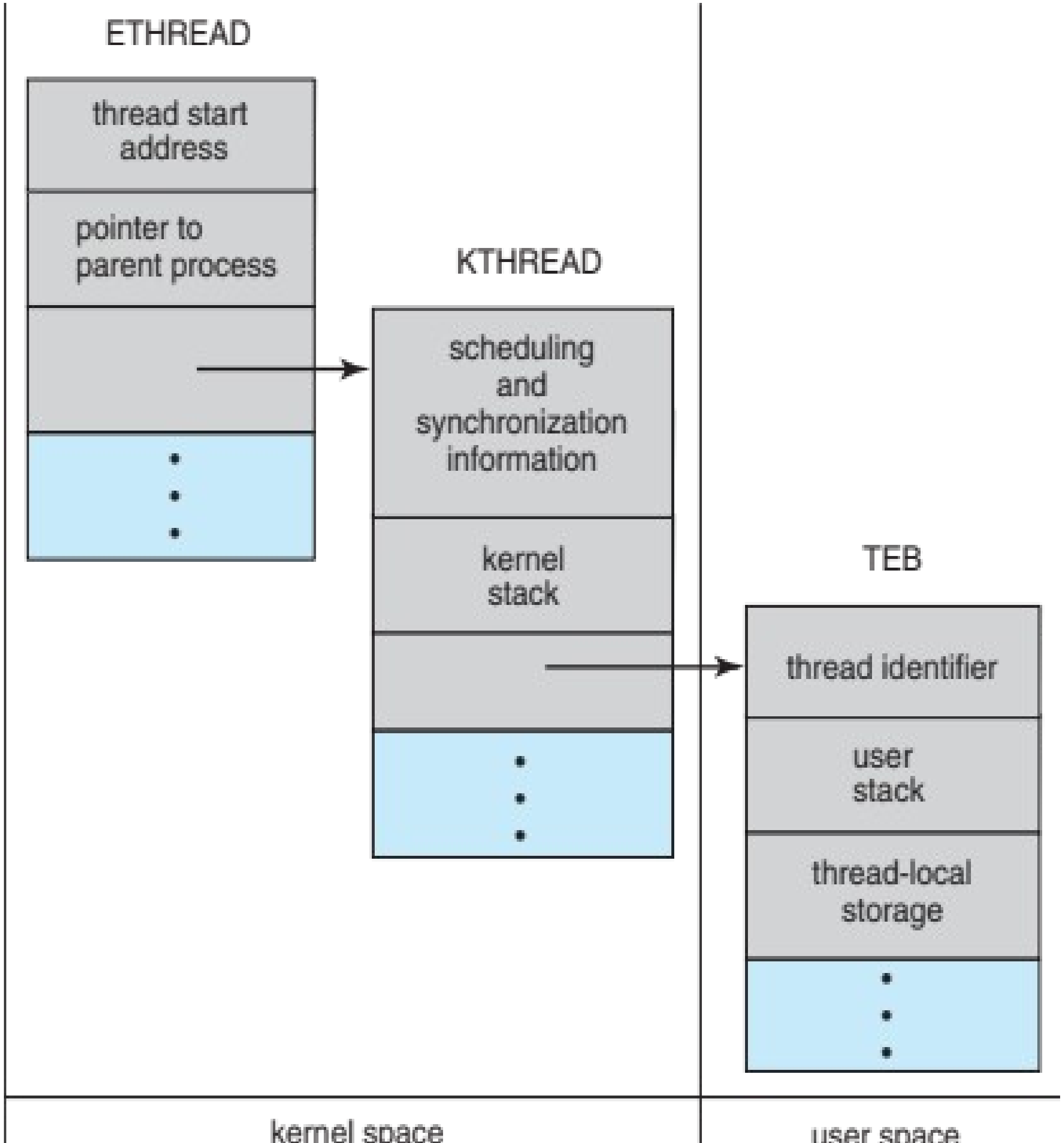
Windows Threads

- Windows implements Windows API
  - Primary API for Win98, WinNT, Win2000, WinXP, Win7
- Implements 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)
- **Context of Thread**: register set, stacks, and private storage area

Windows Threads (Cont.)

- 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

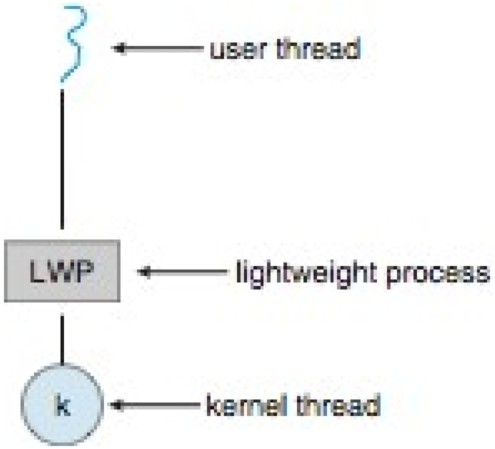


Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation using **`clone()`** system call
- **`clone()`** allows a child task to share address space of parent task (process)
  - Flags control behavior
- **`struct task_struct`** points to process data structures (shared or unique)

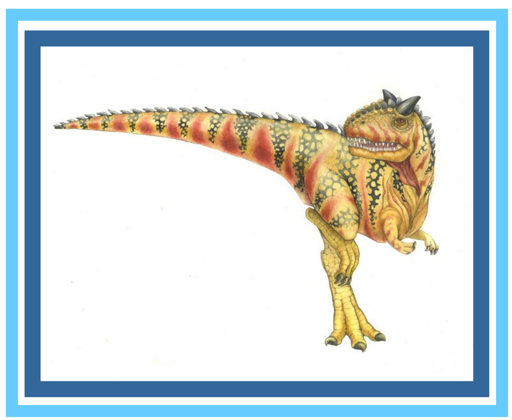
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Scheduler Activations (cont.)



- Scheduler Activations Provides **upcalls**
  - A communication mechanism from kernel to **upcall handler** in thread library
  - This communication allows an application to maintain correct number kernel threads

End of Lecture 4



Silberschatz, Galvin and Gagne ©2013, Edited by H. Asadi, Fall 2024