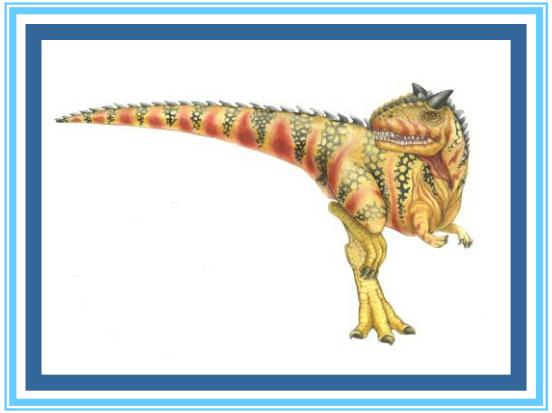
Chapter 4: Threads & Concurrency



Chapter 4: Threads

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

Describe how the Windows and Linux operating systems represent threads

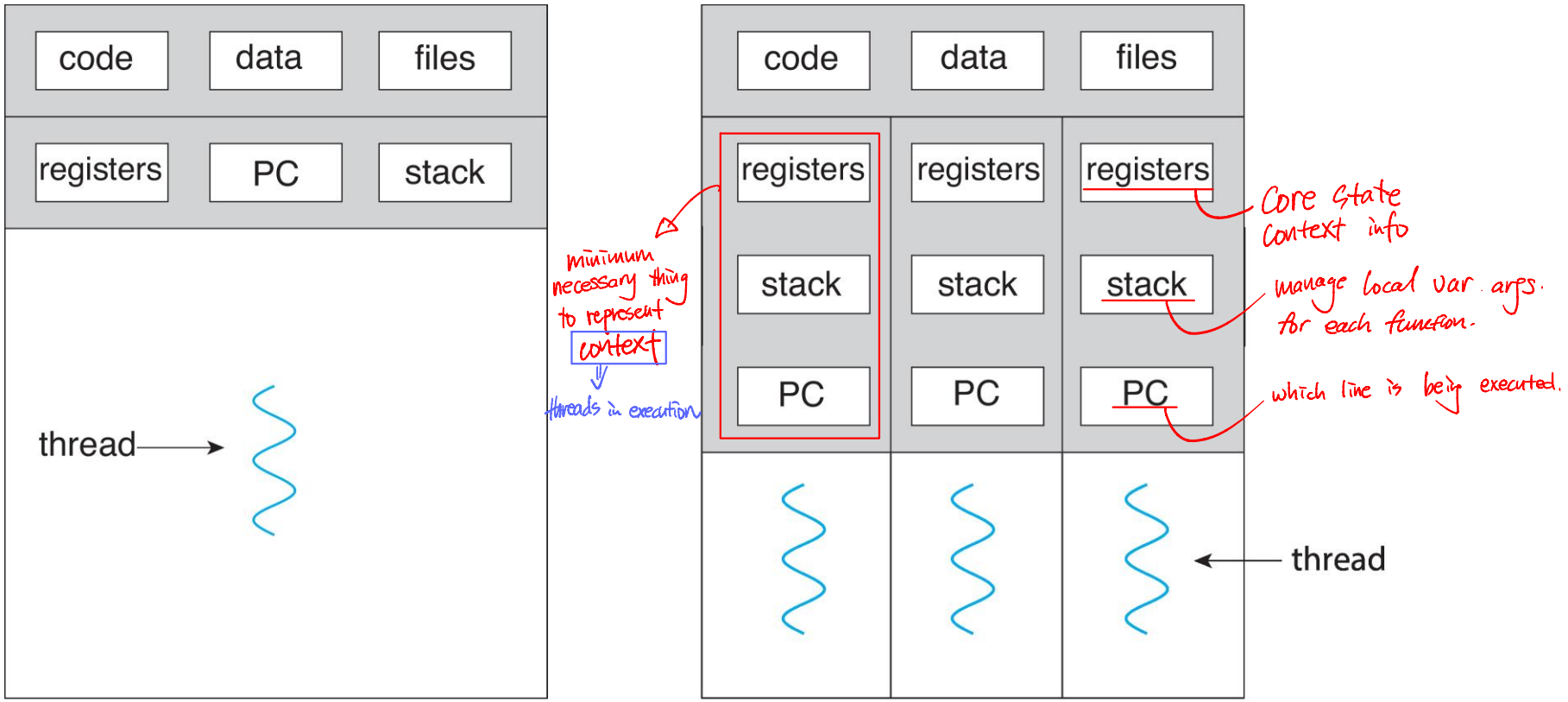
Design multithreaded applications using the Pthreads, Java, and Windows threading APIs

Motivation

- Most modern applications are multithreaded
  - Threads run within application — one process, multiple threads
  - Multiple tasks with the application can be implemented by separate threads — independently.
    - Update display
    - Fetch data
    - Spell checking
    - Answer a network request

    } assigned to threads respectively
  - Process creation is heavy-weight while thread creation is light-weight
    - Can simplify code, increase efficiency
- Kernels are generally multithreaded — for reliability
  - managing devices, memory managment, interrupt handling...

Single and Multithreaded Processes



single-threaded process

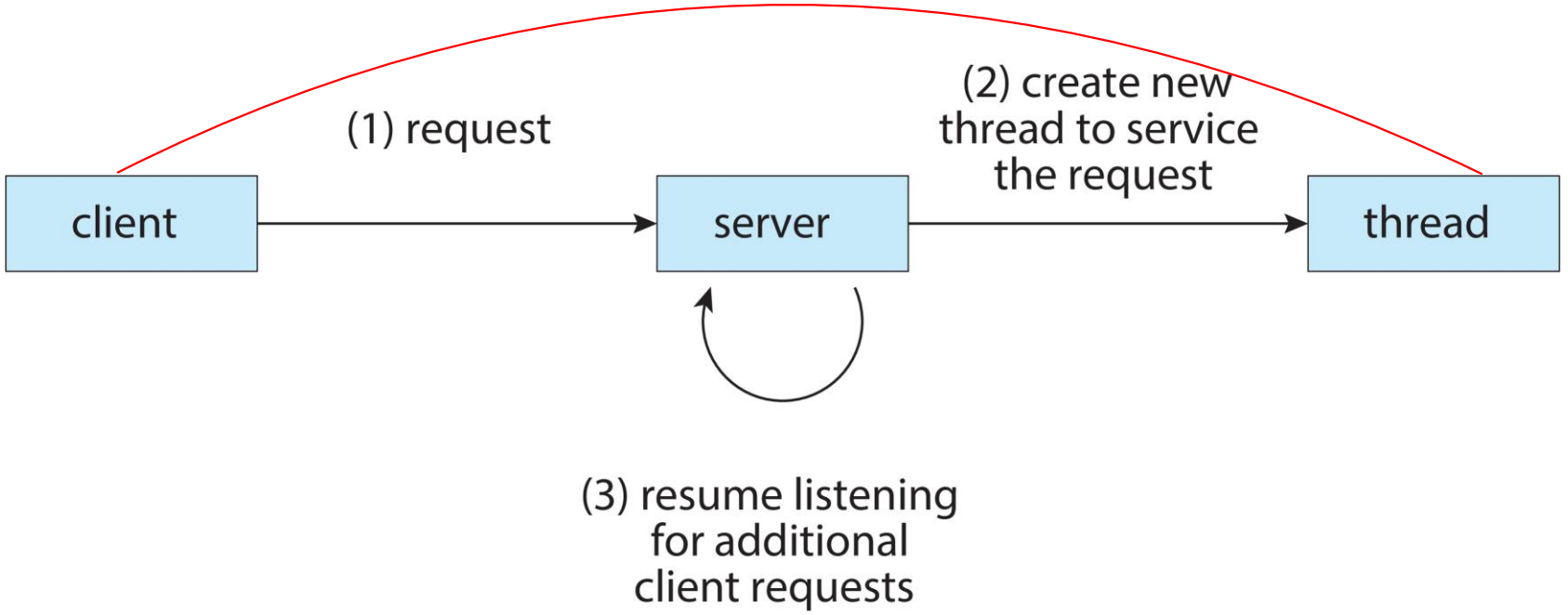
run functions one by one.

multithreaded process

run several functions concurrently

Multithreaded Server Architecture

connection 1:1 mapping
client : thread
concurrent open . termination



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>

#define PORT 8080
#define MAX_CLIENTS 10

void *handle_client(void *);

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // Create server socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

// Bind server socket to port
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
    }
```

```
    // Listen for incoming connections
    if (listen(server_fd, MAX_CLIENTS) < 0) {
        perror("Listen failed");
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d\n", PORT);

    // Accept incoming connections and spawn threads to handle them
    pthread_t threads[MAX_CLIENTS];
    int i = 0;
    while (1) {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t*)&addrlen)) < 0) {
            perror("Accept failed");
            exit(EXIT_FAILURE);
        }

        printf("Client connected\n");

        if (pthread_create(&threads[i], NULL, handle_client, (void *)&new_socket)) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }

        i++;
    }

    // Close server socket
    close(server_fd);

    return 0;
}
```

```
void *handle_client(void *arg) {
    int client_socket = *(int *)arg;
    char buffer[1024] = {0};
    int valread;

    Read client request
    valread = read(client_socket, buffer, 1024);
    printf("Received message: %s\n", buffer);

    Send response to client
    char *response = "Hello from server";
    send(client_socket, response, strlen(response), 0);
    printf("Response sent\n");

    Close client socket
    close(client_socket);

    return NULL;
}
```

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

∵ in the same process. ⇒ easy · flexible

Economy – cheaper than process creation, thread switching lower overhead than context switching

↳ create only context-related resource.
others are already allocated.

Scalability – process can take advantage of multicore architectures

Multicore Programming

Tasks are assigned to each core & executed independently

Multicore or **multiprocessor** systems putting pressure on programmers, challenges include:

- **Dividing activities** independency of tasks
- **Balance** between tasks (running time)
- **Data splitting** no split ⇒ stuck & delay
- **Data dependency** task depends on data from another ⇒ execution of task is sync to accommodate data dependency
- **Testing and debugging** easy ⇒ simple code !

Parallelism implies a system can perform more than one task simultaneously

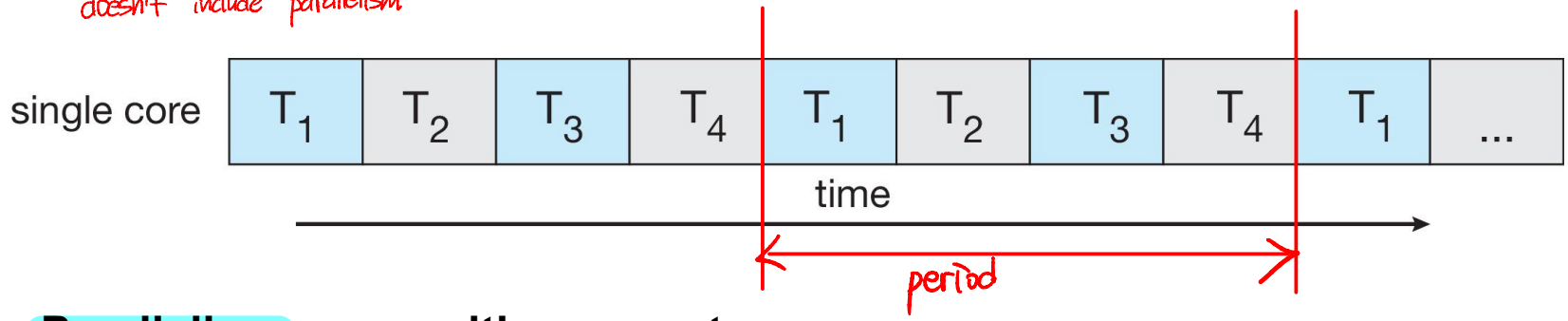
Concurrency supports more than one task making progress

Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

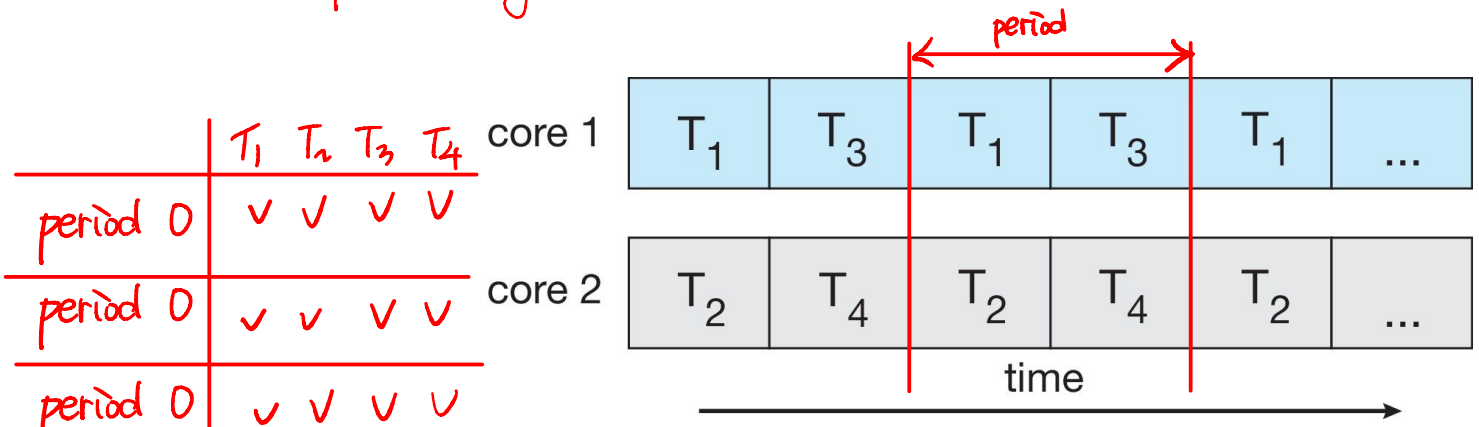
Concurrent execution on single-core system:

doesn't include parallelism



Parallelism on a multi-core system:

implies concurrency



	T_1 T_2 T_3 T_4
period 0	✓ ✓ ✓ ✓
period 0	✓ ✓ ✓ ✓
period 0	✓ ✓ ✓ ✓
⋮	⋮

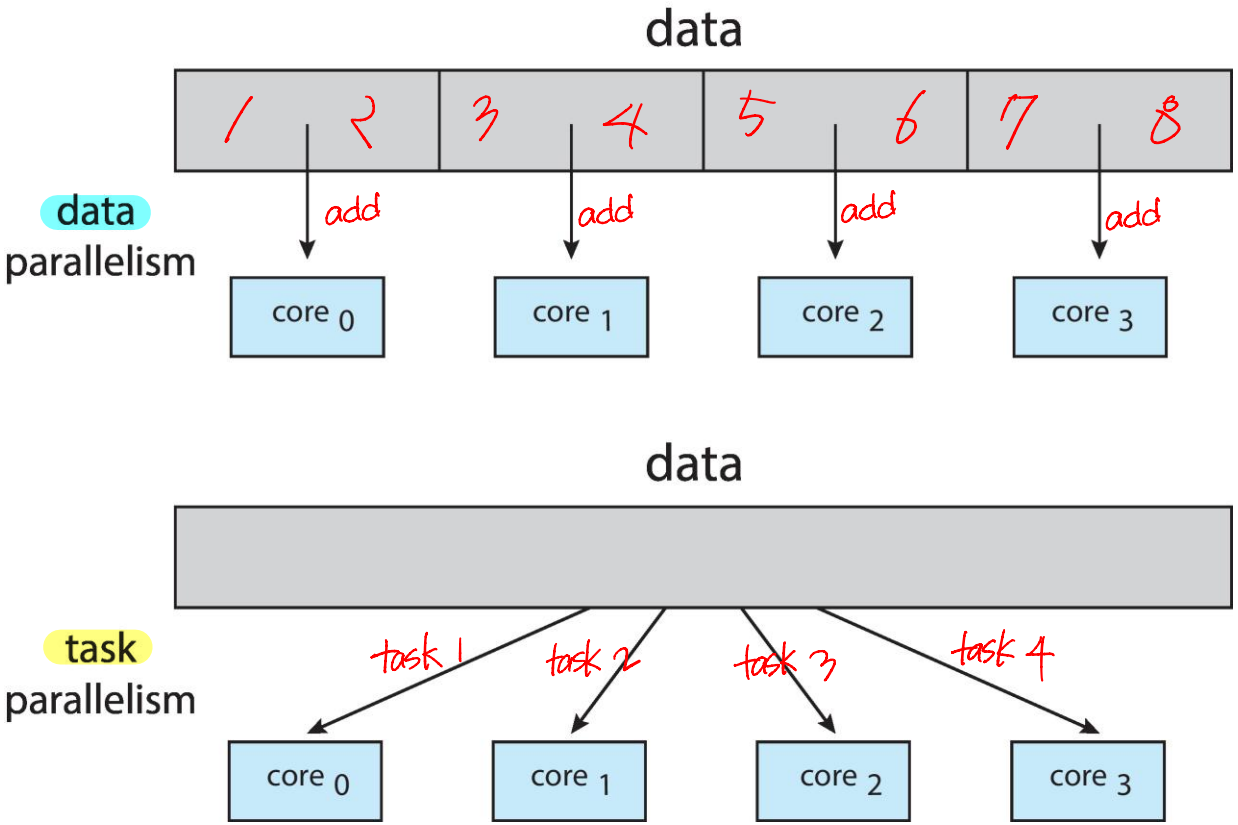
Multicore Programming

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

assumed to be divided by infinite number of task

S is serial portion, 1-S is parallel 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*

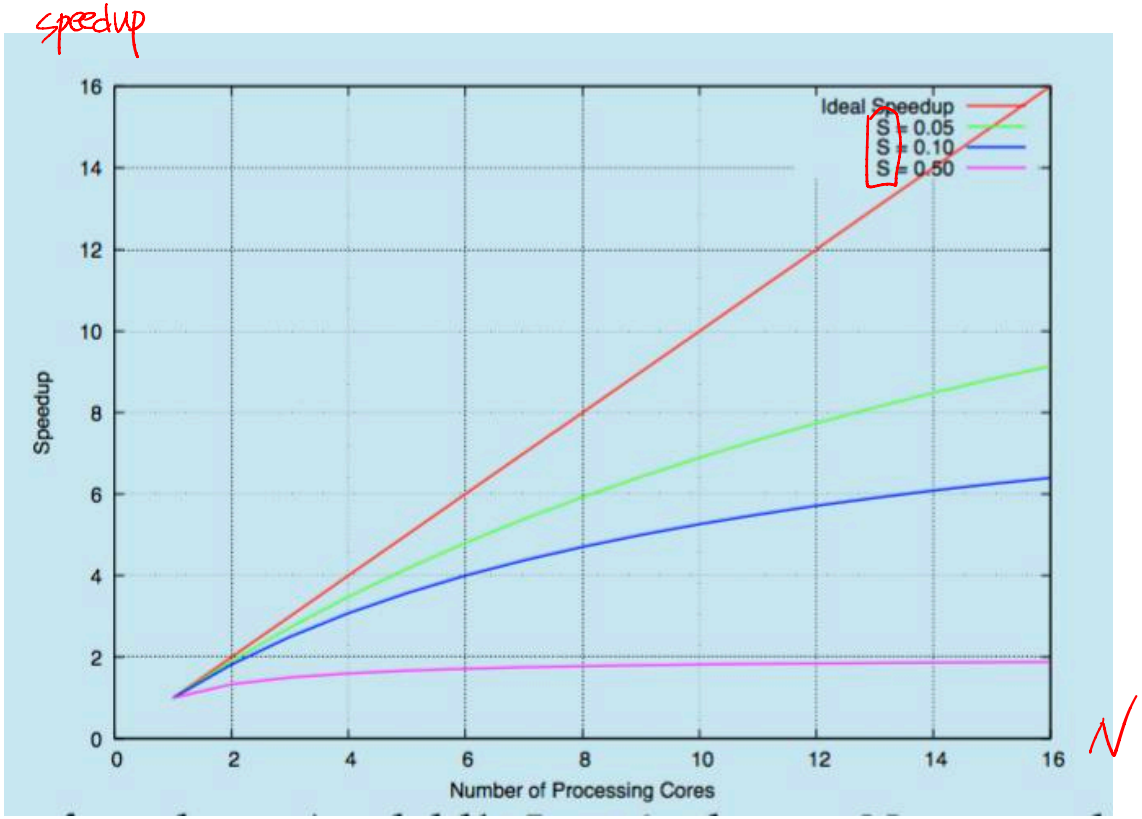
Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Amdahl's Law

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

S: the portion of the application that must be performed serially

N: the number of cores



User Threads and Kernel Threads

User threads - management done by user-level threads library

Three primary thread libraries:

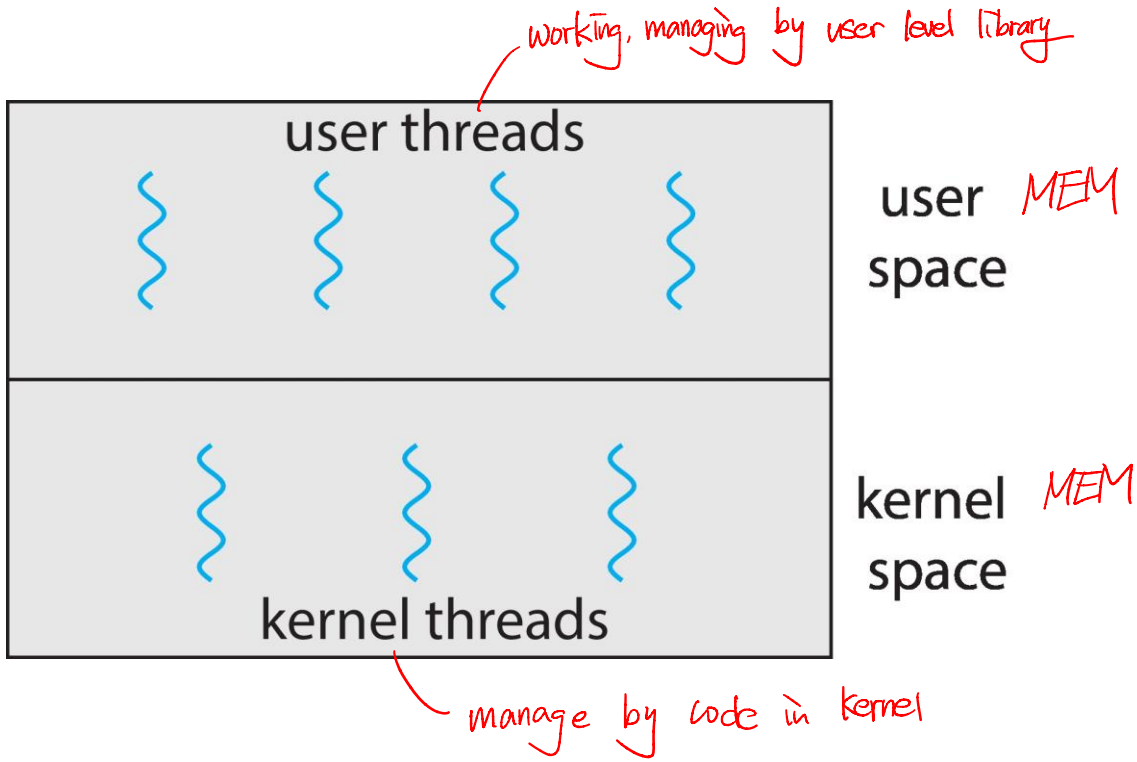
- POSIX **Pthreads**
- Windows threads
- Java threads

Kernel threads - Supported by the Kernel

Examples – virtually all general purpose operating systems, including:

- Windows
- Linux
- Mac OS X
- iOS
- Android

User and Kernel Threads



Multithreading Models

#1 Many-to-One

#2 One-to-One

#3 Many-to-Many

Many-to-One

#1

Many user-level threads mapped to single kernel thread

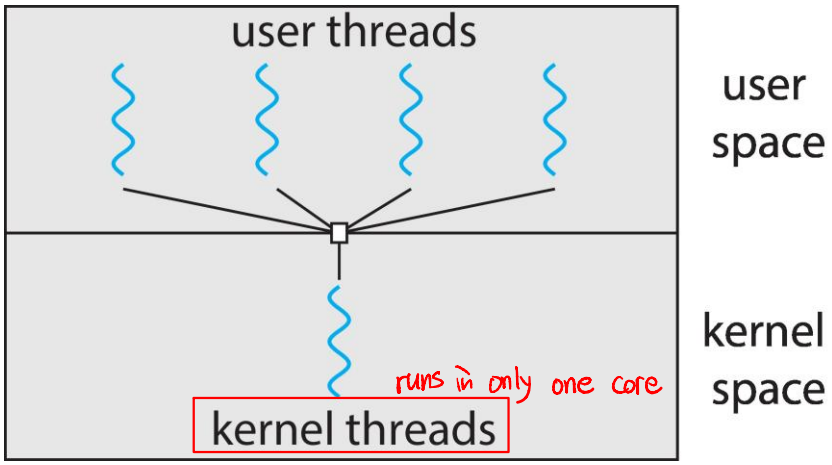
One thread blocking causes all to block — wait until completing requested task

Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time ⇒ single kernel thread can use only one core.

Few systems currently use this model

Examples:

- **Solaris Green Threads**
- **GNU Portable Threads**



being allocated resources.
distribute the resources to mapped user threads

One-to-One

#1

Each user-level thread maps to kernel thread

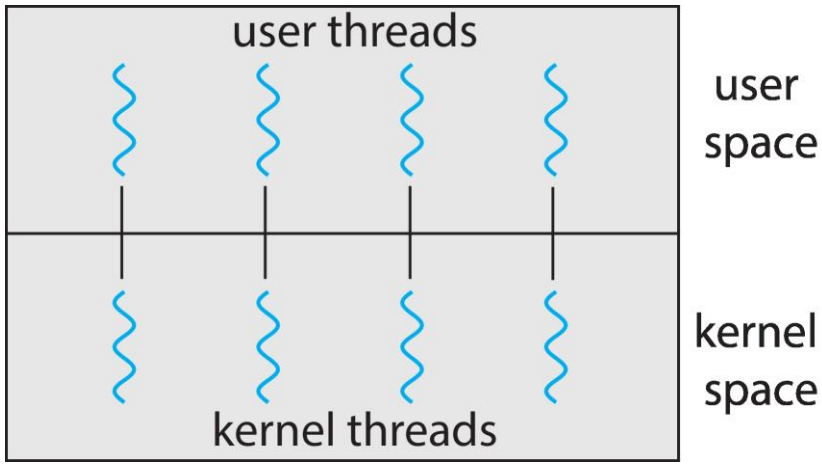
Creating a user-level thread creates a kernel thread ⇒ high cost for managing each threads

More concurrency than many-to-one

Number of threads per process sometimes restricted due to overhead

Examples

- Windows
- Linux



Many-to-Many Model

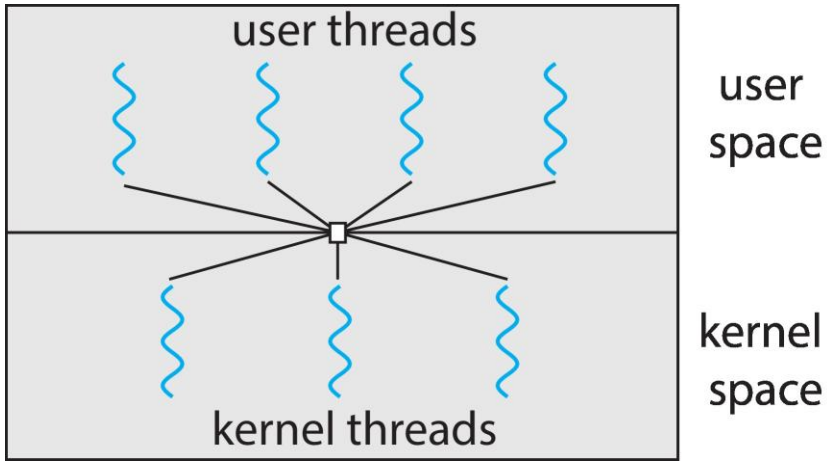
#7

Allows many user level threads to be mapped to many kernel threads

Allows the operating system to create a sufficient number of kernel threads ⇒ less resource consumption for managing threads than one-to-one model

Windows with the *ThreadFiber* package

Otherwise not very common

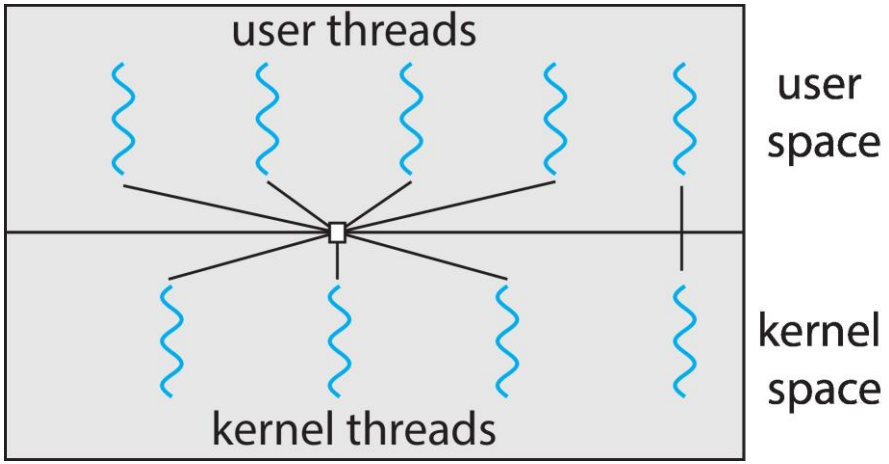


Two-level Model

M2M + 020

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

flexible



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 in Linux

May be provided either as user-level or kernel-level

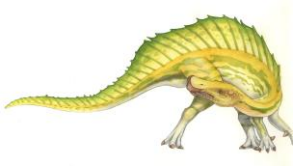
A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

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)

API defines only interface of function. (behavior)
input, output, functionality



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); ... after termination of thread
}
```

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);
```

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;
}
```

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);
}
```

Win64
Win32

Java Threads

Java threads are managed by the JVM ⇒ user level thread

Typically implemented using the threads model provided by underlying OS

Java threads may be created by:

- Extending Thread class
- Implementing the Runnable interface

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

Standard practice is to implement Runnable interface

Java Threads (Cont'd)

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

Rather than explicitly creating threads, Java also allows thread creation around the Executor interface: ⇒ thread manage routine 을 hidden. devs don't need to add managing code

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

The Executor is used as follows:

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

Java Executor Framework (Cont'd)

```
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);
   }
}
```

returns instance ⇒ parent can obtain result of child execution.

Java Executor Framework (cont'd)

```
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

Thread Pools

Create a number of threads in a pool where they await work

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:

```
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();
}
```

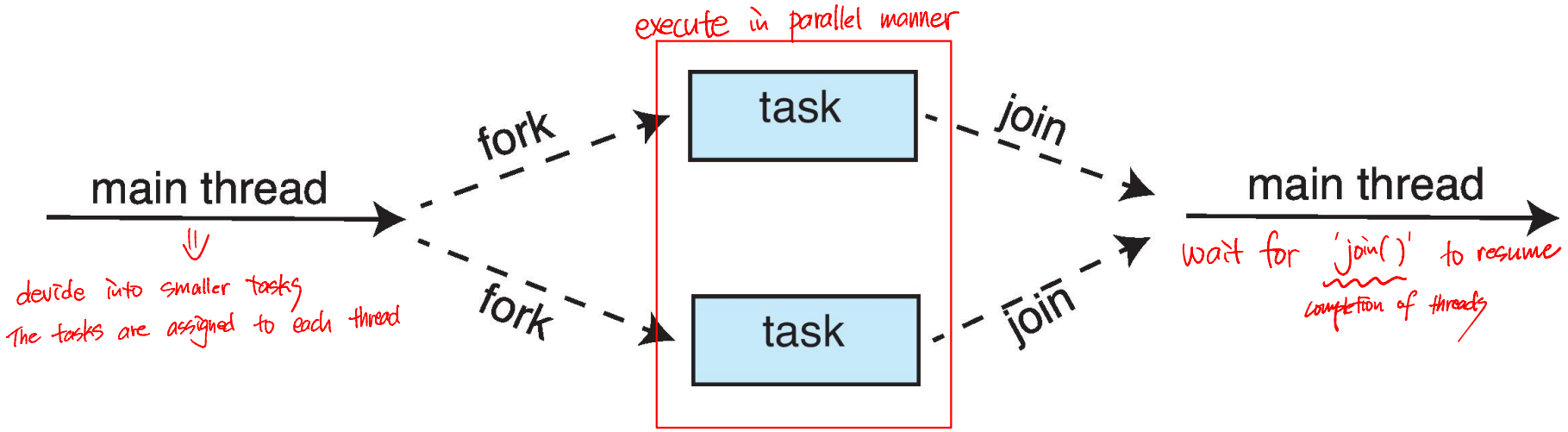
Fork-Join Parallelism

split task combine result

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

In the implicit model, threads are not constructed directly during the fork stage; parallel tasks are deginated

Library manages # of threads that are created and is also responsible for assigning tasks to threads

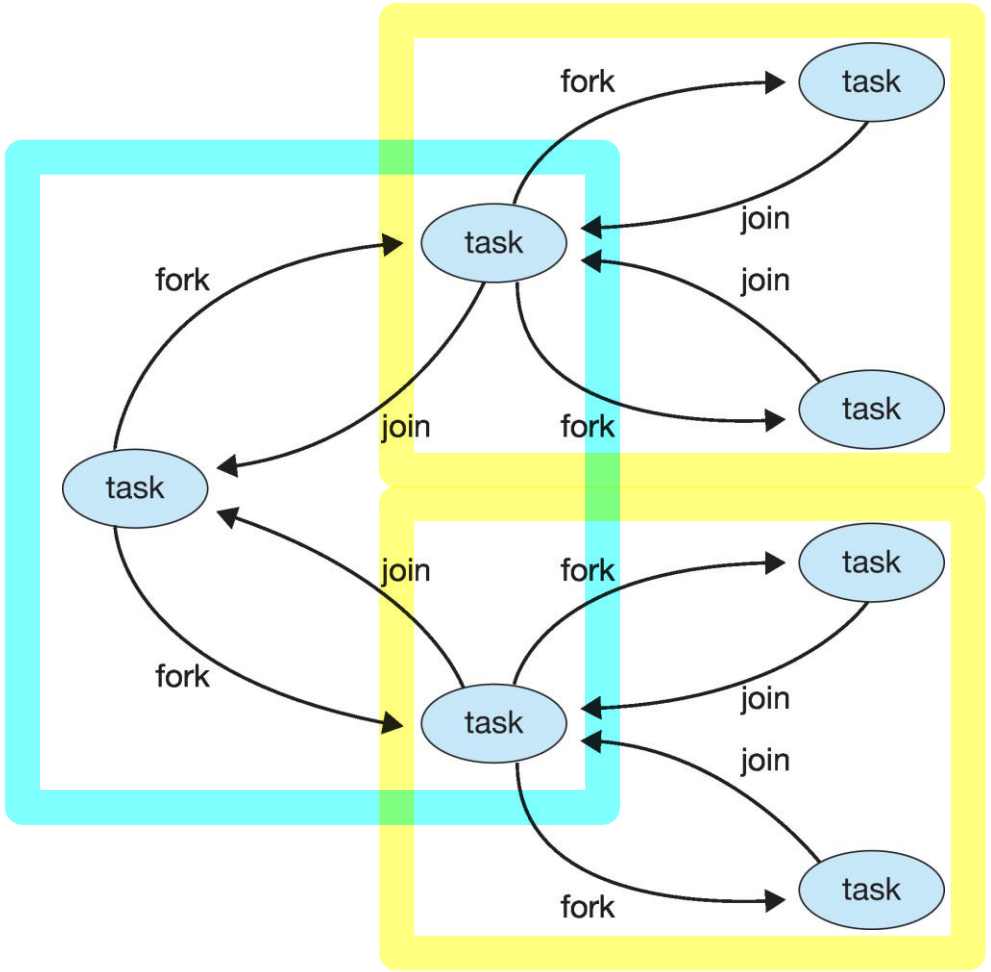


Fork-Join Parallelism

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



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();
      }
   }
}
```

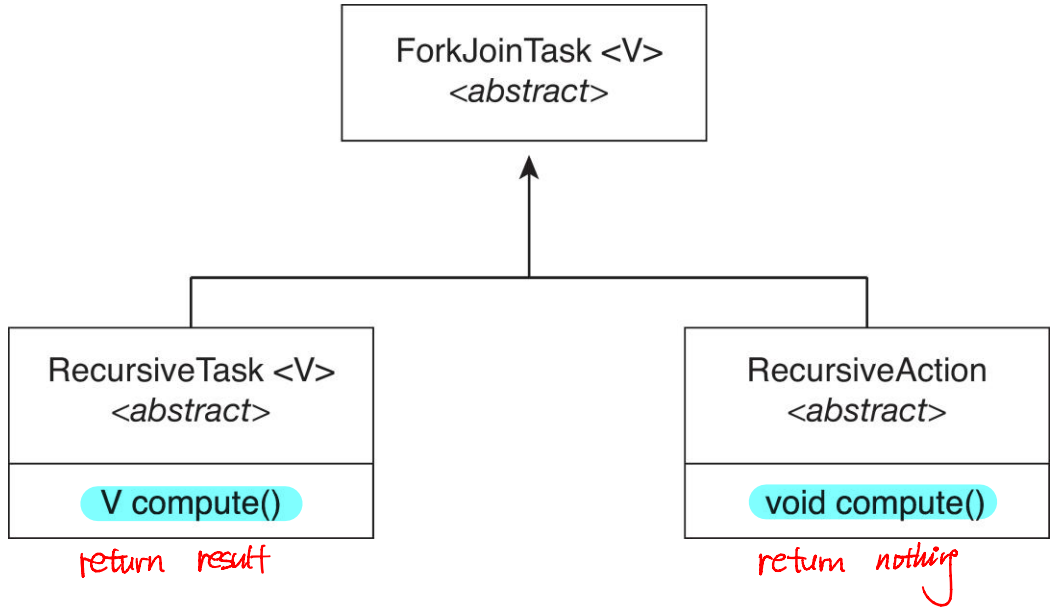
Fork-Join Parallelism in Java

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`

Create as many threads as there are cores

```
#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;
}
```

in parallel

if 8 core, print 8 times

Run the for loop in parallel

size of array

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

if 2 cores env & N=4,

core 0 — i=0, 2

core 1 — i=1, 3

Threading Issues

#1 Semantics of **fork()** and **exec()** system calls

#2 Signal handling

- Synchronous and asynchronous

#3 Thread cancellation of target thread

- Asynchronous or deferred

#4 Thread-local storage

Scheduler Activations

Semantics of fork() and exec()

#1

create new thread and copy the contents

Does **`fork()`** duplicate ① only the calling thread or ② all threads?

Some UNIXes have two versions of fork

`exec()` usually works as normal – replace the running process including all threads

Signal Handling

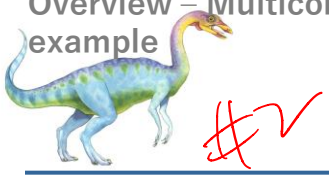
#7

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
  - inter-process
- A **signal handler** is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers:
    - default
    - user-defined
- 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

Signal Example – 1 (Keyboard Interrupt)

#2

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig) {
        printf("OUCH! - I got signal %d\n", sig);  // (A)
        (void) signal(SIGINT, SIG_DFL); // (B)
}

int main(void) {
        (void) signal(SIGINT, ouch); // (C)
        while(1) {
                printf("Hello World!\n"); // (D)
                sleep(1); // (E)
        }
}
```



Signal Example – 2 (Timer)

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
static int alarm_fired = 0;
void ding(int sig) {  alarm_fired = 1; }
int main(void)
{
   int pid;
   printf("alarm application starting\n");

     /* child: wait 5 sec, send
              SIGALRM to parent */

   if((pid = fork()) == 0) {
        sleep(5); // (A)
        kill(getppid(), SIGALRM); //(B)
        exit(0);
   }
```

deliver signal to parent

```
   /* parent: set signal hander, wait for alarm */
   printf("waiting for alarm to go off\n");
   (void) signal(SIGALRM, ding); // (C)
   /* suspend execution until signaled */
   pause();  // (D)
   if (alarm_fired) printf("Ding!\n"); // (E)

   printf("done\n");
   exit(0);
}
```

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);
```

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

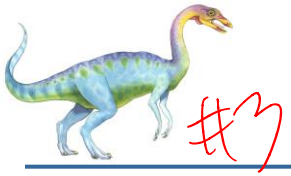
Default type is deferred

- Cancellation only occurs when thread reaches **cancellation point**
  - I.e. **`pthread_testcancel()`** → check delivery of cancellation
  - Then **cleanup handler** is invoked

On Linux systems, thread cancellation is handled through signals

#3

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>

using namespace std;

pthread_t t_thread;
void *ThreadTest(void *arg);

int main(int argc, char *argv[])
{
    int status = 0;
    int arg = 0;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    status = pthread_create(&t_thread, &attr, ThreadTest, &arg);
    pthread_attr_destroy(&attr);

    sleep(10);
    pthread_cancel(t_thread);
    cout << "pthread cancel" << endl;

    return 0;
}

void *ThreadTest(void *arg)
{
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);

    while(1)
    {
        sleep(1);
        cout << "pthread" << endl;
    }
}
```



Thread Cancellation in Java

Deferred cancellation uses the **`interrupt()`** method, which sets the interrupted status of a thread.

```
Thread worker;

. . .

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

A thread can then check to see if it has been interrupted:

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

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

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)

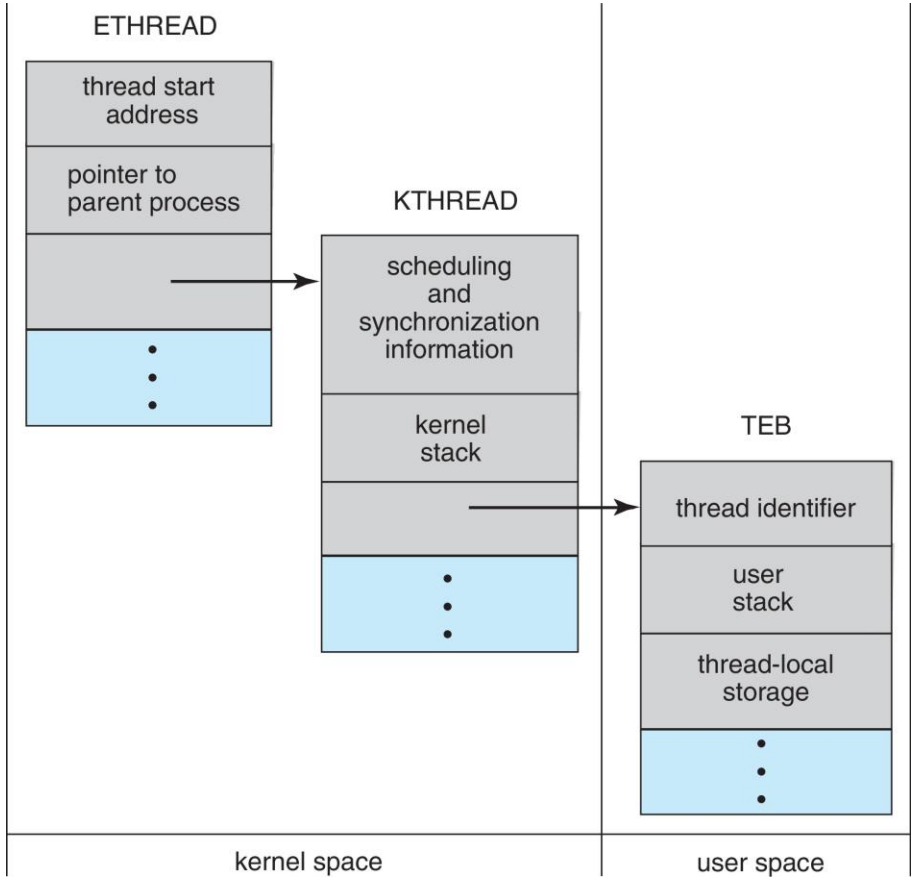
The register set, stacks, and private storage area are known as the **context** of the thread

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



Linux Threads

Linux refers to them as ***tasks*** rather than ***threads***

clone() — make process ↕ … resource sharing or not — make task

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)

task process

End of Chapter 4