CS3701 - Operating Systems

Chapter 3 - Threads and Concurrency

Department of Computer Science

College of Computer Engineering and Science

Prince Sattam bin Abdulaziz University

AY - 2025/2026

Prepared by Dr. Mahdi Khemakhem, Reviewed by Dr. Essra Aldessouky

Updated on January 22, 2026

Outline

1. Thread and Concurrency
1.1 Motivation
1.2 Thread
1.3 Concurrency
1.4 Key Takeaways

2. Multicore Programming
2.1 Multicore Systems
2.2 Multicore Programming Challenges
2.3 Types of Parallelism
2.4 Amdahl's Law
2.5 Key Takeaways

3. User and Kernel Threads
3.1 One-to-One Model
3.2 Many-to-One Model
3.3 Many-to-Many Model
3.4 Key Takeaways

4. Java Thread Library as Example
4.1 Thread Creation
4.2 Thread Management

Outline (Contd.)

4.3 Example: Multithreading for Summation
4.4 Key Takeaways

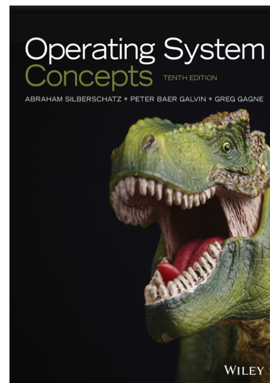
5. Exercices

Textbook Reference

Required Reading

Refer to Chapter 4 of the textbook:

- **Title:** *Operating System Concepts*
- **Authors:** Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne
- **Publisher:** John Wiley & Sons
- **Edition:** 10th Edition (2018)



Outline

1. Thread and Concurrency
1.1 Motivation
1.2 Thread
1.3 Concurrency
1.4 Key Takeaways

2. Multicore Programming

3. User and Kernel Threads

4. Java Thread Library as Example

5. Exercices

Motivation

Why threads in modern operating systems?

- Most applications are **multithreaded** to improve **responsiveness** and **performance**
- **Threads** run *within an application*, share the same *address space*
- **Process creation** is **heavy-weight**, **thread creation** is **light-weight**

Real-world examples:

- **Photo app**: Each **thread** generates a thumbnail from a **separate image**
- **Web browser**: One thread **displays** content, another **retrieves** data from **network**
- **Word processor**: Threads for **graphics**, **keystrokes**, **spell checking**

Key benefits:

- **Simplify code** and **increase efficiency**
- **Kernels** are multithreaded for better *resource management*
- **Leverage** *multicore systems* for **parallel CPU-intensive tasks**

Thread

Definition

A **thread** (also called **light-weight process**) is a *basic unit of CPU utilization* that comprises a **thread ID**, **program counter**, **register set**, and **stack**.
→**Each thread is associated with a process**.

Thread Components

- **Each thread maintains its own: Thread ID, Program Counter, Register Set, Stack.**
- **Threads within the same process share: Code Section, Data Section, Heap, Open Files, Signals.**

Thread vs Process

Key Differences

- **Process**: A *heavyweight* entity with its **own address space**
  - **Independent** memory space (code, data, heap, stack)
  - **Expensive** to create and manage
  - **Inter-process communication** (IPC) required for communication
  - **Context switching** between processes is **costly**
  - **Processes** provide *isolation* and *protection*
- **Thread**: A *lightweight* entity within a process
  - **Shares** address space with other threads in same process
  - **Cheap** to create and manage
  - **Direct communication** through shared memory
  - **Context switching** between threads is **fast**
  - **Threads** provide *efficiency* and *resource sharing*

Single vs Multi-threaded Process

Figure 1: Single vs Multi-threaded Process

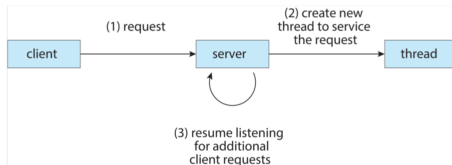
Multithreaded Server as Example

Web Server Example

Challenge: A **single-threaded** web server can only service *one client at a time*, causing **delays** in busy environments.

Solution: A **multithreaded** web server running as a *single process* that:

- **Creates** a **separate thread** for each client request.
- Each thread **handles one client** independently.
- Enables *concurrent processing* of multiple requests.
- **Improves responsiveness** and reduces waiting times.



Concurrency

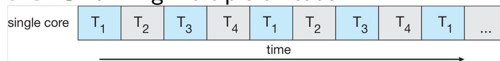
Definition

Concurrency is the ability of a system to *execute multiple tasks* by **interleaving** their execution, creating the *illusion of parallelism*.
→In a *single-core system*, concurrency is achieved through *time-sharing* and *very quick context switching*.

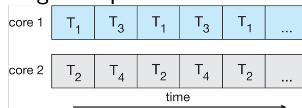
Concurrency vs Parallelism

Concurrency vs Parallelism

- **Concurrency**: *Multiple tasks make progress* over time
  - Tasks **interleave** on a **single CPU**.
  - Quick **context switching** creates illusion of simultaneous execution.
  - Example: **Single-core CPU** running multiple threads.



- **Parallelism**: *Multiple tasks execute simultaneously*
  - Tasks run **at the same time** on **multiple CPUs/cores**.
  - **True simultaneous** execution.
  - Example: **Multicore CPU** running multiple threads.



Concurrency Benefits

Benefits of Concurrency

Concurrency provides several advantages:

- **Responsiveness**: Application remains *responsive* even during long operations
- **Resource Sharing**: Threads share *memory and resources* efficiently
- **Economy**: Thread creation and management is *cheaper* than processes
- **Scalability**: Applications can *leverage multicore* architectures

Concurrency Challenges

Challenges of Concurrency

Concurrency introduces challenges:

- **Race Conditions**: Multiple threads accessing *shared data* simultaneously (see Chapter 5).
- **Deadlocks**: Threads waiting for *resources* held by each other (see Chapter 6).
- **Synchronization**: Need for *coordination* mechanisms (locks, semaphores) (see Chapter 5).
- **Debugging Complexity**: Non-deterministic behavior makes *testing difficult* .

Key Takeaways

Key Takeaways

- **Threads** are **lightweight units** of CPU utilization within a process, sharing the same address space.
- **Concurrency** allows multiple tasks to **make progress** over time, creating the *illusion of parallelism*.
- **Multithreading** improves application **responsiveness** and **performance**.
- **Concurrency** introduces challenges like **race conditions**, **deadlocks**, and **synchronization** issues.

Outline

1. Thread and Concurrency

2. Multicore Programming
2.1 Multicore Systems
2.2 Multicore Programming Challenges
2.3 Types of Parallelism
2.4 Amdahl's Law
2.5 Key Takeaways

3. User and Kernel Threads

4. Java Thread Library as Example

5. Exercices

Multicore Systems

Early Computing Systems

Before the advent of **multiprocessor** and **multicore** architectures, most computer systems had only a **single processor** and **OS** were designed to provide the **illusion of parallelism** by **rapidly switching between processes** (context switching), thereby allowing each process to make progress.

Definition

A **multicore system** is a **single computing component** with **two or more independent processing cores** on the same chip (CPU).
→These cores can execute **multiple tasks concurrently**, providing *true parallelism*.

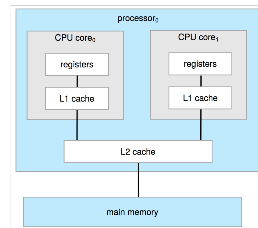


Figure 2: Multi-Core System

Multicore Systems Benefits

Key characteristics of multicore systems

- **Multiple Cores**: Each core can execute *independent instructions* simultaneously
- **Shared Memory**: Cores typically share **main memory** and **cache levels**
- **Increased Throughput**: Multiple processes/threads execute *in parallel*
- **Energy Efficiency**: Better *performance per watt* than single-core at higher speeds
- **Scalability**: Applications can *leverage multiple cores* for improved performance

Single-Core vs Multicore Systems

Comparison: Single-Core vs Multicore

- **Single-Core System**:
  - **One CPU core** executes instructions sequentially
  - **Concurrency** achieved through *time-sharing* and *context switching*
  - **Illusion of parallelism** but not true simultaneous execution
  - **Limited performance** for CPU-intensive tasks
- **Multicore System**:
  - **Multiple CPU cores** execute instructions simultaneously
  - **True parallelism** with *concurrent execution* on different cores
  - **Enhanced performance** for multithreaded applications
  - **Better resource utilization** and *faster task completion*

Multicore Programming

Definition

Multicore Programming is the practice of **designing** and **implementing** software that **effectively utilizes multiple processor cores** to achieve **better performance** and **efficiency**.
→It requires careful consideration of *task division*, *data sharing*, and *synchronization*.

Multicore Programming Challenges

Key Considerations

Multicore programming requires addressing several key challenges:

- **Task Parallelism**: Dividing application into *concurrent tasks*
- **Data Parallelism**: Distributing *data across cores* for parallel processing
- **Load Balancing**: Ensuring *equal work distribution* among cores
- **Synchronization**: Coordinating *access to shared resources* (**see Chapter 5**)

Multicore Programming Challenges (Contd.)

Five Programming Challenges

Developers face these challenges when programming multicore systems:

1. **Identifying Tasks**: Analyzing applications to find areas that can be *divided into separate tasks* that run in parallel
2. **Balance**: Ensuring tasks perform *equal work of equal value* to avoid idle cores
3. **Data Splitting**: Dividing data *accessed and manipulated* by tasks to run on separate cores
4. **Data Dependency**: Ensuring *synchronization* when tasks access shared data
5. **Testing and Debugging**: Dealing with *non-deterministic behavior* and *race conditions* (see Chapter 5) in parallel execution

Data Parallelism vs Task parallelism

Data Parallelism:

- *Same operation* performed on **different subsets of data** simultaneously
- Data is **distributed across multiple cores**
- Each core executes the *same task* on its data subset
- Example: **Vector addition** - each core processes different elements

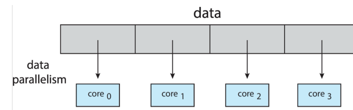


Figure 3: Data Parallelism

Task Parallelism:

- *Different operations* performed on **same or different data** simultaneously
- Tasks are **distributed across multiple cores**
- Each core executes a *different task*
- Example: **Multimedia application** - one core decodes audio, another processes video

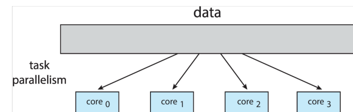


Figure 4: Task Parallelism

Data Parallelism Example

Data Parallelism: Array Summation

Problem: Sum elements of a large array with N elements.

Data Parallel Approach on 4 cores:

- **First Step**: Divide array into 4 subsets
- **Second Step**: Each core sums its subset
  - **Core 1**: Sums elements 0 to $N/4 - 1$
  - **Core 2**: Sums elements $N/4$ to $N/2 - 1$
  - **Core 3**: Sums elements $N/2$ to $3N/4 - 1$
  - **Core 4**: Sums elements $3N/4$ to $N - 1$
- **Final Step**: Combine partial sums from all cores

Result: *Same operation* (summation) on *different data subsets*, achieving *speedup* of **nearly 4x** *(ideally)* (**see Amdahl's Law**).

Task Parallelism Example

Task Parallelism: Web Server

Problem: Handle multiple client requests simultaneously.

Task Parallel Approach:

- **Thread 1**: Accepts incoming **client connections**
- **Thread 2**: Processes **HTTP GET requests**
- **Thread 3**: Processes **HTTP POST requests**
- **Thread 4**: Handles **database queries**
- **Thread 5**: Manages **logging and monitoring**

Result: *Different tasks* executed *concurrently* by **different threads** on **different cores**, improving **responsiveness** and **throughput**.

Amdahl's Law

Definition

Amdahl's Law provides a formula to calculate the *maximum speedup* achievable when parallelizing a program, considering that only a **portion of the program can be parallelized**.

→It highlights the *limitation* imposed by the *sequential portion* of the code.

Amdahl's Law Formula

Amdahl's Law Formula

Let:

- S = **Fraction of the program that must be executed serially** (*sequential portion*). $S \in [0, 1]$
- $P = 1 - S$ *parallel portion*
- N = **Number of processing cores**. $N \in \mathbb{N}^+$

Speedup formula:

$$\text{Speedup} \leq \frac{1}{S + \frac{(1-S)}{N}} = \frac{1}{S + \frac{P}{N}}$$

As $N \to \infty$: $\text{Speedup} \to \frac{1}{S}$

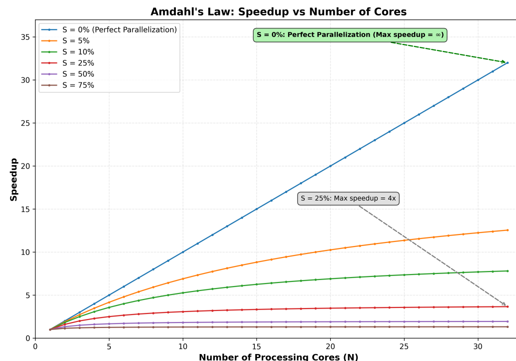


Figure 5: Amdahl's Law

Amdahl's Law: Implications

Key Implications

Amdahl's Law reveals important insights:

- **Serial Bottleneck**: The *sequential portion* S limits maximum speedup
- **Diminishing Returns**: Adding more cores provides *diminishing benefits* if S is large
- **Maximum Speedup**: Even with infinite cores, speedup cannot exceed $\frac{1}{S}$
- **Optimization Focus**: Best to *minimize serial portion* S first

Example Scenarios

- If $S = 0.75$ (75% serial): Max speedup $= \frac{1}{0.75} =$ **1.33x** (even with 1000 cores!)
- If $S = 0.50$ (50% serial): Max speedup $= \frac{1}{0.50} =$ **2x**
- If $S = 0.25$ (25% serial): Max speedup $= \frac{1}{0.25} =$ **4x**
- If $S = 0.10$ (10% serial): Max speedup $= \frac{1}{0.10} =$ **10x**
- If $S = 0.05$ (5% serial): Max speedup $= \frac{1}{0.05} =$ **20x**

Conclusion: *Reducing sequential portion* is crucial for scalability!

Amdahl's Law: Numerical Example

Calculating Speedup

Scenario: An application has **75% parallelizable code** and **25% sequential code** ($S = 0.25$).

Speedup with different numbers of cores:

- **2 cores**: Speedup $= \frac{1}{0.25+\frac{0.75}{2}} = \frac{1}{0.25+0.375} = \frac{1}{0.625} \approx$ **1.6x**
- **4 cores**: Speedup $= \frac{1}{0.25+\frac{0.75}{4}} = \frac{1}{0.25+0.1875} = \frac{1}{0.4375} \approx$ **2.29x**
- **8 cores**: Speedup $= \frac{1}{0.25+\frac{0.75}{8}} = \frac{1}{0.25+0.09375} = \frac{1}{0.34375} \approx$ **2.91x**
- ∞ **cores**: Speedup $= \frac{1}{0.25} =$ **4x** (maximum possible)

Observation: Doubling cores from 4 to 8 only increases speedup from 2.29x to 2.91x!

Key Takeaways

Key Takeaways

- **Multicore systems** have multiple processing cores on a single chip, enabling **true parallelism**.
- **Multicore programming** involves designing software to effectively utilize multiple cores, addressing challenges like **task division**, **data sharing**, and **synchronization**.
- **Data parallelism** involves performing the same operation on different data subsets, while **task parallelism** involves executing different tasks **concurrently**.
- **Amdahl's Law** quantifies the **maximum speedup** achievable through **parallelization**, emphasizing the impact of the **sequential portion** of code on overall performance.

Outline

1. Thread and Concurrency

2. Multicore Programming

3. User and Kernel Threads
3.1 One-to-One Model
3.2 Many-to-One Model
3.3 Many-to-Many Model
3.4 Key Takeaways

4. Java Thread Library as Example

5. Exercices

User and Kernel Threads

User vs Kernel Threads

Support for **threads** may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**.

- **User threads**: Managed by a **user-level thread library** without kernel support.
- **Kernel threads**: Managed directly by the **operating system kernel**.

→ **Ultimately, a relationship must exist between user threads and kernel threads.**

→ There are three different models to establish this relationship: **one-to-one**, **many-to-one**, and **many-to-many**.

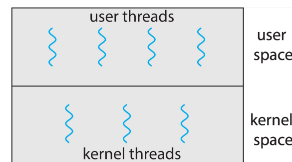


Figure 6: User vs Kernel Threads

One-to-One Model

One-to-One Thread Model

- Each **user-level thread** maps to a **single kernel-level thread**.
- The operating system *creates a kernel thread* for each user thread.
- Provides *better concurrency* as multiple threads can run in parallel on multiple processors.
- However, it can lead to *high overhead* due to the large number of kernel threads.

Example: Windows and Linux operating systems use the one-to-one threading model.

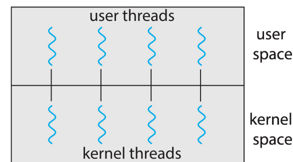


Figure 7: One-to-One Thread Model

Many-to-One Model

Many-to-One Thread Model

- Multiple **user-level threads** map to a **single kernel-level thread**.
- The operating system is unaware of the existence of user threads.
- This model is simple and has low overhead, but it does not take advantage of multiprocessor systems.
- If one thread makes a blocking system call, the entire process is blocked.

Example: Early versions of Java Virtual Machine (JVM) used the many-to-one threading model.

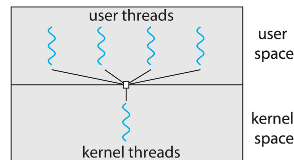


Figure 8: Many-to-One Thread Model

Many-to-Many Model

Many-to-Many Thread Model

- Multiple **user-level threads** map to **multiple kernel-level threads**.
- The operating system can create a sufficient number of kernel threads to handle user threads.
- This model provides flexibility and can take advantage of multiprocessor systems.
- It allows for better resource utilization and can avoid blocking issues.

Example: Solaris operating system uses the many-to-many threading model.

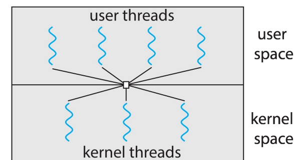


Figure 9: Many-to-Many Thread Model

Key Takeaways

Key Takeaways

- **User threads** are managed by a user-level thread library, while **kernel threads** are managed by the operating system kernel.
- There are three models to map user threads to kernel threads: **one-to-one**, **many-to-one**, and **many-to-many**.
- The **one-to-one model** provides better concurrency but has higher overhead.
- The **many-to-one model** is simple but does not utilize multiprocessor systems effectively.
- The **many-to-many model** offers flexibility and better resource utilization.

Outline

1. Thread and Concurrency

2. Multicore Programming

3. User and Kernel Threads

4. Java Thread Library as Example
4.1 Thread Creation
4.2 Thread Management
4.3 Example: Multithreading for Summation
4.4 Key Takeaways

5. Exercices

Java Thread Library

Java Thread Library Overview

The **Java Thread Library** provides a high-level API for creating and managing threads in Java applications. It supports both **user-level** and **kernel-level** threading models, allowing developers to create multithreaded applications easily.

Key Features of Java Thread Library

- Java uses the **one-to-one threading model**, where each **Java thread** maps directly to a **native OS thread**.
- The main classes for thread creation and management are `java.lang.Thread` and `java.lang.Runnable`.
- Java provides essential mechanisms for thread **synchronization** and **communication**, supporting safe interaction among concurrent threads (**will be covered in Chapter 5**).
- Advanced features such as thread **pools**, **executors**, and **concurrency utilities** are available in the `java.util.concurrent` package (beyond this course scope).

Thread Creation in Java

Creating Threads in Java

Java provides two primary methods for creating threads:

1. **Extending Thread Class**: Create a class that extends `Thread` and override the `run()` method
2. **Implementing Runnable Interface**: Create a class that implements `Runnable` interface and define the `run()` method

Key Points:

- The `run()` method **contains the code to be executed by the thread**
- Call `start()` method **to begin thread execution** (*not* `run()`)
- `start()` **creates a new thread** and **calls `run()` in that thread**
- Implementing `Runnable` is *preferred* as it **allows extending other** classes

Sysntax for Thread Creation using Thread Class

Thread Creation using Thread Class

```
public class ThreadExample extends Thread {
    @Override
    public void run() {
        // Code to be executed in the thread
    }
}
class Main {
    public static void main(String[] args) {
        ThreadExample thread = new ThreadExample();
        thread.start(); // Start the thread
    }
}
```

Syntax for Thread Creation using Runnable Interface

Thread Creation using Runnable Interface

```
1  public class RunnableExample implements Runnable {
2      @Override
3      public void run() {
4          // Code to be executed in the thread
5      }
6  }
7  class Main {
8      public static void main(String[] args) {
9          RunnableExample runnable = new RunnableExample();
10         Thread thread = new Thread(runnable);
11         thread.start(); // Start the thread
12     }
13 }
```

Thread Management in Java

Key Thread Management Methods

Java provides several methods for managing thread lifecycle:

- `start()`: **Begins thread execution** by calling `run()` in new thread
- `join()`: **The main process waits for thread to complete** before continuing
- `sleep(milliseconds)`: **Pauses thread execution** for specified time
- `interrupt()`: **Interrupts a thread**, causing it to throw `InterruptedException` if blocked
- `isAlive()`: **Checks if thread is still running**
- `setName()/getName()`: **Sets/gets thread name** for identification
- `setPriority()`: **Sets thread priority** (1-10, default 5; platform-dependent)
- `wait()`: **Causes thread to wait** until notified (**see Chapter 5**)
- `notify()/notifyAll()`: **Wakes waiting threads** (**see Chapter 5**)
- `yield()`: **Hints thread scheduler to pause current thread and allow other threads to execute** (platform-dependent behavior)
  → differs from `wait()` method in that it does not release locks (**see Chapter 5**)

Thread States and Lifecycle

Thread Lifecycle States

- **NEW**: Thread created but not yet started
- **RUNNABLE**: Thread executing or ready to execute
- **BLOCKED**: Thread blocked waiting for monitor lock (**see Chapter 5**)
- **WAITING**: Thread waiting indefinitely for another thread to perform a particular action
- **TIMED_WAITING**: Thread waiting for specified time
- **TERMINATED**: Thread completed execution

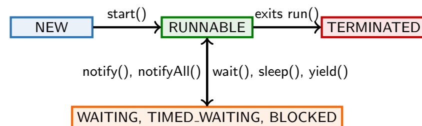


Figure 10: Simplified Thread Lifecycle

Thread Management Methods in Java

Using sleep() and join()

```
1  class ThreadManagementExample extends Thread {
2      @Override
3      public void run() {
4          try {
5              for (int i = 1; i <= 5; i++) {
6                  System.out.println("Worker thread running: step " + i);
7                  Thread.sleep(500); // Pause execution for 500 ms
8              }
9          } catch (InterruptedException e) {
10             // Triggered when interrupt() is called
11             System.out.println("Worker thread interrupted!");
12         }
13     }
14 }
15 public class Main {
16     public static void main(String[] args) {
17         ThreadManagementExample th = new ThreadManagementExample();
18         th.start(); // Start the worker thread
19         try {
20             th.join();      // Wait until the thread terminates
21         } catch (InterruptedException e) {
22             System.out.println("Main interrupted");
23         }
24         System.out.println("Main thread finished execution");
25     }}
```

Thread Management Methods in Java (Contd.)

Using isAlive(), setName(), and setPriority()

```
class ThreadInfoExample extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(getName() + " running: step " + i);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        ThreadInfoExample th = new ThreadInfoExample();
        th.setName("WorkerThread");
        th.setPriority(Thread.MAX_PRIORITY); // Set highest priority
        th.start(); // Start the thread
        if (th.isAlive()) {
            System.out.println(th.getName() + " is alive and running.");
        }
    }
}
```

Example: Multithreading for Summation (Data Parallelism)

Problem Statement

Objective: Create a Java program that calculates the sum of the first N natural numbers $\sum_{i=1}^{N} i$ using Multithreading (K threads, $K \leq N$). Implement two approaches:

1. **Extending the Thread class**
2. **Implementing the Runnable interface**

Requirements:

- The program should accept an integer N and K as input.
- Each thread should compute a partial sum of a subset of N/K numbers.
- The main thread should combine the partial sums from all threads to get the final result.
- Ensure proper thread management using `start()` and `join()` methods.

Solution 1: Extending Thread Class

Solution 1: Summation Using Thread Class

```
class SumThread extends Thread {
    private int start, end, sum;
    public SumThread(int start, int end) {
        this.start = start;
        this.end = end;
    }
    public int getSum() {
        return sum;
    }
    @Override
    public void run() {
        sum = 0;
        for (int i = start; i <= end; i++) {
            sum += i;
        }
    }
}
```

Solution 1: Extending Thread Class (Contd.)

Solution 1: Summation Using Thread Class (Contd.)

```
1  public class ThreadExample {
2      public static void main(String[] args) throws InterruptedException {
3          int N = 100; // Sum of first N natural numbers
4          int K = 4;   // Number of threads
5          SumThread[] threads = new SumThread[K];
6          int range = N / K;
7          for (int i = 0; i < K; i++) {
8              int start = i * range + 1;
9              int end = (i == K - 1) ? N : (i + 1) * range;
10             threads[i] = new SumThread(start, end);
11             threads[i].start(); // Start the thread
12         }
13         int totalSum = 0;
14         for (int i = 0; i < K; i++) {
15             threads[i].join(); // Wait for thread to finish
16             totalSum += threads[i].getSum();
17         }
18         System.out.println("Total Sum = " + totalSum);
19     }
20 }
```

Solution 2: Implementing Runnable Interface

Solution 2: Summation Using Runnable Interface

```
1  class SumRunnable implements Runnable {
2      private int start, end, sum;
3      public SumRunnable(int start, int end) {
4          this.start = start;
5          this.end = end;
6      }
7      public int getSum() {
8          return sum;
9      }
10     @Override
11     public void run() {
12         sum = 0;
13         for (int i = start; i <= end; i++) {
14             sum += i;
15         }
16     }
17 }
```

Solution 2: Implementing Runnable Interface (Contd.)

Solution 2: Summation Using Runnable Interface (Contd.)

```
1  public class RunnableExample {
2      public static void main(String[] args) throws InterruptedException {
3          int N = 100; // Sum of first N natural numbers
4          int K = 4;   // Number of threads
5          SumRunnable[] runnables = new SumRunnable[K];
6          Thread[] threads = new Thread[K];
7          int range = N / K;
8          for (int i = 0; i < K; i++) {
9              int start = i * range + 1;
10             int end = (i == K - 1) ? N : (i + 1) * range;
11             runnables[i] = new SumRunnable(start, end);
12             threads[i] = new Thread(runnables[i]);
13             threads[i].start(); // Start the thread
14         }
15         int totalSum = 0;
16         for (int i = 0; i < K; i++) {
17             threads[i].join(); // Wait for thread to finish
18             totalSum += runnables[i].getSum();
19         }
20         System.out.println("Total Sum = " + totalSum);
21     }
22 }
```

Key Takeaways - Java Threading

Key Takeaways

- Java provides **two main approaches** for thread creation: extending `Thread` class or implementing `Runnable` interface.
- Thread lifecycle includes states: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, and TERMINATED.
- Key thread management methods: `start()`, `join()`, `sleep()`, `interrupt()`, `isAlive()`, `setName()`, and `setPriority()`.
- Proper use of these methods ensures effective thread management and synchronization.
- Using `Runnable` is often preferred for flexibility and code reuse.
- Multithreading can significantly improve performance for **data-parallel tasks** and **task-parallel tasks**.

Outline

1. Thread and Concurrency
2. Multicore Programming
3. User and Kernel Threads
4. Java Thread Library as Example
5. Exercices

Exercise 1: Amdahl's Law Calculation

Exercise 1: Amdahl's Law Calculation

An application has a **60% parallelizable code** and **40% sequential code** ($S = 0.40$).

1. Calculate the expected speedup when running the application on: **2 cores**, **4 cores**, and **8 cores**.
2. What is the maximum speedup achievable with an infinite number of cores?

Exercise 1: Amdahl's Law Calculation - Solution

Exercise 1: Amdahl's Law Calculation - Solution

Given: $S = 0.40$ (40% sequential), $P = 0.60$ (60% parallelizable)

Using Amdahl's Law: Speedup $= \frac{1}{S+\frac{P}{N}}$

1. Expected speedup for different number of cores:
   - **2 cores**: Speedup $= \frac{1}{0.40+\frac{0.60}{2}} = \frac{1}{0.40+0.30} = \frac{1}{0.70} \approx$ **1.43x**
   - **4 cores**: Speedup $= \frac{1}{0.40+\frac{0.60}{4}} = \frac{1}{0.40+0.15} = \frac{1}{0.55} \approx$ **1.82x**
   - **8 cores**: Speedup $= \frac{1}{0.40+\frac{0.60}{8}} = \frac{1}{0.40+0.075} = \frac{1}{0.475} \approx$ **2.11x**
2. Maximum speedup with infinite cores:

$$\text{Speedup} = \frac{1}{S + \frac{P}{\infty}} = \frac{1}{0.40 + 0} = \frac{1}{0.40} = \mathbf{2.5x}$$

→ **The maximum speedup is limited by the sequential portion of the code. Even with infinite cores, the speedup cannot exceed 2.5x due to the 40% sequential code.**

Exercise 2: Java Thread Library - Task Parallelism

Exercise 2: Java Thread Library - Task Parallelism

Given an array of N numbers, create a Java program that performs the following tasks using multithreading:

- Task 1: Calculate the sum of the N numbers.
- Task 2: Find the maximum number in the array.
- Task 3: Calculate the average of the N numbers (using the sum from Task 1).

Implement each task in a separate thread using both approaches:

1. Extending the Thread class (*Given in the solution*)
2. Implementing the Runnable interface (*Todo by the student*)

Hints: (i) Use appropriate thread management methods to ensure that the main program waits for all threads to complete before printing the results. (ii) Task 3 should wait for Task 1 to complete, then use its result to calculate the average (demonstrating thread collaboration).

Exercise 2: Java Thread Library - Task Parallelism - Solution

Exercise 2: Java Thread Library - Task Parallelism - Solution

Task 1: Calculate the sum of N numbers (Extending Thread Class)

```java
class SumThread extends Thread {
    private int[] numbers;
    private volatile int sum;  // volatile ensures visibility across threads
    public SumThread(int[] numbers) {
        this.numbers = numbers;
    }
    public int getSum() {
        return sum;
    }
    @Override
    public void run() {
        sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        System.out.println("Sum calculation completed by " + getName());
    }
}
```

Exercise 2: Java Thread Library - Task Parallelism - Solution

Exercise 2: Java Thread Library - Task Parallelism - Solution (Contd.)

Task 2: Find the maximum number in the array (Extending Thread Class)

```
class MaxThread extends Thread {
    private int[] numbers;
    private int max;
    public MaxThread(int[] numbers) {
        this.numbers = numbers;
    }
    public int getMax() {
        return max;
    }
    @Override
    public void run() {
        max = Integer.MIN_VALUE;
        for (int num : numbers) {
            if (num > max) {
                max = num;
            }
        }
        System.out.println("Max calculation completed by " + getName());
    }
}
```

Exercise 2: Java Thread Library - Task Parallelism - Solution

Exercise 2: Java Thread Library - Task Parallelism - Solution (Contd.)

Task 3: Calculate average using sum from Task 1 (Extending Thread Class)

```
class AverageThread extends Thread {
    private int[] numbers;
    private SumThread sumThread;  // Reference to Task 1 for collaboration
    private double average;
    public AverageThread(int[] numbers, SumThread sumThread) {
        this.numbers = numbers;
        this.sumThread = sumThread;
    }
    public double getAverage() {
        return average;
    }
    @Override
    public void run() {
        try {
            sumThread.join();  // Wait for sum calculation to complete
            int sum = sumThread.getSum();  // Reuse the sum
            average = (double) sum / numbers.length;
            System.out.println("Average calculation completed by " + getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
  }
```

Exercise 2: Java Thread Library - Task Parallelism - Solution

Exercise 2: Java Thread Library - Task Parallelism - Solution (Main Class)

Main Class to Execute the Tasks

```
public class TaskParallelismExample {
    public static void main(String[] args) throws InterruptedException {
        int[] numbers = {12, 45, 7, 23, 89, 34, 56, 78, 90, 11};
        // Create threads
        SumThread sumThread = new SumThread(numbers);
        MaxThread maxThread = new MaxThread(numbers);
        AverageThread averageThread = new AverageThread(numbers, sumThread);
        // Start all threads (AverageThread will wait for SumThread internally)
        sumThread.start();
        maxThread.start();
        averageThread.start();
        // Wait for all threads to complete
        sumThread.join();
        maxThread.join();
        averageThread.join();
        // Print results
        System.out.println("\nResults:");
        System.out.println("Sum: " + sumThread.getSum());
        System.out.println("Max: " + maxThread.getMax());
        System.out.println("Average: " + averageThread.getAverage());
    }
}
```

Exercise 2: Java Thread Library - Task Parallelism

Key Takeaways

Key Learning Points from Exercise 2:

- **Thread Collaboration:** AverageThread demonstrates how threads can collaborate by waiting for another thread's result using `join()`.
- **Visibility:** The `volatile` keyword ensures that the sum value is visible across threads without synchronization.
- **Extending vs Implementing:**
  - Extending `Thread`: Simpler but limits inheritance
  - Implementing `Runnable`: More flexible, better design
- **Join Method:** Ensures main thread waits for all worker threads to complete before accessing results.
- **Efficiency:** By sharing the sum calculation, we avoid duplicate work while demonstrating thread coordination.

End of Chapter 3