Chapter 4

Answer of All the practice Questions and Codes Explained

4.1 Provide three programming examples in which multithreading provides better performance than a single-threaded solution.

From p. 161:

1. "An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image."
2. "A web browser might have one thread display images or text while another thread retrieves data from the network."
3. "A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background."

4.2 Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

From p. 164 (Amdahl's Law formula):
"speedup ≤ 1/(S + (1-S)/N)"
Where S = serial portion (40% or 0.4), N = number of cores
a) For 2 cores: 1/(0.4 + 0.6/2) = 1.43x
b) For 4 cores: 1/(0.4 + 0.6/4) ≈ 1.82x

4.3 Does the multithreaded web server described in Section 4.1 exhibit task or data parallelism?

From p. 161-162 and Figure 4.2:
The web server exhibits task parallelism because "the server will create a separate thread that listens for client requests" and each thread handles different client requests (different tasks).

4.4 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

From p. 166:

1. "User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system."
2. "The entire process will block if a thread makes a blocking system call" in user-level threads (many-to-one model), while kernel threads allow other threads to run when one blocks.

From p. 167-168:
User-level better when need many threads (more efficient), kernel-level better when need true parallelism or don't want whole process blocked.

4.5 Describe the actions taken by a kernel to context-switch between kernel-level threads.

From p. 166 (implied):

1. Save state of current thread (registers, PC, stack pointer)
2. Schedule new thread (via kernel scheduler)
3. Restore state of new thread
4. Resume execution of new thread

4.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

From p. 160:
Thread creation uses:

- Thread ID
- Program counter
- Register set
- Stack

Differs from process creation (p. 160):
"Threads share with other threads belonging to the same process its code section, data section,

and other operating-system resources, such as open files and signals," while processes get their own copies.

4.7 Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.

From p. 193:
Yes, binding is necessary because "the kernel must inform an application about certain events" via scheduler activations, and real-time threads need guaranteed responsiveness. Binding ensures the real-time thread has dedicated kernel resources (LWP) for predictable scheduling.

Key quote: "Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors." (p. 193)

4.8 Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

From p. 161:

1. "A CPU-bound application running on a single processor. In this scenario, only one thread can run at a time, so one LWP is sufficient."
2. Simple sequential programs where tasks must execute in strict order with no parallelism opportunities.

4.9 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

From p. 162:
"Multithreading an interactive application may allow a program to continue running even if part

of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user."

4.10 Which of the following components of program state are shared across threads in a multithreaded process?

From p. 160:
"b. Heap memory" and "c. Global variables" are shared (threads share data section)
"a. Register values" and "d. Stack memory" are not shared (each thread has its own)

4.11 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

From p. 167:
"Green threads—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model. However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores."

4.12 In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.

From p. 161-162:
The benefits of process isolation (crash containment, security separation) wouldn't be achieved with threads since threads share memory space. Threads are more economical but don't provide the same isolation.

4.13 Is it possible to have concurrency but not parallelism? Explain.

From p. 163:
"Yes, it is possible to have concurrency without parallelism. A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously."

4.14 Using Amdahl's Law, calculate the speedup gain for the following applications:

From p. 164 (Amdahl's Law formula):
speedup ≤ 1/(S + (1-S)/N)

4.15 Determine if the following problems exhibit task or data parallelism:

From p. 165:

- Photo thumbnails: Data parallelism (same operation on different data)
- Matrix transpose: Data parallelism
- Networked app: Task parallelism (different operations)
- Fork-join array sum: Data parallelism
- Grand Central Dispatch: Can be both

4.16 A system with two dual-core processors has four processors available for scheduling...

From p. 162 (scalability benefit):

- I/O: 1 thread (sequential)
- CPU portion: 4 threads (one per core)

4.17 Consider the following code segment...

From p. 188-189 (fork() behavior):
a. 4 processes (original + 3 forks)
b. 1 thread (in child process)

4.18 Contrast Linux and Windows approaches to modeling processes and threads.

From p. 195-196:

- Windows: Explicit thread objects with ETHREAD, KTHREAD, TEB structures
- Linux: Uses clone() with flags to determine sharing, no distinction between processes/threads

4.19 The program shown in Figure 4.23...

From p. 170-171:
LINE C: 5 (child modifies value)
LINE P: 0 (parent doesn't see child's change due to fork() copy)

4.20 Consider a multicore system and a multithreaded program...

From p. 168 (many-to-many model):
a. Underutilized cores
b. Optimal utilization
c. Some threads must wait

4.21 Pthreads provides an API for managing thread cancellation...

From p. 190-191:
Operations that shouldn't be interrupted:

1. Updating shared data structures
2. Acquiring/releasing locks

Important Lines to Highlight for Quiz

These are critical concepts that might appear on your quiz:

1. "A thread is a basic unit of CPU utilization" (p. 160)
2. "Four major benefits: Responsiveness, Resource sharing, Economy, Scalability" (p. 162)
3. "Concurrency vs Parallelism" diagram and explanation (p. 163-164)
4. "Many-to-one, one-to-one, many-to-many models" (p. 166-168)
5. "LWPs appear as virtual processors" (p. 193)
6. "Thread cancellation types: asynchronous and deferred" (p. 190)
7. "Linux uses clone() with sharing flags" (p. 195-196)
8. "Amdahl's Law shows limits of parallel speedup" (p. 164)

1. Pthreads Summation Program (Figure 4.11, p. 170)

Problem: Create a multithreaded program that calculates the sum of integers from 1 to N using Pthreads.

Solution:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum; /* shared data */
void *runner(void *param); /* thread function */

int main(int argc, char *argv[]) {
   pthread_t tid; /* thread identifier */
   pthread_attr_t attr; /* thread attributes */

   pthread_attr_init(&attr); /* default attributes */
   pthread_create(&tid, &attr, runner, argv[1]); /* create thread */
   pthread_join(tid, NULL); /* wait for thread */

   printf("sum = %d\n",sum);
}

void *runner(void *param) {
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Explanation:

1. sum is declared globally so threads can share it
2. main() initializes thread attributes with default values
3. pthread_create() spawns new thread running runner() function

4. pthread_join() makes parent wait for child thread to complete
5. runner() calculates sum of integers and stores in shared sum variable
6. pthread_exit(0) terminates the thread

2. Windows Thread Summation (Figure 4.13, p. 172)

Problem: Implement the same summation using Windows threads API.

Solution:

```
#include <windows.h>
#include <stdio.h>

DWORD Sum; /* shared data */

DWORD WINAPI Summation(LPVOID Param) {
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[]) {
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param = atoi(argv[1]);

   ThreadHandle = CreateThread(
      NULL, /* security attributes */
      0, /* stack size */
      Summation, /* thread function */
      &Param, /* parameter */
      0, /* creation flags */
      &ThreadId); /* thread id */

   WaitForSingleObject(ThreadHandle, INFINITE);
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
```

```
}
```

Explanation:

1. Sum declared as DWORD (unsigned 32-bit integer)
2. CreateThread() creates thread running Summation() function
3. WaitForSingleObject() makes parent wait for child thread
4. CloseHandle() releases thread resources
5. Summation() performs same calculation as Pthreads version

3. Java Thread Summation (Figure 4.14, p. 176)

Problem: Implement summation using Java's Executor framework.

Solution:

```
import java.util.concurrent.*;

class Summation implements Callable<Integer> {
   private int upper;

   public Summation(int upper) {
      this.upper = upper;
   }

   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;
      return sum;
   }
}

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) {}

        pool.shutdown();
    }
}
```

Explanation:

1. Summation implements Callable<Integer> for thread with return value
2. call() method contains the summation logic
3. ExecutorService manages thread pool
4. submit() queues the task and returns Future object
5. get() retrieves result when computation is complete
6. shutdown() terminates thread pool

4. Java Fork-Join Summation (Figure 4.18, p. 183)

Problem: Implement array summation using Java's fork-join framework.

Solution:

```
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();
        }
    }
}
```

Explanation:

1. Extends RecursiveTask<Integer> for recursive decomposition
2. THRESHOLD determines when to stop splitting
3. compute() either:
    - Directly computes sum if range is small
    - Splits task in half and forks subtasks
4. fork() asynchronously executes subtasks
5. join() waits for and retrieves results
6. Combines results from left and right subtasks

5. Thread Pool Example (Figure 4.15, p. 180)

Problem: Create a thread pool to execute multiple tasks.

Solution:

```
import java.util.concurrent.*;

public class ThreadPoolExample {
    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0]);
        ExecutorService pool = Executors.newCachedThreadPool();

        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task());
```

```
        pool.shutdown();
    }
}
```

Explanation:

1. Executors.newCachedThreadPool() creates expandable thread pool
2. execute() submits tasks to pool
3. Each Task (not shown) implements Runnable
4. shutdown() initiates orderly shutdown after all tasks complete

Key Patterns in All Solutions:

1. **Shared Data Management**:
   - Pthreads/Windows: Global variables
   - Java: Object fields or return values
2. **Thread Creation**:
   - Pthreads: pthread_create()
   - Windows: CreateThread()
   - Java: Executor services or direct Thread creation
3. **Synchronization**:
   - Pthreads: pthread_join()
   - Windows: WaitForSingleObject()
   - Java: Future.get() or join()
4. **Task Decomposition**:
   - Fork-join recursively splits problems
   - Thread pools handle independent tasks
5. **Resource Cleanup**:
   - Pthreads: Automatic on exit
   - Windows: CloseHandle()
   - Java: shutdown() or try-with-resources