**CHAPTER 4 (Threads)**

**Q1. In Chapter 3, we discussed Google’s Chrome browser and its practice of opening each new website in a separate process. Would the same benefits have been achieved if instead Chrome had been designed to open each new website in a separate thread? Explain.**

**Solution -**

Processes and threads can typically be defined as sequences of execution for any program. The main difference lies beneath the memory space they share RAM or main memory of the system while executing.

* Each process acquires a unique address space on the memory; whereas threads of a program usually share the same memory space for their execution.
* Processes act independently while threads always remain dependent on their consequent thread for their execution. Threads remain a chain of instructions where a small breakdown may terminate the entire execution.
* Google Chrome browser maintains to open each new website as an independent process rather than opening them as threads to ensure that no website breakdown affects others' service.
* One cannot maintain the efficiency of the browser while opening each new website as a thread, because threads are allocated with shared memory spaces. Hence they may affect each other if one of them crashes unexpectedly.

**Q2. Determine if the following problems exhibit task or data parallelism:**

**• The multithreaded statistical program described in Exercise 4.21**

**• The multithreaded Sudoku validator described in Project1 in this chapter**

**• The multithreaded sorting program described in Project 2 in this chapter**

**• The multithreaded web server described in Section 4.1**

**Solution -**

* The statistical program is Data Parallelism. Here, multiple threads are created and each thread is performing functions like calculating average, finding minimum value, finding maximum value on same data.
* So in order to perform these operations it creates threads and does the operation in parallel for task completion.
* The multithreaded Sudoku validator is Task Parallelism. Here, in Sudoku solution example, there are constraints that each row or column should contain the digits from 1 to 9. And each grid should have digits from 1 to 9, which can go when one thread completes, it starts another until all eleven threads So, here it takes task from one thread to another, until it completes and satisfies all tasks and these tasks need not to run concurrently.
* The multithreaded sorting program is Data Parallelism. Here, in sorting list, the list is divided in two half threads runs concurrently and execute individually to provide resultant threads and then
* The multithreaded web server is Task Parallelism. In single threaded, the threads which are created are able to perform only one task, whereas, multithreaded creates threads in such a way that is able to perform multitask at a time leading to better performance.

**Q3. A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between startup and termination, the program is entirely CPU bound.**

**Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).**

**• How many threads will you create to perform the input and output? Explain.**

**• How many threads will you create for the CPU-intensive portion of the application? Explain.**

**Solution -**

Threads count depends upon the priority and requirements of the application. So only thread is enough for this kind of application and this thread is going to handle both input and output operation.

* It is a concurrency approach. Here, it only makes sense to create as many threads as there are blocking system calls, as the threads will be spent blocking.
* It doesn't provides any benefits to create an additional threads.
* Thus, only a signal thread creation makes sense for input and a single thread for output.
* Four threads are created to perform the CPU-intensive portion of the application. It is because, there should be as many threads as there are processing cores.
* It would be the waste of processing resources to use fewer threads.
* Also any number greater than four would be unable to run.

**CHAPTER 5(Process Synchronization)**

**Q1. Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit (amount) and withdraw (amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring.**

**Solution -**

Assume the balance in the account is 250.00 and the husband calls withdraw (50) and the wife calls deposit (100). Obviously the correct value should be 300.00 Since these two transactions will be serialized, the local value of balance for the husband becomes 200.00, but before he can commit the transaction, the deposit (100) operation takes place and updates the shared value of balance to 300.00 We then switch back to the husband and the value of the shared balance is set to 200.00 - obviously an incorrect value.

The solution is to create a "Lock" on the Current Balance variable (or any shared memory).  By agreement, each method must "Lock" the variable before accessing it and changing it.  If a method determines that the variable is "Locked," then it must wait before accessing it.  [note: If, somehow the variable gets "locked" for a long period of time, this results in a "deadlock" condition and one process or the other must be cancelled.]

The pseudocode is easy:

  method withdrawl(Amt)

      If BalanceLock is locked, then WAIT

      Lock (BalanceLock)

          ASSIGN (Balance - Amt) to Balance

     UnLock (BalanceLock)

  end withdrawl

  method deposit(Amt)

      If BalanceLock is locked, then WAIT

      Lock (BalanceLock)

         ASSIGN (Balance + Amt) to Balance

      Unlock (BalanceLock)

  end deposit

**Q2. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, *P*0 and *P*1, share the following variables:**

**boolean flag[2]; /\* initially false \*/**

**int turn;**

**The structure of process *Pi* (i == 0 or 1) is shown in Figure 5.21. The other process is *Pj* (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.**

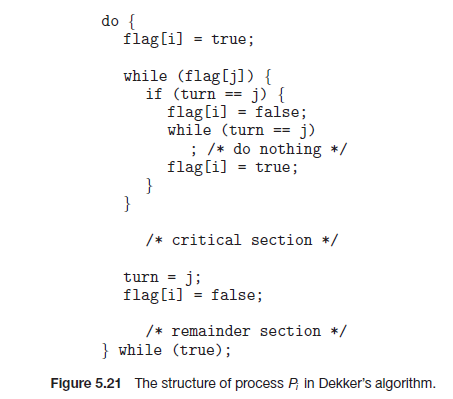
**Solution -**

This algorithm satisfies the three conditions of mutual exclusion. 

1. Mutual exclusion is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn.
2. Progress is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It sets turn to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again—before the other process—it repeats the process of entering its critical section and setting turn to the other process upon exiting.
3. Bounded waiting is preserved through the use of the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered—and exited—its critical section. However, Dekker's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

**CHAPTER 7 (Deadlocks)**

**Q1. Assume a multithreaded application uses only reader–writer locks for synchronization. Applying the four necessary conditions for deadlock, is deadlock still possible if multiple reader–writer locks are used?**

****

**Solution -**

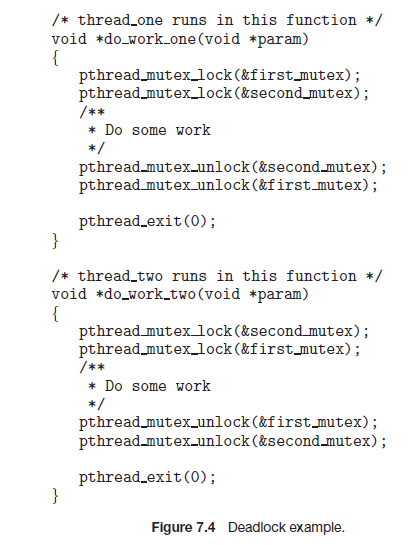
Yes.

1. Mutual exclusion is maintained, as they cannot be shared if there is a writer.

2. Hold-and-wait is possible, as a thread can hold one reader—writer lock while waiting to acquire another.

3. You cannot take lock away, so no preemption is upheld.

4. A circular wait among all threads is possible.

**Q2. The program example shown in Figure 7.4 doesn’t always lead to deadlock. Describe what role the CPU scheduler plays and how it can contribute to deadlock in this program.**

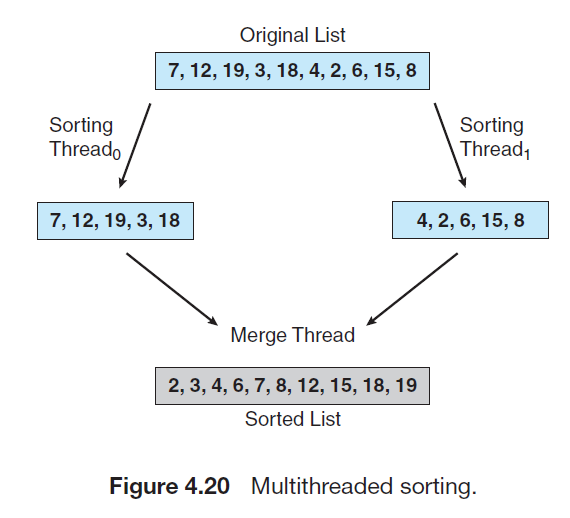
**Solution -**

If thread\_one is scheduled before thread\_two and thread\_one is able to acquire both mutex locks before thread\_two is scheduled, deadlock will not occur. Deadlock can only occur if either thread\_one or thread\_two is able to acquire only one lock before the other thread acquires the second lock.

**Project (Chapter - 4)**

**Project 2. Multithreaded Sorting Application**

**Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term *sorting threads*) sort each sub list using a sorting algorithm of your choice. The two sub lists are then merged by a third thread—a *merging thread* —which merges the two sub lists into a single sorted list.**

**Because global data are shared cross all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work on one half of this array. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sub lists into this second array. Graphically, this program is structured according to Figure 4.20. This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. Refer to the instructions in Project 1 for details on passing parameters to a thread. The parent thread will output the sorted array once all sorting threads have exited.**

**Solution -**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <iostream>

using namespace std;

#define N 2 //for storing number of threads in N

int a[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

typedef struct Arr

{

int low;

int high;

} Array;

void merge(int low, int high) //sorting algorithm and merging function

{

int mid = (low+high)/2;

int left = low;

int right = mid+1;

int b[high-low+1];

int i, temp = 0;

while(left <= mid && right <= high)

{

if (a[left] > a[right])

b[temp++] = a[right++];

else

b[temp++] = a[right++];

}

while(left <= mid)

b[temp++] = a[left++];

while(right <= high)

b[temp++] = a[left++];

for (i = 0; i < (high-low+1) ; i++)

a[low+i] = b[i];

}

void \* mergesort(void \*a) //function that creates threads and calls the merge function which sorts and merges the sub sets

{

Array \*pa = (Array \*)a;

int mid = (pa->low + pa->high)/2;

Array aIndex[N];

pthread\_t thread[N];

aIndex[0].low = pa->low;

aIndex[0].high = mid;

aIndex[1].low = mid+1;

aIndex[1].high = pa->high;

if (pa->low >= pa->high) return 0;

int i;

for(i = 0; i < N; i++)

pthread\_create(&thread[i], NULL, mergesort, &aIndex[i]); /\*creating a new thread with routine mergesort and only parameter as aIndex[i]\*/

for(i = 0; i < N; i++)

pthread\_join(thread[i], NULL); //ending the thread thread[i] with return status set to NULL

merge(pa->low, pa->high);

return 0;

}

int main()

{

Array ai; //creating element ai of struct type arr

ai.low = 0;

ai.high = sizeof(a)/sizeof(a[0])-1; //assigning last location of the array to ai.high

pthread\_t thread; //making new thread with thread label

pthread\_create(&thread, NULL, mergesort, &ai); /\*creating thread with routine mergesort and only parameter ai\*/

pthread\_join(thread,NULL); //ending the thread 'thread' and return i

for (int i = 0; i < 10; i++)

cout << "\t" << a[i];

cout << endl;

return 0;

}