Home Work - Chapter # 4

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

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.

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

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

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

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

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.

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

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

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

a. Register values
b. Heap memory
c. Global variables
d. Stack memory

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

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.

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

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

- 40 percent parallel with (a) eight processing cores and (b) sixteen processing cores
- 67 percent parallel with (a) two processing cores and (b) four processing cores
- 90 percent parallel with (a) four processing cores and (b) eight processing cores

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

- Using a separate thread to generate a thumbnail for each photo in a collection
- Transposing a matrix in parallel
- A networked application where one thread reads from the network and another writes to the network
- The fork-join array summation application
- The Grand Central Dispatch system

4.16 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 start-up 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.

4.17 Consider the following code segment:

```
pid t pid;
pid = fork();
if (pid == 0) { /* child process */
fork();
thread create( . . .);
}
fork();
```

a. How many unique processes are created?

b. How many unique threads are created?

4.18 Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the clone() system call. However, other operating systems, such as Windows, treat processes and threads differently. Typically, such systems use a notation in which the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

4.22 Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value.

For example, suppose your program is passed the integers 90 81 78 95 79 72 85

The program will report

The average value is 82

The minimum value is 72

The maximum value is 95

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. (We could expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

4.23 Write a multithreaded program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

4.19 The program shown below uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

```
#include <pthread.h>
#include <stdio.h>
int value = 0;
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
pid t pid;
pthread t tid;
pthread attr t attr;
pid = fork();
if (pid == 0) { /* child process */
pthread attr init(&attr);
pthread create(&tid,&attr,runner,NULL);
pthread join(tid,NULL);
printf("CHILD: value = %d",value); /* LINE C */
}
else if (pid > 0) { /* parent process */
wait(NULL);
printf("PARENT: value = %d",value); /* LINE P */
}
}
void *runner(void *param) {
value = 5;
pthread exit(0);
}
```