4.Multithreaded Programming

4.1 Provide three programming examples in which multithreading provides better performance than a single-threaded solution.
Answer:
(1) A Web server that services each request in a separate thread.
(2) A parallelized application such as matrix multiplication where different parts of the matrix may be worked on in parallel.
(3) An interactive GUI program such as a debugger where a thread is used to monitor user input, another thread represents the running application, and a third thread monitors performance.

4.2 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?
Answer:
When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multithreaded solution would perform better even on a single-processor system.

4.3 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
Answer:
The threads of a multithreaded process share heap memory and global variables. Each thread has its separate set of register values and a separate stack.

4.4 Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single processor system?
Answer:
A multithreaded system comprising of multiple user level threads cannot make use of the different processors in a multiprocessor system simultaneously. The operating

system sees only a single process and will not schedule the different threads of the process on separate processors. Consequently, there is no performance benefit associated with executing multiple user-level threads on a multiprocessor system.

4.5 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.
Answer:
No because the whole reason why a new process was created for each browser was because if a webpage crashed, it wouldn't crash the whole browser. If you use a thread for each webpage, then if a webpage crashes, it will bring down the whole application.

4.6 Is it possible to have concurrency but not parallelism? Explain.
Answer:
Though it is not possible to have parallelism without concurrency, it is possible to have concurrency but not parallelism.

4.7 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.

Answer: As per Amdahl's law formula for the speedup gain of an application is speedup <= 1/(S + (1 - S)/N) where, S is the portion of the application that must be performed serially and N is the no. of processing cores.
(a) For two processing cores and 60 percent parallel component, S is 40 percent that is 0.4 and N is 2.
speedup <= 1/(0.4 + (1-0.4)/2 speedup <= 1.428 Speedup gain is 1.428 times.
(b) For four processing cores and 60 percent parallel component Here, S is 40 percent that is 0.4 and N is 4
speedup <= 1/(0.4 + (1-0.4)/4) speedup <= 1.81 Speedup gain is 1.81 times.

4.8 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 Project 1 in this chapter
The multithreaded sorting program described in Project 2 in this chapter
The multithreaded web server described in Section 4.1
Answer:
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.

4.9 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.

Answer:

- 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.

4.10 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?

Answer:

- The statement pid = fork(); before the if statement creates one process. The parent process say p creates this process. Let it be p1.
- The statement fork(); in the if statement creates one process. The parent process p creates this process. Let it be p2.
- After the if statement, parent process p, process p1 and process p2 will execute fork(); creating three new processes.
    - One process is created by parent process p.
    - One process is created by process p1.
    - One process is created by process p2.

Hence, 5 unique processes (p1, p2, p3, p4, p5) will be created. If the parent process is also considered, then 6 unique processes (p, p1, p2, p3, p4, p1, p5) will be created.

- Thread creation is done in if block. Only child process p1 is executed in the if block. Therefore, process p1 will be created one thread.
- In the if block one process p2 is created using fork(). Therefore, process p2 will also create a thread.

Hence, 2 unique threads will be created.

4.11 As described in Section 4.7.2, 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.

Answer:

Linux operating systems consider both threads and processes as tasks; it cannot distinguish between them. In contrast, windows operating system threads and processes differently.
This approach has pros and cons while modeling threads and processes inside the kernel.
Pros:

- Linux consider this as similar, so codes belong to operating system can be cut down easily.
- Scheduler present in the Linux operating systems do not need special code to test threads coupled with each processes.
- It considers different threads and processes as a single task during the time of scheduling.

Cons:

- - This ability makes it harder for the Linux operating system to inflict process-wide resource limitations directly.
-
- - Extra steps are needed to recognize the each processes belong to appropriate threads and complexity in performing re levant tasks.

4.12 The program shown in Figure 4.16 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

```
include pthread.h
include stdio.h
include types.h
int value = 0; void runner(void param);
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);
  }
```

Answer:

The output is CHILD: value = 5

- The child process in the thread is forked by parent process and child process each have its own memory space.
- After forking, the parent process waits for the completion of child process.
- New thread is created for child process and the runner() function is called which set the value of the global vairlable to 5.
- Thus, after execution of this line, the value displayed will be 5.

The output of LINE P in the program:
The output is PARENT: value = 0

- After completing the child process, the value of the global variable present in parent process remains 0.
- Thus, after execution of this line, the value displayed will be 0.

4.13 Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

Answer:

a. The number of kernel threads allocated to the program is less than the number of processing cores.

b. The number of kernel threads allocated to the program is equal to the number of processing cores.

c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

4.14 Pthreads provides an API for managing thread cancellation. The pthread setcancelstate() function is used to set the cancellation state. Its prototype appears as follows: pthread setcancelstate(int state, int oldstate) The two possible values for the state are PTHREAD CANCEL ENABLE and PTHREAD CANCEL DISABLE. Using the code segment shown in Figure 4.17, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation

Figure 4.17:

```
#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 pid;
        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);
}
int oldstate;
pthread_setcanelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
/* What operations would be performed here? */
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Answer:

- Thread cancellation is a process of cancelling thread before its completion. Thread cancellation leads to termination of the executing thread in the process. To cancel a thread, below given functions are used.
    - In order to cancel thread, pthread_cancel() function is used.
    - pthread_cancel() functions depends on pthread_setcancelstate() function.
    - To cancel thread immediately PTHREAD_CANCEL_ASYNCHRONOUS type is set.
    - The default cancellation type is PTHREAD_CANCEL_DEFERED which means thread is cancelled only when it reaches its termination point.

Thread cancellation state and type determine when the thread cancellation request is placed. There are two states in thread cancellation:

- PTHREAD_CANCEL_DISABLE in all cancellation state are held pending.
- PTHREAD_CANCEL_ENABLE in which cancellations requests are acted on according to thread cancellation types.
- The default thread cancellation type is PTHREAD_CANCEL_DISABLE.
- The system test for pending cancellation requests in certain blocking functions, if cancellation function is ENABLED and its type is DEFERED. These points are known as cancellation points. pthread_testcancel() function is used to create cancellation points.