Practice Exercises

Q1: Provide three programming examples in which multithreading provides better performance than a single-threaded solution.

1- Web Server.

2- Graphical User Interface Application.

3- Data Mining.

Q2: 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.

A- 1.428

B- 1.81

Q3: Does the multithreaded web server described in Section 4.1 exhibit task or data parallelism?

It is considered a data parallelism as the new thread will perform the same task but with different data.

Q4: What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads. On systems using either many-to-one or many-to-many model mapping, user threads are scheduled by the thread library, and the kernel schedules kernel threads. Kernel threads need not be associated with a process, whereas every user thread belongs to a process. Kernel threads are generally more expensive to maintain than user threads, as they must be represented with a kernel data structure.

Q5: Describe the actions taken by a kernel to context-switch between kernel level threads.

On context-switching between threads, the kernel is doing save/restore actions of the CPU registers associated with each thread.

Q6: What resources are used when a thread is created? How do they differ from those used when a process is created?

Process creation is an expensive action to do. Due to the size of the PCB data structure. On the other hand, thread creation is consuming little resources as it's data structure is smaller. You just need a small amount for stack,registers values, and scheduling information.

Q7: 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.

Because timing is crucial for real-time systems, it is important to bind a real-time thread to an LWP in order to skip the waiting time of the thread to be attached to an LWP in order to run. Consider a situation in which a real-time thread is running (is attached to an LWP) and then proceeds to block (must perform I/O, has been preempted by a higher-priority real-time thread, is waiting for a mutual exclusion lock, etc.). While the real-time thread is blocked, the LWP it was attached to is assigned to another thread. When the real-time thread has been scheduled to run again, it must first wait to be attached to an LWP. By binding an LWP to a real-time thread, you are ensuring that the thread will be able to run with minimal delay once it is scheduled.

Exercises

Q8: Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

1- A program that is sequential naturally like a program calculating tax return.

1- A program that monitors the working environment like open files. Like unix shells.

Q9: Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

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 concurrent solution would perform better even on a single-processor system.

Q10: 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

Q11: Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

No. The operating system sees only a single process and will not schedule the different threads of the process on separate processors.

Q12: 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.

We 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.

Q13: Is it possible to have concurrency but not parallelism? Explain.

Yes. concurrency means to slice the time between the processes or thread being executed. Parallelism means to make both threads or processes to run at the same time depending on the number of system cores. So, an OS may have one of them implemented or both.

Q14: 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

1- a) 1.53 b) 1.6

2- a) 1.50 b) 2.01

3- a) 3.07 b) 4.70

Q15: 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 described in Section 4.5.2

• The Grand Central Dispatch system

1- Task parallelism.

2- Task parallelism.
3- Data parallelism.
4- Data parallelism.
5- Task parallelism.

Q16: 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.

Since the model used is one to one, then using multithreading with this program will allow us to exploit the multiprocessing. We have overall 4 cores in the systems. So, we could create 4 threads maximum for the CPU-intensive portion. But, for the I/O portion we need only one thread as we just read and write from one file and not at the same time.

Q17: 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?

a- 6 Processes.
b- 8 total threads, 2 by thread_create() and 6 corresponding to the six single-threaded processes.

Q18: 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.

Linux considers this as similar, so codes belonging to the operating system can be cut down easily. Schedulers present in the Linux operating systems do not need special code to test threads coupled with each process.

It considers different threads and processes as a single task during the time of scheduling.

Q19: The program shown in Figure 4.23 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);
}
```

Line P: 0
Line C: 5

Q20: 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.

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.

a- The scheduler can only schedule user level processes to the kernel threads, and since some of the processes are not mapped to the kernel threads, they will be idle.
b- All processing cores could be utilized simultaneously.
c- A blocked kernel thread could be replaced by another one.1

Q21: 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 4.24, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation.

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

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 depend 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 states 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.