Chapter 4

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

User-level Thread：

用户级线程由用户实现，内核不知道这些线程的存在。它将它们当作单线程进程来处理。用户级线程很小，而且比内核级线程快得多。它们由程序计数器(PC)、堆栈、寄存器和一个小的进程控制块表示。此外，用户级线程的同步与内核无关。

Kernel-Level Threads：

内核级线程由操作系统直接处理，线程管理由内核完成。线程以及线程的上下文信息都由内核管理。正因为如此，内核级线程比用户级线程慢。

用户线程和内核线程的不同是：

1） 站在操作系统的角度，用户线程是不被操作系统知道的，它由用户进程创建与管理；而内核线程由操作系统的调度算法调度，因此是被操作系统所得知的。

2） 针对多对一或多对多的线程模型，用户线程可以直接由线程库调度，而内核线程则需要由内核调度，也就是上文所说的调度算法。 对于用户线程和内核线程在不同的状况下的优劣比较：

内核线程比用户线程好的情况：

① 针对内核是单线程的情况，内核线程优于用户线程。因为任何执行阻塞系统调用的用户线程都会导致整个进程阻塞，即使应用程序中可以运行其他线程。例如，有两个进程a和b。进程a有2个内核线程，而进程b有2个用户线程。如果a中的一个线程被阻塞，它的第二个线程不会受到影响。但是对于b，如果一个线程被阻塞(比如I/O)，整个进程b和第二个线程就会被阻塞。

② 针对多处理器环境，内核线程优于用户线程，因为内核线程可以在不同的处理器上同时运行，而即使有多个处理器可用，进程的用户线程也只能在一个处理器上运行。

用户线程比内核线程好的情况：

对于时间共享型的内核，用户线程优于内核线程，因为共享系统上下文切换经常发生。内核线程之间的上下文切换具有很高的开销，几乎与进程相同，而与内核线程相比，用户线程之间的上下文切换几乎没有开销。

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

会使用到以下部分：

代码段
数据部分
其他操作系统资源，

不同之处在于：

创建线程时需要分配一个数据结构TCB（thread control block）来保存寄存器集、堆栈、状态和优先级，统一进程的线程之间共享内存，而创建一个进程时需要分配一个数据结构PCB（process control block）包含内存映射、打开文件列表和环境变量。线程是进程的一部分，一个进程中可以有多个线程，因此创建进程意味着需要更大的开销。

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

当内核线程出现页面故障时，可以切换另一个内核线程，以有用的方式使用交错时间。另一方面，当发生页面故障时，单线程进程将无法执行有用的工作。因此，在程序可能经常出现页面故障或不得不等待其他系统事件的情况下，即使在单处理器系统上，多线程解决方案也会表现得更好。

4.8 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

b. Heap memory和c. Global variables，创建线程是会分配一个TCB保存程序计数器、栈、寄存器组和线程ID。

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

不能，因为用户级多线程由线程池管理，操作系统只能看到一个进程，因此只能使用到一个内核线程，只会用到一个处理器，多处理器的优势无法体现。

4.14 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.

1. 1个输入线程、1个输出线程，因为输入输出只需要一个线程进行处理，创建更多的线程没有意义并且会造成开销
2. 4个线程，因为处理器总共只有4个核，更多的线程也无法同时运行

4.15 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?

1. 父进程创建两个子进程，第一个子进程创建三个子进程，共有**6**个进程
2. 第一个子进程创建一个线程，第一个子进程的一个子进程创建一个线程，共有**2**个线程

4.16 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操作系统将线程和进程视为任务，不进行区分，相比之下，Windows操作系统的线程和流程不同。

优点:

1. 简化操作系统代码
2. Linux操作系统中的调度器不需要特殊代码来检查与进程相关的线程。
3. 在调度期间，它可以平等的看待进程和线程

缺点:

1. 这种一致性可能会使以直接方式施加进程范围的资源约束变得更加困难
2. 识别哪些线程对应于哪个进程并执行相关的记账任务需要一些额外的复杂性

4.17 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>
```

```c
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);
}
```

输出为：

```
CHILD: value = 5PARENT: value = 0
```

因为进程之间不共享全局变量，同一进程的线程之间共享全局变量

4.21 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 obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

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

int average = 0;
int min = 65535;
int max = 0;
int array[100];
int n;

void *find_min(){
    int i;
    for(i = 0; i < n; i++){
        if(array[i] < min){
            min = array[i];
        }
    }
    pthread_exit(NULL);
}

void *find_max(){
    int i;
    for(i = 0; i < n; i++){
        if(array[i] > max){
            max = array[i];
        }
    }
    pthread_exit(NULL);
}

void *find_average(){
    int i;
    for(i = 0; i < n; i++){
        average += array[i];
    }
    average = average / n;
    pthread_exit(NULL);
}

int main(){
    printf("Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("Enter the elements of the array: ");
    for(int i = 0; i < n; i++){
        scanf("%d", &array[i]);
    }
    pthread_t thread1, thread2, thread3;
    pthread_create(&thread1, NULL, find_min, NULL);
    pthread_create(&thread2, NULL, find_max, NULL);
```

```
    pthread_create(&thread3, NULL, find_average, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_join(thread3, NULL);

    printf("The average value is: %d\n", average);
    printf("The minimum value is: %d\n", min);
    printf("The maximum value is: %d\n", max);

}
```