Homework Assignment #4

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

Ans 1:

User Level Threads

User-level threads are supported at the user level, above the kernel and are managed without kernel support. Because they are implemented by users the kernel is not aware of their existence. Generally they are smaller, easier and faster to create,and they can be managed more easily compared to their counterparts and there in their synchronization there is no kernel involvement. However, if one user level thread runs a blocking operation the whole process will be blocked which is a disadvantage.

Kernel Level Threads

Kernel threads are supported and managed directly by the operating system. Since they are handled by the operating system the kernel does the thread management If a thread performs blocking operation another thread of the same process can be scheduled on another processor however these threads are slower to create then user level threads.

From the above we can find two differences between user-level threads and kernel-level threads:

• Kernel level threads are supported directly by the kernel while the user-level threads are unknown to the kernel.
• Kernel threads are slower to be created and more expensive to maintain then their counterparts because they require more resources
More difference are presented below:

Sr.No	User level thread	Kernel level thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to createand manage.

Sr.No	User level thread	Kernel level thread
2	Implementation is by a thread library at the user level.	Operating system supports creation ofKernel threads.
3	User-level thread is generic and canrun on any operating system.	Kernel-level thread is specific to theoperating system.
4	Multi-threaded applications cannottake advantage of multiprocessing	Kernel routines themselves can be multithreaded.

Answer to the second part of the Question

Generally a kernel thread is more appropriate for higher-priority tasks which need system resources while the user threads are more appropriate for lower-level tasks which do not need to be necessarily managed by the kernel.

- User-level threads are much faster to switch between, as there is no context switch; further, a problem-domain-dependent algorithm can be used to schedule among them. CPU-bound tasks with interdependent computations, or a task that will switch among threads often, might best be handled by user-level threads.
- Kernel-level threads are scheduled by the OS, and each thread can be granted its own timeslices by the scheduling algorithm. The kernel scheduler can thus make intelligent decisions among threads, and avoid scheduling processes which consist of entirely idle threads (or I/O bound threads). A task that has multiple threads that are I/O bound, or that has many threads (and thus will benefit from the additional timeslices that kernel threads will receive) might best be handled by kernel threads.

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

Ans 2:

Thread

When a thread is created, register set, stacks, and private storage area are created which are the context of the thread.

1. A register set is the storage location for storage during context switching,
2. A local stack to record the procedure call arguments, return values, and return addresses, and
3. Thread-private storage is used by various run-time libraries and dynamic link libraries (DLLs).

When a thread is created, the threads does not require any new resources to execute the thread shares the resources like memory of the process to which they belong to. The benefit of code sharing is that it allows an application to have several different threads of activity all within the same address space.

Process

When a process is created, memory is allocated for program instructions and data, as well as thread storage.

Whereas a new process creation is very heavyweight because it always requires new address space to be created and even if they share the memory then the inter process communication is expensive when compared to the communication between the threads.

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

Ans 3:

b. Heap memory c. Global variables

Reason:

A heap memory and global variables are shared among the threads of a multi-threaded process.

Every thread has its separate set of register values and a separate stack.

4. Consider the following code segment:

```
Pid_t pid;
pid = fork();
if (pid == 0) { /* child process */
fork();
pthread_create( . . .);
}
fork();
```

a. How many unique processes are created?
b. How many unique threads are created?

Ans 4:

- 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.

5. 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>
#include <types.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);
}
```

Ans 5:

LINE C ,CHILD: value = 5

Line P, PARENT: value = 0

</> source code

```c
#include <sys/types.h>
#include <unistd.h>
int value = 0;
void *runner(void *param); /* the thread */
int main(int argc, char * argv[])
{
int 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\n",value);
}
else if (pid > 0) { /* parent provess */
wait(NULL);
printf("PARENT:value =%d\n",value);
}
}
void *runner(void *param) {
value = 5;
pthread_exit(0);
}
```

input Output

Success #stdin #stdout 0s 5520KB

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

Success #stdin #stdout 0s 5520KB

6. The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as:

$$f\,ib0 = 0$$
$$f\,ib1 = 1$$
$$f\,ibn = f\,ibn\text{-}1 + f\,ibn\text{-}2$$

Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence

in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, the parent thread will have to wait for the child thread to finish.

Ans 6:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
  int a=0, b=1, n, i, user_input;
  pid_t pid;

  printf("Enter the number of a Fibonacci Sequence:\n");
  scanf("%d", &user_input);

  if (user_input < 0)
    printf("Please enter a non-negative integer!\n");
  else
  {
    pid = fork();

    if(pid < 0){
      printf("fork failed");
    }
    else if (pid == 0)
    {
      printf("Child is producing the Fibonacci Sequence...\n");
```

```
        printf("%d %d ",a,b);
        for (i=0;i<user_input;i++)
        {
          n=a+b;
          printf("%d ", n);
          a=b;
          b=n;
        }
        printf("Child complete\n");
      }
      else
      {
        printf("Parent is waiting for child to complete...\n");
        wait(NULL);
        printf("Parent ends\n");
      }
    }
    return 0;
}
```

```
Enter the number of a Fibonacci Sequence:
10946
Parent is waiting for child to complete...
Child is producing the Fibonacci Sequence...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 83204
0 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134
903170 1836311903 -1323752223 512559680 -811192543 -298632863 -1109825406 -1408458269 1776683621 368225352 2144908973 -1781832971 36
3076002 -1418756969 -1055680967 1820529360 764848393 -1709589543 -944741150 1640636603 695895453 -1958435240 -1262539787 1073992269
-188547518 885444751 696897233 1582341984 -2015728079 -433386095 1845853122 1412467027 -1036647147 375819880 -660827267 -285007387 -
945834654 -1230842041 2118290601 887448560 -1289228135 -401779575 -1691007710 -2092787285 511172301 -1581614984 -1070442683 16429096
29 572466946 -2079590721 -1507123775 708252800 -798870975 -90618175 -889489150 -980107325 -1869596475 1445263496 -424332979 10209305
17 596597538 1617528055 -2080841703 -463313648 1750811945 1287498297 -1256657054 30841243 -1225815811 -1194974568 1874176917 6792023
49 -1741588030 -1062385681 1490993585 428607904 1919601489 -1946757903 -27156414 -1973914317 -2001070731 319982248 -1681088483 -1361
106235 1252772578 -108333657 1144438921 1036105264 -2114423111 -1078317847 1102226338 23908491 1126134829 1150043320 -2018789147 -86
8745827 1407432322 538686495 1946118817 -1810161984 135956833 -1674205151 -1538248318 1082513827 -455734491 626779336 171044845 7978
24181 968869026 1766693207 -1559405063 207288144 -1352116919 -1144828775 1798021602 653192827 -1843752867 -1190560040 1260654389 700
94349 1330748738 1400843087 -1563375471 -162532384 -1725907855 -1888440239 680619202 -1207821037 -527201835 -1735022872 2032742589 2
97719717 -1964504990 -1666785273 663677033 -1003108240 -339431207 -1342539447 -1681970654 1270457195 -411513459 858943736 447430277
```