Experiment - 2
Demonstration of Thread related APIs

Aim: To identify and demonstrate variuos thread related system calls

Description: A thread is lightweight process. Several threads may run siultaneously within an application. Threads may share code, data and files along with other threads. Every thread will have its own stack, registers and PC. Each thread is identified by a thread ID. There are several benefits with threads as shown below:

Responsiveness: allows several processes to run simultaneously.
Resource sharing: mutliple threads share resources of process easier than shared memory or message queues.
Economy: cheaper than process creation, thread switching lower the overhead than context switching
Scalability: process can take advantage of multicore architectures

Threads are two types namely user level andkernel level threads. User-level threads are managed by the thread library. Kernel-level threads are supported by the kernel. Some of the operating systems that they support threads: Linux, Mac OS X, iOS, Android

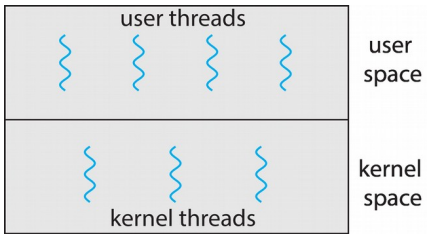

***Figure 1**: Thread types*

Thread models: There are several thread models as shown below:

- many-to-may
- one-to-one
- many-to-many

many-to-one: In this model, all the user level threads are mapped to a single kernel thread as shown in Figure 2.a. In this model, blocking of a single thread may cause all thread to block. Multiple threads may not run parallel on ulticore systems because only one kernel thread at a time. Few systems use this model like: *Solaris green threads, GNU portable threads.*

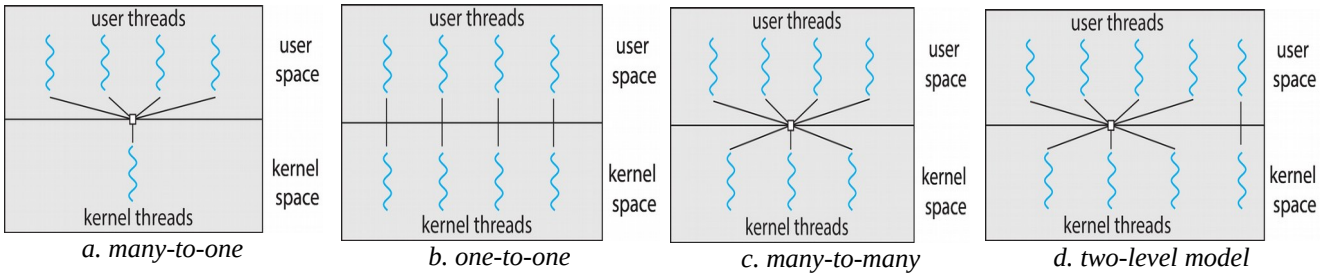

***Figure 2:** Various thread models*

one-one-one: each user-level thread is maps to kernel level thread. For each user level thread creation, corresponding kernel thread is created. Here, there will be more concurrency than many-to-one model. Ex: Windows, Linux.

Many-to-many: allows many-user level threads to many kernel threads. OS will create sufficient number of kernel threads. Windows supports this type through ThredFiber package.

Thread library: Provides APIs for the programmers to develop applications. These libraries may be user level which are available as libraries or kernel level library that are supported by OS.

Pthread: is a POSIX standar (IEEE 1003.1c) API for creation and synchronization. These are common in UNIX operating systems (Linux & Mac OS X) . Linux supports pthreads. Some of the pthread APIs are:
pthread_create(), pthread_cancel(), pthread_attach(), pthread_detach(), pthread_equal(), pthread_exit(), pthread_join(), pthread_kill(), pthread_self(), pthread_sigmask(), pthread_spin_init(), pthread_spin_lock(), pthread_spin_unlock(), pthread_spin_destroy(), pthread_spin_tryock(), pthread_attr_init(), pthread_attr_destroy(), pthread_attr_getscope(), pthread_attr_getschedpolicy() pthread_attr_getstack(), pthread_attr_getschedparam(), pthread_attr_getstacksize(), etc.

pthread_attr_init(), pthread_attr_destroy() - initialize and destroy thread attributes object
Syntax:

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

pthread_create() - statrs a new thread in the calling process. The new thread starts execution by invoking 'start_routine()' arg is passed as the sole argument of start_routine(). The ew thread terminates in one of the following ways:

1. it calls pthread_exit(), specifying an exit status value that is available to another thread in the same process that calls pthread_join().
2. It returns from 'start_routine()'. This is equivalent to calling pthread_exit() with the value supplied in the return statement.
3. It is cancelled
4. any of the threads in the process calls exit(), or the main() thread performs a return from main(). This causes the termination of all threads in the process.

The '*attr*' argument points to a pthread_attr_t structure whose contents are used at thread creation time to determine attributes for the new thread; this is initialized using pthread_attr_init() and related functions. If attr is NULL, then the thread is created with default attribtes. Before returning, a successful call to pthread_create() stores the ID of the new thread in the buffer pointed by 'thread' argument. This identifier is used to refer to the thread in subsequent calls to other pthreads functions.The new thread inherits a copy of the creating thread's signal mask. On sucess it reurns 0; on faillure, it returns an error number, and the contents of *thread are undefined.

Syntax:

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

Architecture	Default stack size
i386	2 MB
IA - 64	32 MB
Power PC	4 MB
S/390	2 MB
Sparc-32	2 MB
sparc-64	4 MB
x86-64	2 MB

Task-1: use the 'man page of pthread_create() and pthread_attr_init() APIs'. Test the sample code given at the end of those two manuals and observe the output. Then write your inferences.

Example-1: Program to demonstrate the 'pthread_create()' API.

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
int sum;      //this data is shared by the threads
void *runner(void *param);

int main(int argc, char *argv[])
{
        pthread_t tid;                                  //thread identifier
        pthread_attr_t attr;                            //set of thread attributes
        pthread_attr_init(&attr);                       //set the default attributes of the thread
        printf("\nSum value before thread execution: %d\n",sum);
        pthread_create(&tid, &attr, runner, argv[1]);
        pthread_join(tid, NULL);        //suspends the parent until the thread is terminated

        printf("Sum=%d:\n\n",sum);
        return 0;
}

void *runner(void *parm)                //thread will execute in this function to compute the sum
{
        int i,upper;
        printf("Thread execution started....\n");
        upper=atoi(parm);
        sum=0;
        for(i=1;i<=upper; ++i)
                sum+=i;         /sum computation which will update the global variable 'sum'
        pthread_exit(0);
}
```

to compile include the option **-pthread**.

Task-2: Using the above program structure, write a program that creates a thread which will compute the factorial of a number passed as a command line argument, and the parent displays the result.

Example-2: Program to shared data

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
int g=0;      //global variable to change it in threads
void *func(void *);

int main(int argc, char *argv[])
{
        pthread_t tid;    //thread identifier
        int i;
        for(i=0;i<5;++i)
                pthread_create(&tid, NULL, func, (void*)i);
        printf("\nin main g value=%d\n",g);
        pthread_exit(NULL);
        return 0;
}
```

```
void *func(void *parm)     //thread will execute in this function
{
        int th;
        th=(int*)parm;  // store the argument
        static int s=0;
        ++s; ++g;
        printf("Thread No: %d, Static value: %d Global value: %d\n", th,++s,++g);
}
```

Example-3: Program to demonstrate multi threading and also to determine the thread attributes

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *runner();
int main(int argc, char *argv[]) {
  int i, scope;       pthread_t tid[NUM_THREADS];
  pthread_attr_t attr;
  pthread_attr_init(&attr);                                  ///* get the default attributes */

  if (pthread_attr_getscope(&attr, &scope) != 0)             /* first inquire on the current scope */
       fprintf(stderr, "Unable to get scheduling scope\n");
  else {
    if (scope == PTHREAD_SCOPE_PROCESS)
                printf("PTHREAD_SCOPE_PROCESS\n");
    else if (scope == PTHREAD_SCOPE_SYSTEM)
                printf("PTHREAD_SCOPE_SYSTEM\n");
             else
                fprintf(stderr, "Illegal scope value.\n");
  }

  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);  //set the scheduling alg. To PCS or SCS

  for (i = 0; i < NUM_THREADS; i++)                          //create threads
    pthread_create(&tid[i],&attr,routine,NULL);

  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);                              //wait for the threads to terminate
}

void *routine(void *param)                                   /* Each thread will begin control in this function */
{
  /* do some work ... */
        static int s=0;
        s++;
  printf("Thread %d is running \n",s);
  pthread_exit(0);
}
```

Task-3: Modify the above prgram that creates three threads where the 1st thread computes the sum, 2nd thread computes the factorial and 3rd thread checks wheter the given number is a palindrome or not. The argument should be passed as a command line argument and the results are to be displayed by the parent after the respective threads are terminated.

References:

1. ‘man’ pages of varius process related system calls
2. https://www.usna.edu/Users/cs/aviv/classes/ic221/s17/units/04/unit.html