University of Central Punjab

FOIT (Operating Systems)

Threads

Objective

- Students will be able to create multiple threads in a program.
- Students will be able to get and set various attributes of a thread.
- Students will be able to pass parameters to a thread and return a value from it.
- Students will be able to cancel the threads instantly or at a safe point.
- Students will be able to break a large problem among various threads to achieve speedup.

What is a Thread?

A thread is a flow of control within a process. A process can contain multiple threads.

Why Multithreading?

A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. More advantages of multithreading are discussed below **Process vs Thread?**

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one another like processes are, and as a result thread share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

Advantages of Thread over Process

1. **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.

2. **Faster context switch:** Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.

3. **Effective utilization of multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.

4. **Resource sharing:** Resources like code, data, and files can be shared among all threads within a process.
Note: stack and registers can’t be shared among the threads. Each thread has its own stack and registers.

5. **Communication:** Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two processes.

6. **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered

as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

```
#include <pthread.h>
  int pthread_create(pthread_t *thread, const pthread_attr_t
      *attr, void *(*start_routine) (void *), void *arg);
```

Compile and link with *-pthread*.

The **pthread_create**() function starts 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 *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 structure is initialized using pthread_attr_init(3) and related functions. If *attr* is NULL, then the thread is created with default attributes.

Before returning, a successful call to **pthread_create**() stores the ID of the new thread in the buffer pointed to by *thread*; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

```
#include <pthread.h>
      int pthread_join(pthread_t thread, void **retval);
```

Compile and link with *-pthread*.

The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread supplied to pthread_exit(3)) into the location pointed to by *retval*. If the target thread was canceled, then **PTHREAD_CANCELED** is placed in the location pointed to by *retval*.

On success, **pthread_join**() returns 0; on error, it returns an error number.

Example: Pthread Creation and Termination
This simple example code creates 5 threads with the pthread_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit().

Creating Hello World Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}
int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

Example: Parameter passing and returning value from a thread. In this example, we will see how we can pass a structure and an array to two different threads and returning value from them.

Passing parameters and returning values from Threads

```
#include<stdio.h>
#include<pthread.h>
struct two_numbers
{
    int a;
    int b;
};
void* f1(void* num)
{
    int* n = (int*)num;
    int sum = n[0] + n[1];
    return (void*)((long)sum);
}
void* f2(void* num)
{
    struct two_numbers *n = (struct two_numbers*)num;
    return (void*)((long)n->a+n->b);
}
int main()
{
    pthread_t pid1,pid2;
    void* stat1,*stat2;
    int numbers[2] = {5,4};
    struct two_numbers two_num;
    two_num.a = 1;
    two_num.b = 2;
    void* ptr = (void*)&two_num;
    pthread_create(&pid1,NULL,&f1,(void*)numbers);
    pthread_create(&pid2,NULL,&f2,ptr);
    pthread_join(pid1,&stat1);
    pthread_join(pid2,&stat2);
    if(stat1!=NULL && stat2 != NULL)
        printf("Sum are %ld and %ld \n",(long)stat1,(long)stat2);
    return 0;
}
```

```
#include <pthread.h>
  int pthread_cancel(pthread_t thread);
```

Compile and link with - *pthread*.

The **pthread_cancel**() function sends a cancellation request to the thread *thread*. Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability *state* and *type*.

A thread's cancelability state, determined by pthread setcancelstate(3), can be *enabled* (the default for new threads) or *disabled*. If a thread has disabled cancellation, then a cancellation request remains queued until the thread enables cancellation. If a thread has enabled cancellation, then its cancelability type determines when cancellation occurs.

A thread's cancellation type, determined by pthread setcanceltype(3), may be either *asynchronous* or *deferred* (the default for new threads). Asynchronous cancelability means that the thread can be canceled at any time (usually immediately, but the system does not guarantee this). Deferred cancelability means that cancellation will be delayed until the thread next calls a function ~~pthread_testcancel()~~ at *cancellation point*.

```
#include <pthread.h>

 int pthread_setcancelstate(int state, int *oldstate);
  int pthread_setcanceltype(int type, int *oldtype);
```

Compile and link with *-pthread*.

The pthread_setcancelstate() sets the cancelability state of the calling thread to the value given in the state. The previous cancelability state of the thread is returned in the buffer pointed to by the oldstate. The state argument must have one of the following values:

PTHREAD_CANCEL_ENABLE
The thread is cancelable. This is the default cancelability state in all new threads, including the initial thread. The thread's cancelability type determines when a cancellable thread will respond to a cancellation request.
PTHREAD_CANCEL_DISABLE

The thread is not cancelable. If a cancellation request is received, it is blocked until cancelability is enabled.

The **pthread_setcanceltype**() sets the cancelability type of the calling thread to the value given in *type*. The previous cancelability type of the thread is returned in the buffer pointed to by *oldtype*. The *type* argument must have one of the following values:

PTHREAD_CANCEL_DEFERRED
A cancellation request is deferred until the thread next calls a function that is the cancellation point. This is the default cancelability type in all new threads, including the initial thread. Even with deferred cancellation, a cancellation point in an asynchronous signal handler may still be acted upon and the effect is as if it was an asynchronous cancellation.
PTHREAD_CANCEL_ASYNCHRONOUS
The thread can be canceled at any time. (Typically, it will be canceled immediately upon receiving a cancellation request, but the system doesn't guarantee this.)

The set-and-get operation performed by each of these functions is atomic with respect to other threads in the process calling the same function.

On success, these functions return 0; on error, they return a nonzero error number.

Example: In the given example, we will create two threads. One thread will change its cancel type to Asynchronous and other to Deffered. The one with asynchronous type is cancelled instantly while the other requires pthread_testcancel() to be called for its cancellation.

Thread cancellation & changing cancellation type

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<pthread.h>
void* f1()
{
    pthread_setcanceltype( PTHREAD_CANCEL_ASYNCHRONOUS,NULL);
    while(1)
        printf("\nThis is thread 1");
}
void* f2()
{
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,    NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL);
    while(1)
    {
        for(int i=0;i<2;i++)
        {
            printf("\nThis is thread 2");
        }
        pthread_testcancel();
    }
}
int main()
{
    pthread_t pid1,pid2;
    pthread_create(&pid1,NULL,&f1,(void*)NULL);
    pthread_create(&pid2,NULL,&f2,(void*)NULL);
    sleep(5);
    pthread_cancel(pid1);
    sleep(5);
    pthread_cancel(pid2);
    pthread_join(pid1,NULL);
    pthread_join(pid2,NULL);
    return 0;
}
```

```
#include <pthread.h>
    int pthread_attr_init(pthread_attr_t *attr);
   int pthread_attr_destroy(pthread_attr_t *attr);
```

Compile and link with *-pthread*.

The **pthread_attr_init**() function initializes the thread attributes object pointed to by *attr* with default attribute values. After this call, individual attributes of the object can be set using various related functions, and then the object can be used in one or more pthread create(3) calls that create threads.
Calling **pthread_attr_init**() on a thread attributes object that has already been initialized results in undefined behavior.

When a thread attributes object is no longer required, it should be destroyed using the **read_attr_destroy**() function. Destroying a thread attributes object has no effect on threads that were created using that object.

Once a thread attributes object has been destroyed, it can be reinitialized using **pthread_attr_init**(). Any other use of a destroyed thread attributes object has undefined results.

On success, these functions return 0; on error, they return a nonzero error number.

```
#include <pthread.h>

 int pthread_attr_setstack(pthread_attr_t *attr, void
            *stackaddr, size_t stacksize);
   int pthread_attr_getstack(const pthread_attr_t *attr, void
                 **stackaddr, size_t *stacksize);
```

Compile and link with *-pthread*.

The pthread_attr_setstack() function sets the stack address and stack size attributes of the thread attributes object referred to by *attr* to the values specified in *stackaddr* and *stacksize*, respectively. These attributes specify the location and size of the stack that should be used by a thread that is created using the thread attributes object *attr*. *stackaddr* should point to the lowest addressable byte of a buffer of *stacksize* bytes that was allocated by the caller. The pages of the allocated buffer should be both readable and writable.

The **pthread_attr_getstack**() function returns the stack address and stack size attributes of the thread attributes object referred to by *attr* in the buffers pointed to by *stackaddr* and *stacksize*,respectively.

On success, these functions return 0; on error, they return a nonzero error number.

Example: In this example, we will initialize a variable with default attributes. We will get the stack size from attribute variable and display it. We will then set the size of stack to 16 MB.

Getting & setting stack size in pthread attributes

```
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
void* f1()
{
    printf("\nThis is thread 1\n");
}
int main()
{
    pthread_t pid1;
    size_t stacksize;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_getstacksize(&attr, &stacksize);
    printf("\nCurrent size of stack is %d",(int)stacksize);
    pthread_attr_setstacksize(&attr,1024*1024*16); /*setting stack to 16 MB*/
    pthread_create(&pid1,&attr,&f1,(void*)NULL);
    pthread_join(pid1,NULL);
    pthread_attr_destroy(&attr);
    return 0;
}
```

University of Central Punjab

FOIT (Operating Systems)

Threads

Task 1 (to be submitted in the class)

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 two separate worker threads. One thread will determine the even numbers and the second will determine the odd numbers.

Example:

Input:

./a.out 90 81 78 95 79 72 85

Output:

The even numbers are: 90 78 72

The odd numbers are: 81 95 79 85

Task 2 (to be submitted in the class)

Write a C program using multi-threading (3) to compute C *(n,p)* given by

$$C(n,p) = \frac{n!}{(n-p)! * p!}$$

Each of the three threads computes and write the value of n!, p! and (n-p)! and the main computes $C(n,p)$ function by using the data of the threads and displays the result.

Task 3 (Home task to be submitted before 11:55PM)

Write a program that is passed a filename, a number N and a string S through command-line argument. The program opens a file to search for a string S using N number of threads. If any of the threads find sting S, it prints an appropriate message along with line and column number and exits. The other threads will also exit immediately once the string is found by any thread.