---- PART - 1 ---- SUMMARY OF PTHREAD APIs ----

1)

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

Creates a new thread. Attribues are specified by attr argument. If attr is null then default attributes are used. Stores the ID in the first argument and runs the function passed as start routine with argument arg.

2)

```
pthread_join(pthread_t thread, void **retval);
```

PTHREAD_JOIN: waits for the thread specified by 1st argument to exit (if that thread is not detached) and stores the result in the second argument. In case multiple threads are waiting for the same thread only one of them will get the return value

3)

```
pthread_exit (void *status);
```

PTHREAD_EXIT: The thread exits and the pointer to the result is made avialable to any fuction that calls join on it if it is not detached.

4)

```
pthread_cancel (pthread_t thread);
```

PTHREAD_CANCEL: Cancels the execution of the thread specified by it.

5)

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

PTHREAD_ATTR_INIT: Initializes the attribute object passed to default values.

6)

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

PTHREAD_ATTR_DESTROY: Initializes the attribute object to invalid values.(implementation dependent)

7)

```
pthread_setschedparam (pthread_t thread, int policy,
                  const struct sched_param *param);
```

PTHREAD_SETSCHEDPARAM: The thread passed is assigned the schedule params in param. The scheduling policy in policy is assigned to scheduler.

---- PART - 2 ---- OUR IMPLEMENTAION OF PTHREAD APIs ----

-------- DATA STRUCTURES --------

Following are the data structures and functions were defined by us for our implementation:

1) Datatypes defined in userprog/sys.h:

```
    Defining the detachstate and schedparam
      typedef enum {DETACHED, JOINED} ;
      typedef enum {SCHED_FCFS, SCHED_RR, SCHED_PRIORITY} ;

  Typedef for pthread_t
    typedef int pthread_t;

  Struct for attributes object
    typedef struct attri {
      int detachstate;
      int inheritsched;
      int schedpolicy;
```

```
       int sched_priority;
     } pthread_attr_t;

   /*pthread errors defined:
     1 = max threads already running,
     2 = invalid attr,
     3 = invalid pthread_id)
   */
   #define EAGAIN 1
   #define EINVAL 2
   #define ESRCH  3

   //destroy value for attr object
   #define ATTR_DESTROY 100
```

2) Datatypes Defined in userprog/syscall.c

```
     struct thread_info   // defined in detail below.
     list pthread_list    // a list of above data structures
     struct lock listuse  // to ensure synchronization in above list
     int thread_count     // counts the no of threads
     struct lock threadcount  // to ensure synchronization of thread_count
```

3) Added datatype to threads/thread.c:

```
    //stores the current scheduling policy
     int sched_policy = SCHED_RR; //default is round robin
```

New functions defined in threads/thread.c:

```
    //to remove a thread from the ready queue (required in pthread_cancel)
     void thread_cancel(pthread_t n, enum intr_level old_level);

    //to set the priority of a thread (required in pthread_setschedparam)
     void thread_set_priority_now(pthread_t n , int priority);

    //to set required policy in the global variable sched_policy
     void set_sched_policy(int n)
```

Modified the following functions in threads/thread.c:

```
    //to incorporate for priority scheduling and fcfs scheduling
     static struct thread *next_thread_to_run (void);

    //to prevent preemption for priority scheduling and fcfs scheduling
     void thread_tick (void);
```

4) New function defined in thread/synch.c:

```
    //to take out all the waiting threads out to the ready queue
     void sema_up_all(struct semaphore *);
```

-------- IMPLEMENTATION ALGORITHM --------

1) We define following struct and then initialize a list of it

```
  struct pthread_info{
    int pthread_id;
    int detachstate;
    void *value_ptr;
    struct semaphore running;
    struct list_elem elem;
    int done;
  };
```

The various elements are described below:

2) When a new thread is created a new struct of above type is created and added to the list. It contains the detachstate, return value(value_ptr) etc in it. (Each of them is explained below.)

3) When a thread is exited and its state is detached then this struct is deleted from the list. Otherwise it calls sema_up_all(&running) which is defined by us in synch.c and it brings all the process waiting for join on this pocess into the ready queue. The first function which acquires listlock will get the return value, sets done = 1(so that other waiting elements dont get the return val), and sets invalid pthread_id so that new calls on this thread see that this thread has exited.

4) For pthread_cancel, the list element of the above thread is removed and thread_cancel (defined above) is called to remove the thread from the ready queue and all_queue.

5) Initialization and destruction of attr elements is self explanatory.

6) For pthread_setschedparam, we call set_sched_policy(defined above) to set the global scheduling policy, and call thread_set_priority_now(defined above) to set the priority for the thread. note: we are passing only an integer as the third argument as all the 3 scheduling schemes require atmost sched_priority.

-------- SYNCHRONIZATION --------

1) 1 semaphore per thread is used for waiting to join.
2) overall 2 locks are used to wait for synchronizing reading and writing from pthread_list and thread_count.
3) There can be NO DEADLOCKS because following condition for deadlock is not satisfied:
   a) There is NO HOLD AND WAIT. There is no waiting for another process while holding a semaphore/lock.
  The only way a user can run into a deadlock is by deliberately doing it like joining on itself or 2 processes mutually joining on each other.
4) MUTUAL EXCLUSION: It is satisfied as any access to shared resources is bounded by locks
5) Bounded waiting cant be ensured in case of priority scheduling.

-------- TEST CASES --------

1) tests/threads/mytest.c : Tests the basic functioning of all the functions. (uncomment 25th line to test pthread_cancel)
2) tests/thread/mytest2.c : Tests the working of priority scheduling (Thread which is passed in line 30 is executed first because it gets higher priority (40))
3) tests/threads/mytest3.c: Round robin vs FCFS can be tested using this. Put a printf in thread_ticks to see that thread_ticks reset in case of RR but do not in case of FCFS
4) tests/threads/mytest4.c: Tests the case when multiple threads join on 1 thread simultaneously. Also checks priority scheduling. According to current implementation 2 threads join on a thread of lower priority. The exit value of the thread gets passed to the process with higher priority.

---- PART - 3 ---- PRODUCER CONSUMER PROBLEM ----

1) The solution for producer consumer problem is located at: tests/threads/prod_cons.c

2) It can be clearly seen by running the code that producer is able too produce 52 items numbered 0-51 and consumer is able to consume all of them, despite the fact that the value produced was available at a time different than the time it was loaded into buffer.

-------- IMPLEMENTATION ALGORITHM --------

1) A Bounded buffer of size 6 is there.
2) We define 3 semaphores: empty, mutex and full.
3) empty: Counts the no. of empty buffers. Initialized to 6.
4) full: Counts the no. of full buffers. Initialized to 0.
5) mutex: providex mutual exclusion to the buffer pools.
6) The basic outline of the code looks as follows:

Producers code

```
while(1){
 //produce the item
 wait(empty);
 wait(mutex);
 //add the item to the buffer
 signal(mutex);
 signal(full);
}
```

Consumers code

```
while(1){
 wait(full);
 wait(mutex);
 //remove item from buffer
 signal(mutex);
 signal(empty);
 //consume the item
}
```

7) MUTUAL EXCLUSION: The variable mutex provides mutual exclusion
8) PROGRESS AND BOUNDED WAITING: If a thread is waiting on full, then the other thread cant be waiting on empty and viceversa. Since the buffer is bounded so a process cant go on producing.