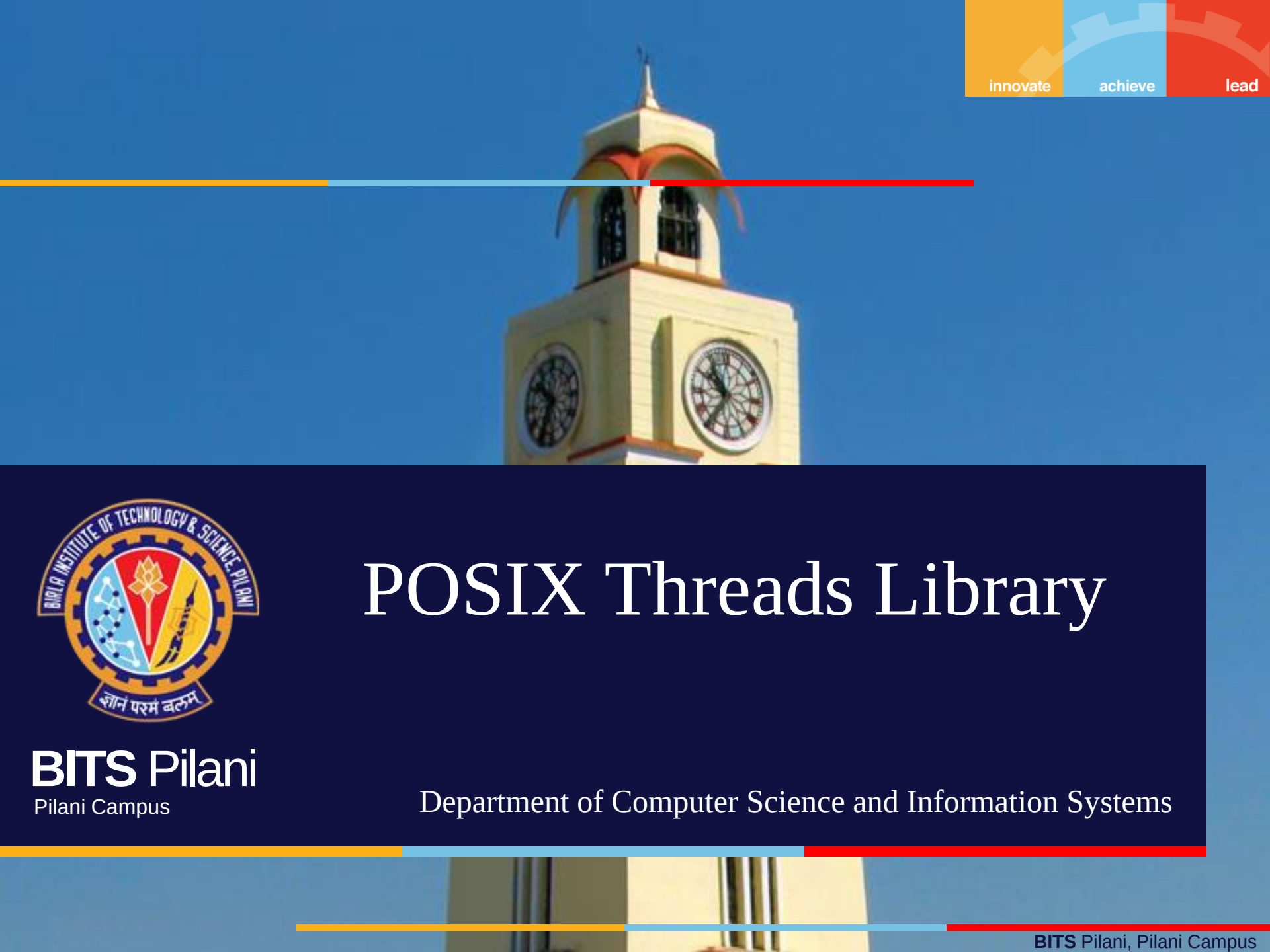


innovate achieve lead

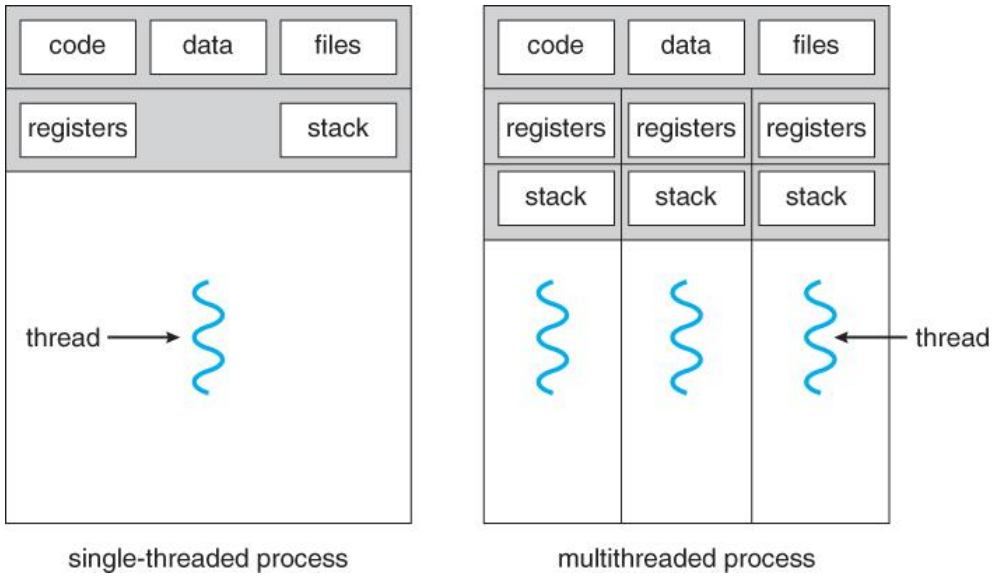
POSIX Threads Library

BITS Pilani
Pilani Campus

Department of Computer Science and Information Systems

What are threads?

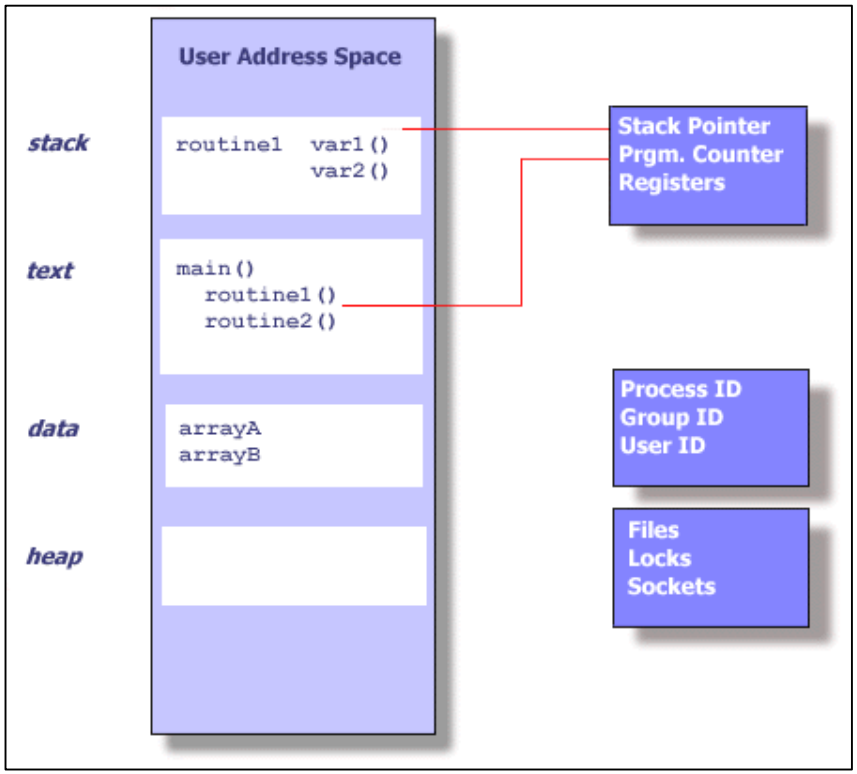
- A ***thread***, also called a ***lightweight process***, is a flow of control that can execute in parallel with other threads in the same program.
- A traditional or heavyweight process is equal to a task with one thread.
- As an OS may support multiple processes, a process can have multiple threads.



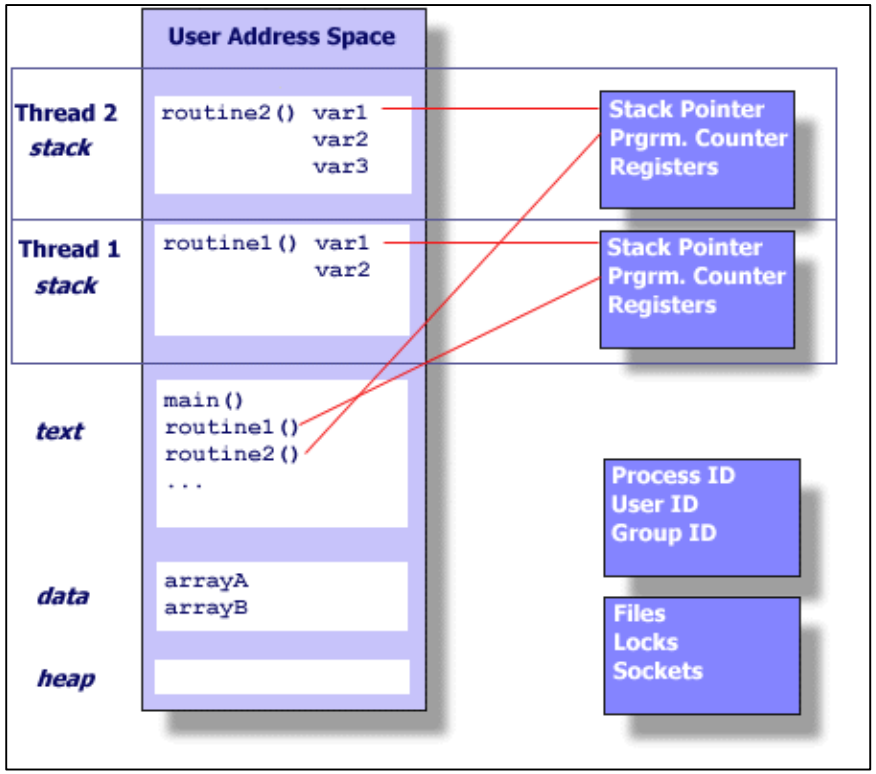
Thread Basics

- Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share: Process instructions, data, open files (descriptors), signals and signal handlers, current working directory, user and group id.
- Each thread has a unique: Thread ID, set of registers, stack pointer, stack for local variables, return addresses, signal mask, priority, Return value: *errno.*

Process vs Threads



UNIX process



Threads within a UNIX process

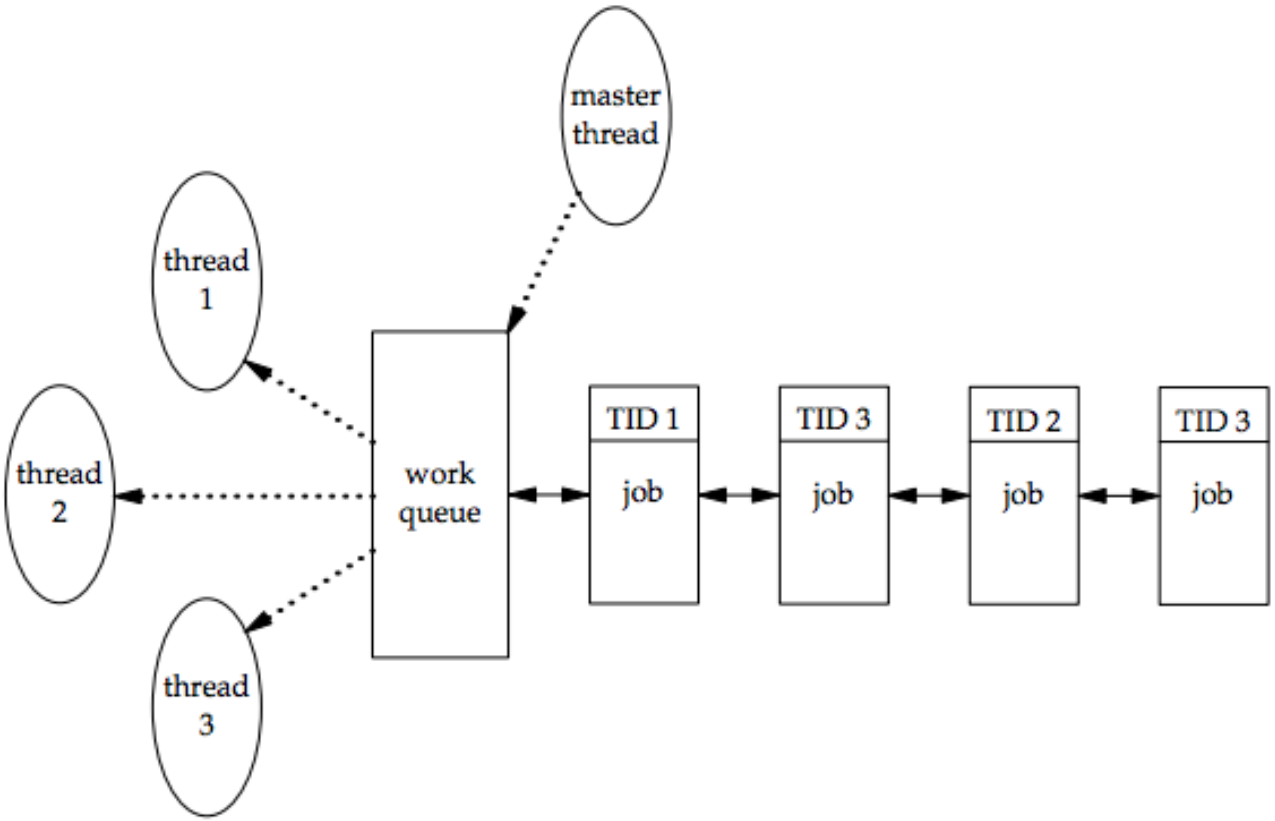
Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request.

Benefits of Threads

- Less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Less communication overheads

Work Queue Example



POSIX Threads Library

- POSIX (*Portable Operating System Interface for uni-X*) is a set of standard operating system interfaces based on the UNIX operating system.
- POSIX specifies a set of interfaces (functions, header files) for threaded programming commonly known as POSIX threads, or *Pthreads*.
- *Pthreads* is the key model for programming with threads in nearly every language and setup that is built with high level languages, such as C, Java and python, etc.
- The lifecycle of a thread, much like a process, begins with creation. But, threads are not forked from a parent to create a child, instead they are simply created with a starting function as the entry point. A thread does not terminated, like a process, instead they are joined with the main thread or they are detached to run on their own until completion.

POSIX THREAD SYSTEM CALLS

pthread_create(): creating a thread

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

Description:

The pthread_create() function starts a new thread in the calling process. On success, it returns 0; on error, it returns an error number.

Parameters:

thread	is the location where the ID of the newly created thread should be stored, or NULL if the thread ID is not required
attr	is the thread attribute object specifying the attributes for the thread that is being created. If attr is NULL, the thread is created with default attributes
start	is the main function for the thread; the thread begins executing user code at this address
arg	is the argument passed to start

Example:

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

void * hello(void *input)
{
    printf("%s\n", (char *)input);
    pthread_exit(NULL);
}

int main(void)
{
    pthread_t tid;
    pthread_create(&tid, NULL, hello, "hello world");
    pthread_join(tid, NULL);
    return 0;
}
```

Example:

```
pthread_t ntid;

Void printids(const char *s) {
    pid_t  pid;
    pthread_t  tid;
    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid, (unsigned long)tid, (unsigned long)tid);
}

void *thr_fn(void *arg) {
    printids("new thread: ");
    return((void *)0);
}

Int main(void) {
    int  err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "can't create thread");
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

O/P (on solaris):

main thread: pid 20075 tid 1 (0x1)
new thread: pid 20075 tid 2 (0x2)

pthread_join: wait for thread termination

A join is performed when one wants to wait for a thread to finish. A thread calling routine may launch multiple threads then wait for them to finish to get the results.

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

Description:

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. On success, it returns 0; on error, it returns an error number.

Parameters:

thread	Is the thread to wait for
status	Is the location where the exit status of the joined thread is stored. This can be set to NULL if the exit status is not required

pthread_detach: detach a thread

Indicates that storage for the thread can be reclaimed when the thread terminates.

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

Description:

A Detached thread automatically releases it allocated resources on exit. No other thread needs to join it. But by default all threads are joinable, so to make a thread detached we need to call **pthread_detach()** with thread id.

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

pthread_exit: thread termination

```
#include <pthread.h>
void pthread_exit(void *status);
```

Description:

Ends the calling thread and makes status available to any thread that calls pthread_join() with the ending thread's thread ID.

It cannot return to its caller. This function always succeeds.

Example:

```
#include <stdio.h><stdlib.h><pthread.h>
void *print_message_function( void *ptr );
main(){
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;

   /* Create independent threads each of which will execute function */
    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

    /* Wait till threads are complete before main continues. Unless we wait we run the risk */
    /* of executing an exit which will terminate the process and all threads before the */
    /* threads have completed.   */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}
```

Example:

```
void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

O/P:

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Thread Synchronization:

- Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section.
- The threads library provides three synchronization mechanisms:
  - **Mutex (Mutual exclusion lock)**: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
  - **Join-** Make a thread wait till others are complete (terminated).
  - **Condition variable-** data type pthread_cond_t

MUTEX:

- MUTEX syste calls:

  *int pthread_mutex_lock(pthread_mutex_t *mutex);*

  *int pthread_mutex_trylock(pthread_mutex_t *mutex);*

  *int pthread_mutex_unlock(pthread_mutex_t *mutex);*

- Return values:
  - If successful, the pthread_mutex_lock() and pthread_mutex_unlock() functions shall return zero; otherwise, an error number shall be returned to indicate the error.
  - The pthread_mutex_trylock() function shall return zero if a lock on the mutex object referenced by mutex is acquired. Otherwise, an error number is returned to indicate the error.

Example:

```
void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main(){
  int rc1, rc2;
  pthread_t thread1, thread2;

  /* Create independent threads each of which will execute functionC */
  if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
  {     printf("Thread creation failed: %d\n", rc1); }
  if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
  {     printf("Thread creation failed: %d\n", rc2);   }

  /* Wait till threads are complete before main continues. Unless we wait we run the risk of */
  /* executing an exit which will terminate the process and all threads before the threads */
  /* have completed.   */

  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);
  exit(0);
}
```

Example:

```
void *functionC()
{
  pthread_mutex_lock( &mutex1 );
  counter++;
  printf("Counter value: %d\n",counter);
  pthread_mutex_unlock( &mutex1 );
}
```

O/P:
Counter value: 1
Counter value: 2

Comparison of process and thread primitives

Process primitive	Thread primitive	Description
fork	pthread_create	create a new flow of control
exit	pthread_exit	exit from an existing flow of control
waitpid	pthread_join	get exit status from flow of control
atexit	pthread_cleanup_push	register function to be called at exit from flow of control
getpid	pthread_self	get ID for flow of control
abort	pthread_cancel	request abnormal termination of flow of control

QUERY?