Pthread

1 TABLE OF CONTENT

Contents

2 ABSTRACT & OVERVIEW

In shared memory multiprocessor architectures, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

The subroutines which comprise the Pthreads API can be informally grouped into four major groups:

1. **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
2. **Mutexes:** Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
3. **Condition variables:** Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
4. **Synchronization:** Routines that manage read/write locks and barriers.

3 SETUP & PROTOTYPE FOR CODE

In linux:

Write C-file

```
#include <file.h> // Include .h file required for your program
#include <pthread.h> // contains the Special Variables and Declaration of all pthread functions
….
….
pthread_[function group name]_<groupspecific work>(arg1, arg2, …)  //Remember every
pthread fuction is of this form, Every pthread function name starts with “pthread_” and
followed by that functionality group name (sometimes it is not present) and after that work or
functionality from that group (for ex. 1. pthread_create() => “create” is functionality , 2.
Pthread_key_init() => “key” is functionality group name of in further topics we will see its use,
“init” is function related to key we will see it in detail in further topic),

…
…
Call the various pthread function as per requirement
….
```

Compile: gcc filename.c –lpthread

In case of error such as undefined references to pthread functions:

Do : sudo apt-get install libpthread-stubs0-dev or sudo apt-get install build-essential

Run: ./a.out

4 THREAD MANAGEMENT

pthread_create

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

Description:

Creates & Starts executing, new thread whose identifier get saved in location "thread", attr can be NULL which defines various attributes for special functionalities we will see it in detail now consider, "start_routine" expects the function pointer of this [void *function(void *)] type (i.e. Just function name which is going to run as your thread), arg – argument to newly created thread here it expects the location of arguments where you have defined your arguments.

Remember* : Every thread must access unique address else you will may or may not endup with race condition. It is OK to access the common "static"-> not changing variable by all threads.

Returns 0 on success else non-zero value.

For more details man pthread_create

pthread_join

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

Description:

After creation of threads, We should let them finish completely. So to achive this intention pthread_join waits till the thread specified in argument "thread" is not get finished and second argument implies the "retval" of the thread

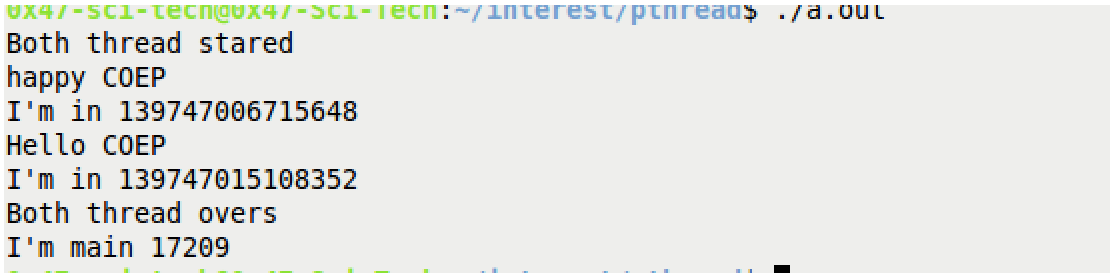
Remember*:"thread" for which you are waiting must be joinable, by default every thread is joinable if the arugment to "attr" is NULL and we can detach function provided by the pthread library.

Example(ex-001)

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

```
void  *print_coep(void *arg) {
   printf("%s COEP\n", (char *)arg);
   printf("I'm in %lu\n", pthread_self());
}
int main() {
   int iret;
   pthread_t t1, t2;
   char *arg1 = "Hello", *arg2= "happy";
   printf("Both thread stared\n");
   if(pthread_create(&t1, NULL, print_coep, arg1)) {
      fprintf(stderr, "create failed\n");
      exit(-1);
   }
   if(pthread_create(&t2, NULL, print_coep, arg2)) {
      fprintf(stderr, "create failed\n");
      exit(-1);
   }
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   printf("Both thread overs\n");
   printf("I'm main %lld\n", getpid());
   return 0;
}
```

Output of the above example(ex-001):



```
0x47-sci-tech@0x47-Sci-Tech:~/interest/pthread$ ./a.out
Both thread stared
happy COEP
I'm in 139747006715648
Hello COEP
I'm in 139747015108352
Both thread overs
I'm main 17209
```

Explanation : Above C-code(ex-001): Creates two threads one prints “Hello COEP” & on prints “happy COEP” ;but due to scheduling problem order of printing changes every time you run the code after compiling it appropriately.

There are many other functions which can be used for Thread Management such as **void pthread_exit(void *retval**) : To exit the current thread with particular status.

Cleanup: the pthread_exit() routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

Discussion on calling pthread_exit() from main():

There is a definite problem if main() finishes before the threads it spawned if you don't call pthread_exit() explicitly. All of the threads it created will terminate because main() is done and no longer exists to support the threads.

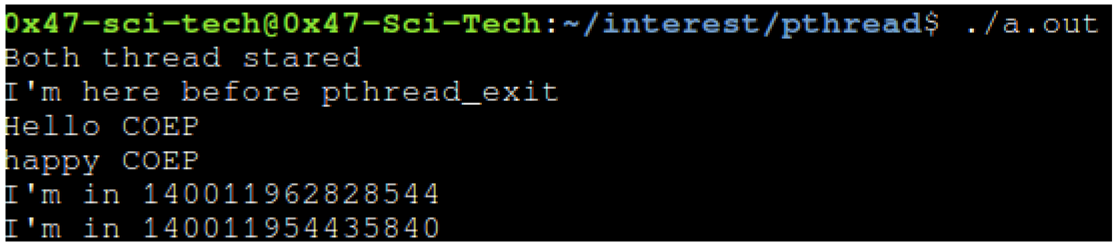
By having main() explicitly call pthread_exit() as the last thing it does, main() will block and be kept alive to support the threads it created until they are done.

Example(ex-002):pthread_exit

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

void  *print_coep(void *arg) {
   printf("%s COEP\n", (char *)arg);
   sleep(5);
   printf("I'm in %lu\n", pthread_self());
}

int main() {
   int iret;
   pthread_t t1, t2;
   char *arg1 = "Hello", *arg2= "happy";

   printf("Both thread stared\n");
   if(pthread_create(&t1, NULL, print_coep, arg1)){
      fprintf(stderr, "create failed\n");
      exit(-1);
   }
   if(pthread_create(&t2, NULL, print_coep, arg2)){
      fprintf(stderr, "create failed\n");
      exit(-1);
   }
```

```
    printf("I'm here before pthread_exit\n");
    pthread_exit(NULL); //this tries to exit main ;but main doesn't exits until t1, t2 not get finished
    printf("I'm here after pthread_exit\n");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Both thread overs\n");
    printf("I'm main %lld\n", getpid());
    return 0;
}
```

Output (ex-002):


```
0x47-sci-tech@0x47-Sci-Tech:~/interest/pthread$ ./a.out
Both thread stared
I'm here before pthread_exit
Hello COEP
happy COEP
I'm in 140011962828544
I'm in 140011954435840
```

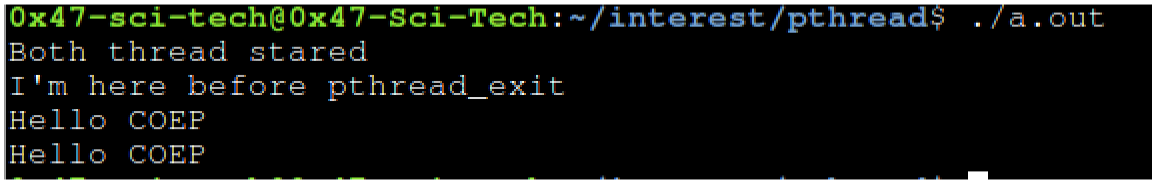
Explanation (ex-002):

Although both the threads are in progress, then, master thread(main) trying to exit using pthread_exit, but it gets blocked till all thread does get finished.

Example of **exit** system call (ex-003):

If we change pthread_exit(NULL); by exit(0) in ex-002 program, It will not wait to finish t1 and t2

Output of (ex-003)


```
0x47-sci-tech@0x47-Sci-Tech:~/interest/pthread$ ./a.out
Both thread stared
I'm here before pthread_exit
Hello COEP
Hello COEP
```

int pthread_detach(pthread_t thread): Detach the current thread from the "thread" specified in the argument and fails if threads are not joinable or NO thread with have id "thread"

pthread_t pthread_self(): on success returns the identifier of the current thread (this is long unsigned int), this function always succeeds as per their documentation

int pthread_equal(pthread_t thread1, pthread_t thread2): Checks the equality of both the thread ids thread1 and thread2

int pthread_cancel(pthread_t thread): Sends the cancellation request to the thread whose identifier is "thread", Returns 0 on success and non-zero when it fails, It fails if there is NO thread with identifier "thread" example is provided in manpage of this function

We can set the cancel state & its type using the functions **pthread_setcancelstate, pthread_setcanceltype** of the pthread library.

int pthread_yield(void): Causes the calling thread to leave the CPU . The thread is placed at the last of the queue and another thread is scheduled to run.

There are many such functionalities provided by the pthread library just do "man pthread_" and hit <tab button> in the terminal of linux to see all of them.

5 THREAD ATTRIBUTE OBJECTS

- pthread_attr_init and pthread_attr_destroy are used to initialize/destroy the thread attribute object.
- Other routines are then used to query/set specific attributes in the thread attribute object. Attributes include:
  - Detached or joinable state
  - Scheduling inheritance
  - Scheduling policy
  - Scheduling parameters
  - Scheduling contention scope
  - Stack size
  - Stack address
  - Stack guard (overflow) size

There are many more attributes that we can set using the all functions "pthread_attr_<some name>(arguments, ….)" . We will see the one example of the attribute in the scheduling of the threads where we set the value of some of parameters mentioned above. And its depends upon the application that which attributes are necessary for you.

6 SYNCHRONIZATION-BASED

To Synchronize the parallel access to data various functionalities, such as mutexes, barriers

Mutexes:

Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Various function related to Mutexes:

pthread_mutex_<function work name>(arguments, ...)

pthread_mutexattr_<function to handle various attribute of mutex>(arguments, …)

int pthread_mutex_init(pthread_mutex_t *restrict mutex,

const pthread_mutexattr_t *restrict attr);

Initializes the mutex variable as the specified attr and to default values if “restrict attr” is NULL

REMEMBER*: pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; is same as pthread_mutex_init(address of mutex variable, NULL);

Use of mutex variable without initialization is undefined. Refer man pages

int pthread_mutex_destroy(pthread_mutex_t *mutex);

Resources related to mutex variable is given back to system.

We can set our own attribute variable using following functions

int pthread_mutexattr_init(pthread_mutexattr_t *attr); : follow this link for example

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

There are many other function to set functionalities of **“mutexattr”, refer man pages**

int pthread_mutex_lock(pthread_mutex_t *mutex);//blocking lock function

Wait until lock is not acquired, It also returns success and failure values refer man pages.

int pthread_mutex_trylock(pthread_mutex_t *mutex);//non-blocking lock function

Never wait until lock is acquired. If lock is not acquired return failure. It returns specific success and failure values on that basis we have to decide what to do

int pthread_mutex_unlock(pthread_mutex_t *mutex);

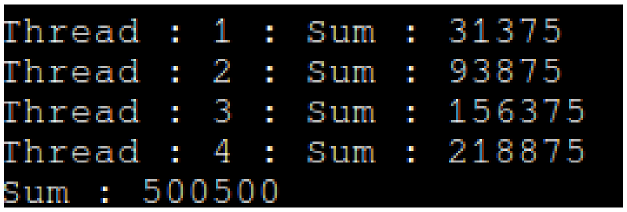
Unlock the mutex lock. Unlocking the mutex which not locked then, result is undefined as per their documentation refer man pages.

Example(ex-004):Mutexes

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

#define NUM_THREADS 4
#define N 1000

pthread_mutex_t sum_mutex1=  PTHREAD_MUTEX_INITIALIZER;//Mutex for sum variable

int a[N], sum;

void initialize_array(int *a, int n) {
   int i;
   for (i = 1; i <= n; i++) {
      a[i-1] = i;
   }
}
void *thread_add(void *argument) {
   int i, thread_sum, arg;
   thread_sum = 0;
   arg = *(int *)argument;
   for (i = ((arg-1)*(N/NUM_THREADS)) ; i < (arg*(N/NUM_THREADS)); i++) {
      thread_sum += a[i];
   }
   printf("Thread : %d : Sum : %d\n", arg, thread_sum);
   pthread_mutex_lock(&sum_mutex1);
   sum += thread_sum;// mutex for sum variable accessed by all thread simultaneously
   pthread_mutex_unlock(&sum_mutex1);
}
```

```
void print_array(int *a, int size) {
  int i;
  for(i = 0; i < size; i++)
    printf(" %d ", a[i]);
  putchar('\n');
}
int main() {
  int i, *range;
  pthread_t thread_id[NUM_THREADS];
  sum = 0;
  initialize_array(a, N);
  printf("Array : \n");
  print_array(a, N);
  for (i = 0; i < NUM_THREADS; i++) {
    range = (int *)malloc(sizeof(int));
    *range = (i+1);
    if (pthread_create(&thread_id[i], NULL, thread_add, (void *)range))
      printf("Thread creation failed for thread num : %d\n", *range);
  }
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(thread_id[i], NULL);
  }
  printf("Sum : %d\n", sum);
  return 0;
}
```

Output (ex-004)



```
Thread : 1 : Sum : 31375
Thread : 2 : Sum : 93875
Thread : 3 : Sum : 156375
Thread : 4 : Sum : 218875
Sum : 500500
```

Explanation (ex-004):

Above divides the work of addition in 4-different threads and the partial sum done by every thread is get added to common global variable **sum** and every thread trying to write the same memory location, so, to provide the exclusive access **sum_mutex1** is used to lock and unlock the access to **sum** variable.

RW Locks(Read Write):

Every time mutex lock might be wastage of time instead, we can continue the parallelism by using "pthread_rwlock" which behaves as per following table.

	TRY FOR READ LOCK	TRY FOR WRITE LOCK
ALREADY READ LOCKED	ALLOWED	NOT ALLOWED
ALREADY WRITE LOCKED	NOT ALLOWED	NOT ALLOWED

Pthread provides various functions related to rwlock as follows:

int pthread_rwlock_init(pthread_rwlock_t *restrict *rwlock*,
const pthread_rwlockattr_t *restrict *attr*);

Initializes the "rwlock" using "restrict attr", and if it is NULL default attributes are set.

To set various attributes of "**rwlock**" using "**pthread_rwlockattr_t"** functions are provided

int pthread_rwlockattr_setkind_np(pthread_rwlockattr_t **attr*, int *pref***);**

int pthread_rwlockattr_getkind_np(const pthread_rwlockattr_t **attr*, **int **pref*);**

int pthread_rwlock_destroy(pthread_rwlock_t **rwlock*);

Destroys the "rwlock" that is resources related to "rwlock" get freed.

int pthread_rwlock_rdlock(pthread_rwlock_t **rwlock*);

Acquires the read lock and wait till "Read" lock is not get acquired or returns the failure if fails to acquire.
int pthread_rwlock_tryrdlock(pthread_rwlock_t **rwlock*);

Acquires the read lock ;but it never wait till the "Read" lock is not get acquired. It is non-blocking function. It fails it is not able to acquire the read lock and returns the failure status

Similar to read lock there are functions for write locks

int pthread_rwlock_wrlock(pthread_rwlock_t **rwlock*);

int pthread_rwlock_trywrlock(pthread_rwlock_t **rwlock*);

There other functions also refer man pages.

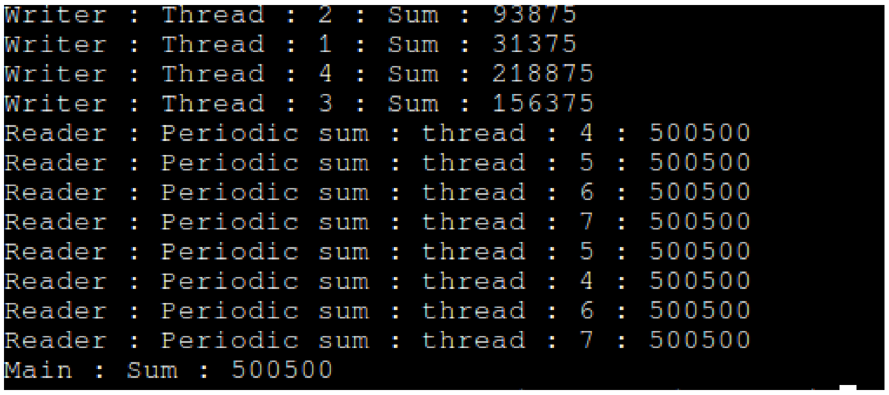
int pthread_rwlock_unlock(pthread_rwlock_t *_rwlock_);

The *pthread_rwlock_unlock*() function shall release a lock held on the read-write lock object referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

Example (ex-005)

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

#define NUM_THREADS 4
#define N 1000

pthread_rwlock_t sum_rwlock=PTHREAD_RWLOCK_INITIALIZER;

int a[N], sum;

void initialize_array(int *a, int n) {
   int i;
   for (i = 1; i <= n; i++) {
      a[i-1] = i;
   }
}
void *thread_add(void *argument) {
   int i, thread_sum, arg;
   thread_sum = 0;
   arg = *(int *)argument;
   for (i = ((arg-1)*(N/NUM_THREADS)) ; i < (arg*(N/NUM_THREADS)); i++) {
      thread_sum += a[i];
   }
   pthread_rwlock_wrlock(&sum_rwlock);
   printf("Writer : Thread : %d : Sum : %d\n", arg, thread_sum);
   sum += thread_sum;
   pthread_rwlock_unlock(&sum_rwlock);
   sleep(2);
}

void *read_periodic_sum(void *argument){
   int i=0;
   while(i < 2) {
      pthread_rwlock_rdlock(&sum_rwlock);
      printf("Reader : Periodic sum : thread : %ld : %d\n", (long)argument, sum);
      pthread_rwlock_unlock(&sum_rwlock);
      i++;
      sleep(1);
   }
}
void print_array(int *a, int size) {
   int i;
   for(i = 0; i < size; i++)
      printf(" %d ", a[i]);
   putchar('\n');
}
int main() {
   int i, *range;
   pthread_t thread_id[NUM_THREADS * 2];

   sum = 0;
   initialize_array(a, N);
   printf("Array : \n");
   print_array(a, N);

   for (i = 0; i < NUM_THREADS; i++) {
      range = (int *)malloc(sizeof(int));
      *range = (i+1);
      if (pthread_create(&thread_id[i], NULL, thread_add, (void *)range))
         printf("Thread creation failed for thread num : Writer : %d\n", *range);
   }
   for (; i < NUM_THREADS + NUM_THREADS; i++)
      if (pthread_create(&thread_id[i], NULL, read_periodic_sum, (void *)i))
         printf("Thread creation failed for thread num : Reader : %d\n", *range+1);

   for (i = 0; i < NUM_THREADS; i++) {
      pthread_join(thread_id[i], NULL);
   }
   pthread_join(thread_id[i], NULL);
   printf("Main : Sum : %d\n", sum);
   return 0;
}
```

Output (ex-005)



```
Writer : Thread : 2 : Sum : 93875
Writer : Thread : 1 : Sum : 31375
Writer : Thread : 4 : Sum : 218875
Writer : Thread : 3 : Sum : 156375
Reader : Periodic sum : thread : 4 : 500500
Reader : Periodic sum : thread : 5 : 500500
Reader : Periodic sum : thread : 6 : 500500
Reader : Periodic sum : thread : 7 : 500500
Reader : Periodic sum : thread : 5 : 500500
Reader : Periodic sum : thread : 4 : 500500
Reader : Periodic sum : thread : 6 : 500500
Reader : Periodic sum : thread : 7 : 500500
Main : Sum : 500500
```

Explanation (ex-005):

Above code is the copy of the code of the “mutex code” but instead of mutex lock every thread uses the “wrlock” and new threads are added with prints the random value of the “sum” global variable as “read_periodic_sum” get scheduled using “rdlock”.

Condition Variables:

Many times it is not good to wait, rather it is good to wait to happen particular event without wasting the processing for processing.

So, Pthread provides the conditional variables by which you can wait conditionally and by signaling to conditional variable we can stop the waiting of all thread which are waiting on that condition variable.

There are many condition variable related function provided by pthread as follows:

int pthread_cond_init(pthread_cond_t *restrict cond,

const pthread_condattr_t *restrict attr);

Initializes the condition variable by specifying the address of it as first argument and address of the pthread_condattr_t variable by which we can specify various attributes of condition variable and if it NULL then default values are considered

Remember*: Use of Uninitialized condition variable may result into the undefined output

pthread_cond_t cond = PTHREAD_COND_INITIALIZER; which same as passing NULL in for restrict_attr in pthread_cond_init

int pthread_cond_destroy(pthread_cond_t *cond);

Destroys the condition variable specified whose address is specifed as “cond”. (i.e. giving all resources used by condition variable whose addressed is specified in “cond”)

int pthread_cond_wait(pthread_cond_t *restrict *cond*,
pthread_mutex_t *restrict *mutex*);

Lock mutex and wait for signal. Note that the pthread_cond_wait routine
will automatically and atomically unlock mutex while it waits.

int pthread_cond_timedwait(pthread_cond_t *restrict *cond*,
pthread_mutex_t *restrict *mutex*,
const struct timespec *restrict *abstime*);

The *pthread_cond_timedwait*() function shall be equivalent to *pthread_cond_wait*(), except that an error is returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time specified by *abstime* has already been passed at the time of the call. It is up to us what to do next If it fails to lock.

int pthread_cond_broadcast(pthread_cond_t *cond*);
int pthread_cond_signal(pthread_cond_t *cond*);

“pthread_cond_signal” stops the waiting of **one** of the thread on condition variable specified by “cond” in the **queue**, where as “broadcast” stops the waiting of **all** of the thread on condition variable specified by “cond”

There are many function provided by pthread to configure condition attribute variables some of them as follows:

int pthread_condattr_init(pthread_condattr_t *attr);

Initializes the pthread_condattr_t variable with the default values specified pthread.

Remember*: Use of Initialized variables may result into undefined output.

int pthread_condattr_destroy(pthread_condattr_t *attr);

Destroys the pthread_condattr_t variable specified in “attr”

There are many other function which are actually useful to set many properties of pthread_condattr_t variable. Refer man pages for pthread_condattr_<function name>(arguments, ..)

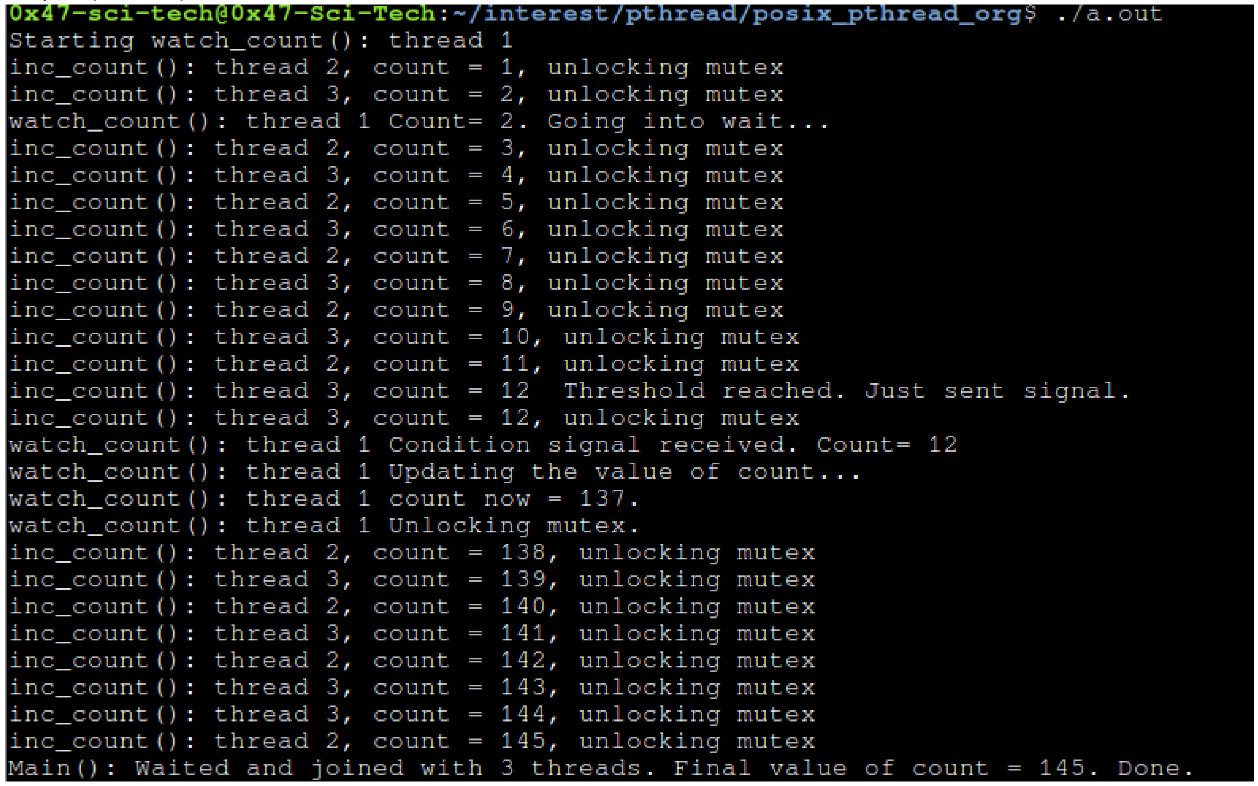
Example : Condition Variable(ex-006)

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

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ",
             my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
                my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
```

```
   sleep(1);
   }
  pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id = (long)t;
  printf("Starting watch_count(): thread %ld\n", my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
  will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld Count= %d. Going into wait...\n", my_id,count);
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received. Count= %d\n", my_id,count);
    printf("watch_count(): thread %ld Updating the value of count...\n", my_id,count);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;
```

```
 /* Initialize mutex and condition variable objects */
 pthread_mutex_init(&count_mutex, NULL);
 pthread_cond_init (&count_threshold_cv, NULL);

 /* For portability, explicitly create threads in a joinable state */
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
 pthread_create(&threads[0], &attr, watch_count, (void *)t1);
 pthread_create(&threads[1], &attr, inc_count, (void *)t2);
 pthread_create(&threads[2], &attr, inc_count, (void *)t3);
 /* Wait for all threads to complete */
 for (i = 0; i < NUM_THREADS; i++) {
   pthread_join(threads[i], NULL);
 }
 printf ("Main(): Waited and joined with %d threads. Final value of count = %d. Done.\n",
         NUM_THREADS, count);
 /* Clean up and exit */
 pthread_attr_destroy(&attr);
 pthread_mutex_destroy(&count_mutex);
 pthread_cond_destroy(&count_threshold_cv);
 pthread_exit (NULL);
}
```

Output (ex-006)



```
0x47-sci-tech@0x47-Sci-Tech:~/interest/pthread/posix_pthread_org$ ./a.out
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 Count= 2. Going into wait...
inc_count(): thread 2, count = 3, unlocking mutex
inc_count(): thread 3, count = 4, unlocking mutex
inc_count(): thread 2, count = 5, unlocking mutex
inc_count(): thread 3, count = 6, unlocking mutex
inc_count(): thread 2, count = 7, unlocking mutex
inc_count(): thread 3, count = 8, unlocking mutex
inc_count(): thread 2, count = 9, unlocking mutex
inc_count(): thread 3, count = 10, unlocking mutex
inc_count(): thread 2, count = 11, unlocking mutex
inc_count(): thread 3, count = 12  Threshold reached. Just sent signal.
inc_count(): thread 3, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received. Count= 12
watch_count(): thread 1 Updating the value of count...
watch_count(): thread 1 count now = 137.
watch_count(): thread 1 Unlocking mutex.
inc_count(): thread 2, count = 138, unlocking mutex
inc_count(): thread 3, count = 139, unlocking mutex
inc_count(): thread 2, count = 140, unlocking mutex
inc_count(): thread 3, count = 141, unlocking mutex
inc_count(): thread 2, count = 142, unlocking mutex
inc_count(): thread 3, count = 143, unlocking mutex
inc_count(): thread 3, count = 144, unlocking mutex
inc_count(): thread 2, count = 145, unlocking mutex
Main(): Waited and joined with 3 threads. Final value of count = 145. Done.
```

Explanation(ex-006)

Above code, contains two types of thread one which increatments “count” variable inc_count() and another which is waiting for “count == COUNT_LIMIT” it signals to the all the thread which are waiting on the condition variable “count_threshold_cv”.

Here we require the mutex, because to avoid the race condition such as one thread is accessing the “count_threshold_cv” itself is the shared the variable among the various threads and “pthread_cond_wait” unlocks the mutex after wait has set and mutex again get locked once the wait is over due to signal from another thread. visit here for ref http://stackoverflow.com/questions/2763714/why-do-pthreads-condition-variable-functions-require-a-mutex

Waiting thread must be in the loop or some sort of condition, because, To avoid waiting of the threads which are created after “count == COUNT_LIMIT” reached(i.e when count > COUNT_LIMIT)
thats why above “watch_count” contains “while(count < COUNT_LIMIT)”.

For remaining “attr” and “destroy” related function refer man pages

As an assignment we have solved the problem from POSIX exercise. And problem statement is “Now, review, compile and run the bug1.c program. Observe the output of the five threads. What happens? See if you can determine why and fix the problem. “ google it you will find this.

Actually bug is There are five threads, Four contains the “pthread_cond_wait” and One thread signals i.e. contains “pthread_cond_signal” , One thread was get freed due to signal ;but stops waiting forever. Thats why, code get hanged.

So, solution for it is instead of “signal” use “broadcast”.

Barriers:

As every thread get different timeslice during it's execution of time till particular time. (I.e every thread may or may not be at different statement at any point of time). so, if you want to ensure that every thread should wait at particular statement at same time then, barrier is the solution for it.

For example following every thread will at BARRIER point till all three thread can't come to BARRIER point

THREAD 1	THREAD 2	THREAD 3
Statement 1	Statement 1	Statement 1
Statement 2	Statement 2	Statement 2
---------------BARRIER---------	---------------BARRIER---------	---------------BARRIER---------
…	…	…
Statement n	Statement n	Statement n

Barrier functions provided by pthread library

int pthread_barrier_init (pthread_barrier_t ****barrier*, const pthread_barrierattr_t ****attr***, unsigned int** ***count*****);**

Initializes the barrier variable with attributes variables and takes the count of number of thread you are going to create for which you want to barrier.

int pthread_barrier_wait (pthread_barrier_t ****barrier***);**

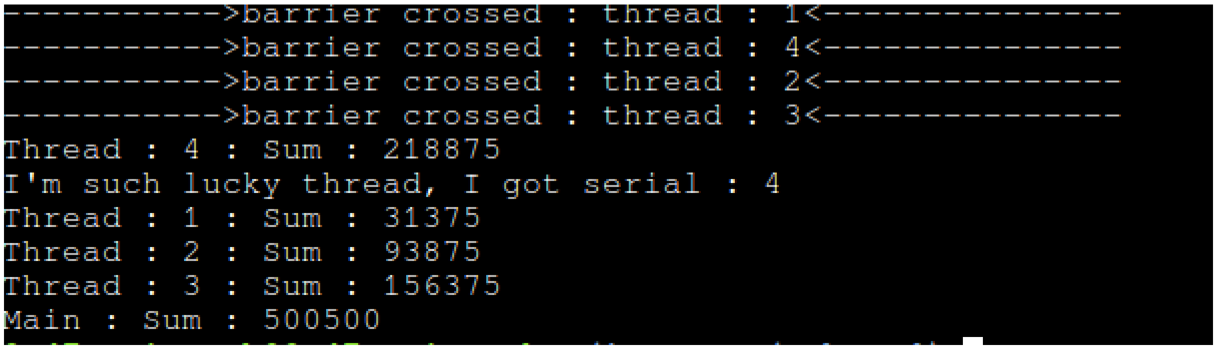
In above table the statement BARRIER is nothing but this function call. It waits on the “barrier” variable. And It returns the 0 in all thread except one in which It returns PTHREAD_BARRIER_SERIAL_THREAD. You can use this return value to do some work once. After passing the barrier point. It returns some error values when it fails. We as programmer have to decide what to do next, if barrier fails.

Example(ex-007)

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

#define NUM_THREADS 4
#define N 1000

pthread_barrier_t sum_barrier;//barrier
pthread_mutex_t sum_mutex1;//Mutex for sum variable

int a[N], sum;

void initialize_array(int *a, int n) {
   int i;
   for (i = 1; i <= n; i++) {
      a[i-1] = i;
   }
}
void i_got_serial(int arg) {
   printf("I'm such lucky thread, I got serial : %d\n", arg);
}

void *thread_add(void *argument) {
   int i, thread_sum, arg, ret;
   thread_sum = 0;
   arg = *(int *)argument;

   for (i = ((arg-1)*(N/NUM_THREADS)) ; i < (arg*(N/NUM_THREADS)); i++) {
      thread_sum += a[i];
   }
   ret = pthread_barrier_wait(&sum_barrier);//barrier point
printf("----------->barrier crossed : thread : %d<---------------\n", arg);
   sleep(1);// To give chance to print barrier crossed
   if (ret == PTHREAD_BARRIER_SERIAL_THREAD) {
      printf("Thread : %d : Sum : %d\n", arg, thread_sum);
      i_got_serial(arg);
```

```
  }
  else if (ret == 0)
    printf("Thread : %d : Sum : %d\n", arg, thread_sum);
  else
    printf("Barrier failed in thread: %d\n", arg);
//sum is shared global variable to maintain exclusive access
  pthread_mutex_lock(&sum_mutex1);
  sum += thread_sum;
  pthread_mutex_unlock(&sum_mutex1);
}
void print_array(int *a, int size) {
  int i;
  for(i = 0; i < size; i++)
    printf(" %d ", a[i]);
  putchar('\n');
}
void new_parent() {
  int i, *range;
  pthread_t thread_id[NUM_THREADS];

  sum = 0;
  initialize_array(a, N);
  printf("New Parent : Array : \n");
  print_array(a, N);

  for (i = 0; i < NUM_THREADS; i++) {
    range = (int *)malloc(sizeof(int));
    *range = (i+1);
    if (pthread_create(&thread_id[i], NULL, thread_add, (void *)range))
      printf("New Parent : Thread creation failed for thread num : %d\n", *range);
  }
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(thread_id[i], NULL);
  }
  printf("New Parent : Sum : %d\n", sum);
}
int main() {
  int i, *range;
  pthread_t thread_id[NUM_THREADS];

  sum = 0;
  initialize_array(a, N);

  pthread_barrier_init(&sum_barrier, NULL, NUM_THREADS);
  printf("Array : \n");
  print_array(a, N);

  for (i = 0; i < NUM_THREADS; i++) {
    range = (int *)malloc(sizeof(int));
    *range = (i+1);
    if (pthread_create(&thread_id[i], NULL, thread_add, (void *)range))
      printf("Thread creation failed for thread num : %d\n", *range);
  }
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(thread_id[i], NULL);
  }
  printf("Main : Sum : %d\n", sum);
  return 0;
}
```

Output(ex-007):



```
----------->barrier crossed : thread : 1<---------------
----------->barrier crossed : thread : 4<---------------
----------->barrier crossed : thread : 2<---------------
----------->barrier crossed : thread : 3<---------------
Thread : 4 : Sum : 218875
I'm such lucky thread, I got serial : 4
Thread : 1 : Sum : 31375
Thread : 2 : Sum : 93875
Thread : 3 : Sum : 156375
Main : Sum : 500500
```

Explanation(ex-007):

Above code is the copy of the (ex-004) only changes is, it puts the barrier after the calculation of partial sum of every thread.

7 THREAD-SPECIFIC DATA

Sometimes it is difficult to maintain the data related to every thread. For example consider following table

THREAD 1	THREAD 2	THREAD 3
….Generated Data...	….Generated Data...	….Generated Data...
Assume 50 function calls requires full data generated above	Assume 50 function calls requires full data generated above	Assume 50 function calls requires full data generated above

In Above example, It is very difficult to send the data to the 50-functions by using the “argument” or we can use global variables ;but if we don't know the no. of threads we are going to create then, static allocation will fail.

So, Solution for it is use dynamic allocation at run time use our own data structure in thread function and which becomes the difficult and you will endup with the memory leakage and so on. Or Simple solution is pthread library provides the concept called as the **key** which maintains the thread specific data generated at run time appropriately.

Keys:

There various function provided by pthread library as follow:

int pthread_key_create(pthread_key_t ****key*, **void (**destructor*)**(void*));**

Once you declare your “pthread_key_t” variable, you should pass it to “pthread_key_create” function its like the initialization function. It actually creates the **key** associated with you program. It takes address of “key” variable and function pointer of function which will be automatically get called as soon as thread will over and when it will get called one pointer will get passed to that function which is nothing ;but the data associated with your thread which you allocates during run time and you assign data to the key using pthread_setspecific function.

So this function which is passed “pthread_key_create” should free the allocated memory using the passed pointer. Refer example (ex-008)

int pthread_setspecific(pthread_key_t ***key*****, const void *****value*****);**

This function assigns / associates the data to key variable which can be accessed in future using “key” variable. Using get specific function.

You should call pthread_key_create function before calling this function. Otherwise results are undefined

void *pthread_getspecific(pthread_key_t *key*);

Returns the pointer to the data which is assigned using pthread_setspecific function of particular key. For more information refer man pages

int pthread_key_delete(pthread_key_t *key*);

Deletes the "Key". So, resources related to keys are freed.

Sequence of the function call in you program should be

pthread_key_create, pthread_setspecific, pthread_getspecific, pthread_key_delete

To get appropriate results

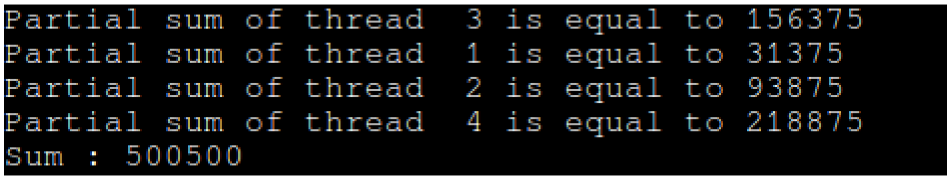
Example (ex-008)

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>


#define NUM_THREADS 4

#define N 1000


pthread_mutex_t sum_mutex1;//Mutex for sum variable

pthread_key_t psum_key;//Key

int a[N], sum;


void initialize_array(int *a, int n) {

   int i;

   for (i = 1; i <= n; i++) {

      a[i-1] = i;

   }

}


void print_partial_sum (int thread_no) {

   int *psum;


   psum = pthread_getspecific(psum_key);

   printf("Partial sum of thread  %d is equal to %d\n", thread_no, *psum);

}
```

```
void *thread_add(void *argument) {
   int i, thread_sum, arg;
   int *thread_sum_heap;
   thread_sum = 0;
   arg = *(int *)argument;
   for (i = ((arg-1)*(N/NUM_THREADS)) ; i < (arg*(N/NUM_THREADS)); i++) {
      thread_sum += a[i];
   }
   thread_sum_heap = (int *)malloc(sizeof(int));
   *thread_sum_heap = thread_sum;

   pthread_setspecific(psum_key, (void *)thread_sum_heap);
   print_partial_sum(arg); // it is going to use the psum_key to print the partial sum

   pthread_mutex_lock(&sum_mutex1);
   sum += thread_sum;
   pthread_mutex_unlock(&sum_mutex1);

}
void free_all(int *p) {
   free(p);
}
void initialize_keys () {
   pthread_key_create(&psum_key, (void *)free_all);
}
void print_array(int *a, int size) {
   int i;
   for(i = 0; i < size; i++)
      printf(" %d ", a[i]);
   putchar('\n');
}
int main() {
   int i, *range;
   pthread_t thread_id[NUM_THREADS];

   sum = 0;
   initialize_array(a, N);
```

```
  initialize_keys();//create key

  printf("Array : \n");
  print_array(a, N);

  for (i = 0; i < NUM_THREADS; i++) {
    range = (int *)malloc(sizeof(int));
    *range = (i+1);
    if (pthread_create(&thread_id[i], NULL, thread_add, (void *)range))
      printf("Thread creation failed for thread num : %d\n", *range);
  }

  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(thread_id[i], NULL);
  }
  pthread_key_delete(psum_key);
  printf("Sum : %d\n", sum);
  return 0;
}
```

Output (ex-008):



```
Partial sum of thread  3 is equal to 156375
Partial sum of thread  1 is equal to 31375
Partial sum of thread  2 is equal to 93875
Partial sum of thread  4 is equal to 218875
Sum : 500500
```

Explanation (ex-008):

Above example is the same as the (ex-004) only difference is to print the partial sum of every thread their function which takes the key and prints the data associated with it.

8 SCHEDULING

As we have seen that there are various attribute that we can set during thread creation and using pthread_attr_<function name> (arguments, …);

There are functionalities provided by the pthread by which we can schedule our thread within the process and outside the process on the global level by treating every thread as an indisivual process. Also we can specify which scheduling algorithm is to be used FIFO or Round Robin or Other custom.

Following are the function which helps us to achieve this power of thread scheduling

Your Threads ==>	PTHREAD ===> SCHEDULER	OS Scheduler ==>	CPU and resources

int pthread_attr_init(pthread_attr_t *attr);

Initializes the pthread_attr_t variable whose address is passed to it.

int pthread_attr_destroy(pthread_attr_t *attr);

Destorys the pthread_attr_t variable, that is frees the resources related to it.

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

It is important function for scheduling, By using this function only we specify, Where to do the scheduling between the threads of the same process (then scope will be : PTHREAD_SCOPE_PROCESS) or scheduling of the threads between the processes of the system (then scope will be : PTHREAD_SCOPE_SYSTEM)

It depends upon the underlying operating system what to support.

int pthread_attr_getscope(pthread_attr_t *attr, int *scope);

Fills the scope associated with “attr” variable at “scope” address

int pthread_attr_setinheritsched(pthread_attr_t *attr,

int inheritsched);

This functions tells that we are going to use the same attributes as the parent thread or NOT by specifying appropriate standard value in the variable “inheritsched”

PTHREAD_INHERIT_SCHED : Use parent settings

Threads that are created using attr inherit scheduling attributes from the creating thread; the scheduling attributes in attr are ignored.

PTHREAD_EXPLICIT_SCHED : Don't use parent settings, Instead use settings associated with “attr” variable settings

Threads that are created using attr take their scheduling attributes from the values specified by the attributes object.

int pthread_attr_getinheritsched(pthread_attr_t *attr, int *inheritsched);

Fills the inheritenced settings associated with “attr” variable at “inheritsched” address

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

We can specify the which scheduling algorithm we want for our thread scheduling by specifying “policy” variable appropriately.

The supported values for policy are SCHED_FIFO, SCHED_RR, and SCHED_OTHER, with the semantics described in sched_setscheduler(2).

int pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy);

Fills the scheduling policy name(#defined number) associated with “attr” variable at “policy” address

int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);

Set the priority of the thread we can use this function. For more details follow man pages

int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *param);

Fills the scheduling priority name(#defined number) associated with “attr” variable at “param” address

refer man pages for more details

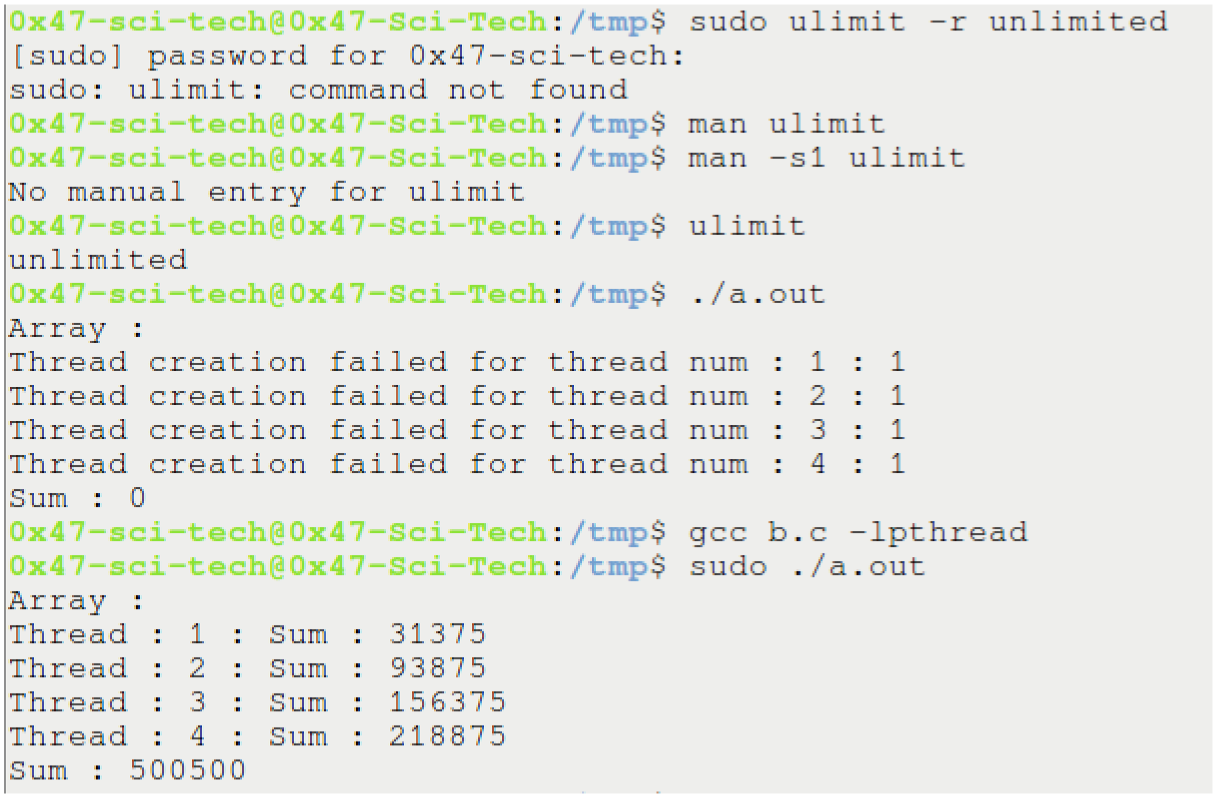
REMEMBER* : Linux doesn't provide thread scheduling within the PROCESS. In Linux we can schedule the like the process. So every thread will be considered as the separate process by kernel thread. I.e there is 1:1 mapping of user level and kernel level threads.

Example (ex-009):

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

#define NUM_THREADS 4
#define N 1000

pthread_mutex_t sum_mutex1;//Mutex for sum variable

int a[N], sum;

void initialize_array(int *a, int n) {
   int i;
   for (i = 1; i <= n; i++) {
      a[i-1] = i;
   }
}
void *thread_add(void *argument) {
   int i, thread_sum, arg;
   thread_sum = 0;
   arg = *(int *)argument;
   for (i = ((arg-1)*(N/NUM_THREADS)) ; i < (arg*(N/NUM_THREADS)); i++) {
      thread_sum += a[i];
   }
   printf("Thread : %d : Sum : %d\n", arg, thread_sum);
   pthread_mutex_lock(&sum_mutex1);
   sum += thread_sum;
   pthread_mutex_unlock(&sum_mutex1);
}

void print_array(int *a, int size) {
   int i;
   for(i = 0; i < size; i++)
      printf(" %d ", a[i]);
   putchar('\n');
}
int main() {
   int i, *range;
   pthread_t thread_id[NUM_THREADS];
//Scheduling attributes
   pthread_attr_t custom_sched_attr;
   int fifo_max_prio, fifo_min_prio, fifo_mid_prio;
   struct sched_param fifo_param;

//System initialization
   sum = 0;
   initialize_array(a, N);

//Attribute initialization
   pthread_attr_init(&custom_sched_attr);
   pthread_attr_setscope(&custom_sched_attr, PTHREAD_SCOPE_SYSTEM);
   pthread_attr_init(&custom_sched_attr);
   pthread_attr_setinheritsched(&custom_sched_attr, PTHREAD_EXPLICIT_SCHED);
   pthread_attr_setschedpolicy(&custom_sched_attr, SCHED_FIFO);

   fifo_max_prio = sched_get_priority_max(SCHED_FIFO);
   fifo_min_prio = sched_get_priority_min(SCHED_FIFO);
   fifo_mid_prio = (fifo_min_prio + fifo_max_prio)/2;
//  fifo_param.sched_priority = fifo_mid_prio;
   fifo_param.sched_priority = fifo_max_prio;
```

```
    pthread_attr_setschedparam(&custom_sched_attr, &fifo_param);

    printf("Array : \n");
    print_array(a, N);

    for (i = 0; i < NUM_THREADS; i++) {
        range = (int *)malloc(sizeof(int));
        *range = (i+1);
        if (pthread_create(&thread_id[i], &custom_sched_attr, thread_add, (void *)range))
            printf("Thread creation failed for thread num : %d\n", *range);
    }

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(thread_id[i], NULL);
    }

    printf("Sum : %d\n", sum);
    pthread_attr_destroy(&custom_sched_attr);
    return 0;
}
```

Output (ex-009):



```
0x47-sci-tech@0x47-Sci-Tech:/tmp$ sudo ulimit -r unlimited
[sudo] password for 0x47-sci-tech:
sudo: ulimit: command not found
0x47-sci-tech@0x47-Sci-Tech:/tmp$ man ulimit
0x47-sci-tech@0x47-Sci-Tech:/tmp$ man -s1 ulimit
No manual entry for ulimit
0x47-sci-tech@0x47-Sci-Tech:/tmp$ ulimit
unlimited
0x47-sci-tech@0x47-Sci-Tech:/tmp$ ./a.out
Array :
Thread creation failed for thread num : 1 : 1
Thread creation failed for thread num : 2 : 1
Thread creation failed for thread num : 3 : 1
Thread creation failed for thread num : 4 : 1
Sum : 0
0x47-sci-tech@0x47-Sci-Tech:/tmp$ gcc b.c -lpthread
0x47-sci-tech@0x47-Sci-Tech:/tmp$ sudo ./a.out
Array :
Thread : 1 : Sum : 31375
Thread : 2 : Sum : 93875
Thread : 3 : Sum : 156375
Thread : 4 : Sum : 218875
Sum : 500500
```

Explanation (ex-009):

Above code is same as (ex-004), only addition is that we have added the thread scheduling part into it. In which we are using the FIFO thread scheduling algorithm with maximum priority taken from system functions explicitly.

Remember* : To set certain attributes you requires the permission, Use sudo in those cases to run the code, refer man page of ulimit or visit http://stackoverflow.com/questions/9313428/getting-eperm-when-calling-pthread-create-for-sched-fifo-thread-as-root-on-lin

9 References

https://computing.llnl.gov/tutorials/pthreads/

http://man7.org/linux/man-pages/man3/

http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html