Creating a thread

```
// About pthread_create and its arguments

https://man7.org/linux/man-pages/man3/pthread_create.3.html

SYNOPSIS         top
       #include <pthread.h>

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

       Compile and link with -pthread.

 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.
```

- Four arguments to pthread_create

- Pointer to the thread
- Attributes to describe life cycle of the thread
- Function which the thread should execute
- Arguments to the previously mentioned function

Join threads

- Join – waiting until thread is done with its execution
- A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.
- The first argument to pthread_join() is the identifier of the thread to join. The second argument is a void pointer.
- pthread_join(pthread_t tid, void * return_value);
- If the return_value pointer is non-NULL, pthread_join will place at the memory location pointed to by return_value, the value passed by the thread tid through the pthread_exit call.
- Since we don't care about return value of the thread, we set it to NULL.

- Pthread_create()
- Pthread_join()

- `pthread_mutex_t mutex; pthread_cond_t cond;`

- pthread_mutex_init(&mutex, NULL); pthread_cond_init(&cond, NULL);
- `pthread_mutex_lock(&mutex);`
- `pthread_cond_wait(&cond, &mutex);`
- `pthread_cond_signal(&cond);pthread_cond_broadcast(&cond);`

Example

- Sending data and receiving data from threads – **quick digression**
  - Using memory allocation – first example
  - Using structs – second example

C pthreads-4.c > ...

```c
/*
Create a multi-threaded program which takes in which each thread takes an element from an input array
and generates first 5 multiples for the input it gets and print the result. Then each thread should return the
sum of first 5 multiples to the main function.
*/

// the result to be sent from each thread will be a reference to a pointer.

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

int numbers[5] = {2,3,5,6,7};

void * workerTask(void * arg){
    int pos = *(int *)arg;
    free(arg);
    int sum=0;
    printf("\nWe have for number %d at index [%d] first 5 multiples as :", numbers[pos], pos);
    for(int i=1; i<=5; i++){
        printf("%d ", numbers[pos]*i);
        sum = sum + numbers[pos]*i;
    }
    sleep(2);
    printf("\nSum for number %d at index [%d] is:%d", numbers[pos], pos, sum);
    int *result = malloc(sizeof(int));
    *result = sum;
    return (void *) result;
}
```

C pthreads-4.c > ...

```c
    }
    sleep(2);
    printf("\nSum for number %d at index [%d] is:%d", numbers[pos], pos, sum);
    int *result = malloc(sizeof(int));
    *result = sum;
    return (void *) result;
}

/*
If we would have passed the address of the variable index variable in order to comply with the pthread_create() function definition,
    it would have been problematic. Why? Because as each thread progresses, the value in the address of the index var will change
    leading to inconsistent outputs.

Changing the scope of the index variable would not help because inconsistency would still be there when reading value based on address
    because it will continue to change.

We only want to send in the value of the index when the thread is launched, then we do not care how that value changes.
*/

int main(int argc, char *argv[]){
    int *thread_output;
    pthread_t threads[5];

    for (int i=0;i<5;i++){
        //pthread_create(threads+i, NULL, workerTask, numbers+i);
        int * ptr_i = malloc(sizeof(int));
        *ptr_i = i;
        if (pthread_create(threads+i, NULL, workerTask, ptr_i) != 0){
            printf("There is some error in thread creation.");
            return 1;
```

```c
int main(int argc, char *argv[]){
    int *thread_output;
    pthread_t threads[5];

    for (int i=0;i<5;i++){
        //pthread_create(threads+i, NULL, workerTask, numbers+i);
        int * ptr_i = malloc(sizeof(int));
        *ptr_i = i;
        if (pthread_create(threads+i, NULL, workerTask, ptr_i) != 0){
            printf("There is some error in thread creation.");
            return 1;
        }
        // free(ptr_i) after its usage is complete. Here, usage of ptr_i is not complete because we have only launched threads here.
        // When threads are completed, then memory for ptr_i should be deallocated.
    }

    for (int i=0;i<5;i++){
        if( pthread_join(*(threads+i), (void **)&thread_output) != 0){
            return 2;
        }

        /*
        thread_output is supposed to be a double pointer. Reference to a pointer. Reference to a variable which stores reference to
            var having actual value.
        */
        printf("\nOutput from thread is %d", *thread_output);
        free(thread_output);
    }

    return 0;
}
```

Talk about

- Sending inputs to thread worker
  - Why cannot we send address of index?
  - Why we need to have memory allocated for sending value?
  - Second approach – directly send the address of elements in array

- Receiving outputs from the thread worker
  - Double pointer
  - Return value for worker function – pthread_exit value
  - Freeing up memory – where is it appropriate to do it?

Spawning threads based on command-line arguments

pthreads-5.c > main(int, char * [])

```c
/*
Calculate the sum of supplied set of numbers over multiple threads and output separate results.
We want each thread to calculate the sum of first 'n' positive integers given separately in input.
*/

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

struct sum_helper{
    long long calcTillNum;
    long long finalSum;
};

void * workerTask(void * args){
    struct sum_helper *ptr_sum_helper = (struct sum_helper *) args;
    long long sum = 0;
    for(long long i=0; i< ptr_sum_helper->calcTillNum+1; i++){
        sum = sum + i;
    }
    ptr_sum_helper->finalSum = sum;
    pthread_exit(NULL);
}

```

```c

int main(int argc, char *argv[]){

    // if less than 2 arguments given then something is missing in input
    if( argc < 2){
        printf("Missing value in input - give input as - executable <num1> <num2> <num3>");
        exit(-1);
    }

    int num_threads = argc-1;

    pthread_t threads[num_threads];
    struct sum_helper sumArgs[num_threads];

    for (int i = 0; i < num_threads; i++){
        // spawn a new thread for calculating sum till first n positive integers.

        sumArgs[i].calcTillNum = atoll(argv[i+1]);
        if( pthread_create(threads+i, NULL, workerTask, &sumArgs[i]) != 0){
            printf("There is an error in thread creation.");
            return 1;
        }
    }

    for (int i = 0; i < num_threads; i++){
        if (pthread_join(*(threads+i), NULL) != 0){
            return 2;
        }
        printf("The sum for %lld input is: %lld\n", sumArgs[i].calcTillNum, sumArgs[i].finalSum);
    }

    return 0;
}
```

- We don’t want to return pointer to a local variable that is on the function call stack because when the function goes out of scope, the memory will be deallocated at some point.

```
akapur12@lab2-15:~/c-lab-1$ ./fifth-example 1 2 3 4 10 50 10000 10000000 1000000000 1000000
The sum for 1 input is: 1
The sum for 2 input is: 3
The sum for 3 input is: 6
The sum for 4 input is: 10
The sum for 10 input is: 55
The sum for 50 input is: 1275
The sum for 10000 input is: 50005000
The sum for 10000000 input is: 50000005000000
The sum for 1000000000 input is: 5000000000500000000
The sum for 1000000 input is: 500000500000
```