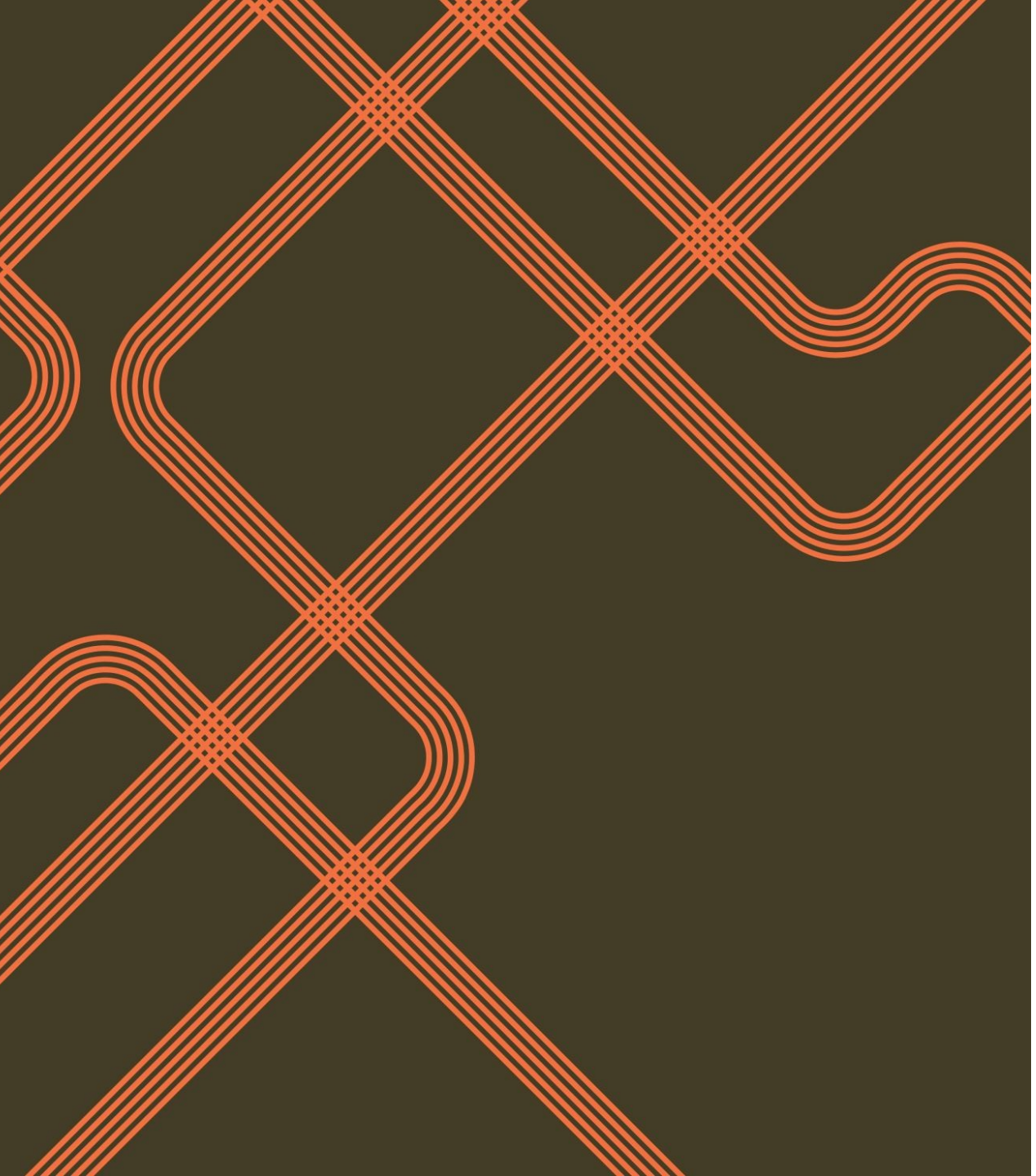


Pthreads I

Tutorials: **Mon 11:35am - 12:25pm, Rutherford Physics Building 112**

by Aayush Kapur

Intention behind multithreading

- Ability of a single core to quickly switch from doing certain operations (one thread to the next).
- Benefit is gaining flexibility in the order of operations.

Is there a parent or child thread?

- It is just semantics.
- Creator threads and spawned threads have the same properties in the eyes of the Pthreads.

Things to consider

- Attributes for the thread (controls how a thread will function)
  - Use NULL in pthread_create [keep it to default]

- Return type and exit status for threads:
  - Thread terminates explicitly with a pthread_exit, its argument becomes the exit status.
  - If no call to pthread_exit made, return value of the thread routine becomes its exit status.

- Local variables will go out of scope so other threads will not be able to access them.

How to run the program?

- To compile: gcc pthreads-abz.c -lpthread -o second-example
- To run: run the executable - ./name_of_the_executable (./second-example in this case)
- Why do we need to use –lpthread?
  - We want the linker to be able to find the symbols defined in the pthread library.

Example

- Consider a randomly initialized array consisting of positive integers.
- You need to collect all the even indexed elements on one thread and all the odd indexed elements on a separate thread.
- Make sure to go over the indexes in increasing order without skipping any.
- Lastly, gather sum of only odd integers in even indexed thread and gather sum of only even integers in the odd indexed thread.

An example:

Some random array like = 3 1 55 4 5 8 7

Thread 1 will have:

Even indexed elements of the array: 3, 55, 5, 7

collect sum of only odd integers from this - 3+55+5+7 = 70 is the output

thread 2

Odd indexed elements of the array: 1, 4, 8

collect sum of only even integers from this - 4+8 = 12 is the output

C odd-even-threads.c > ...

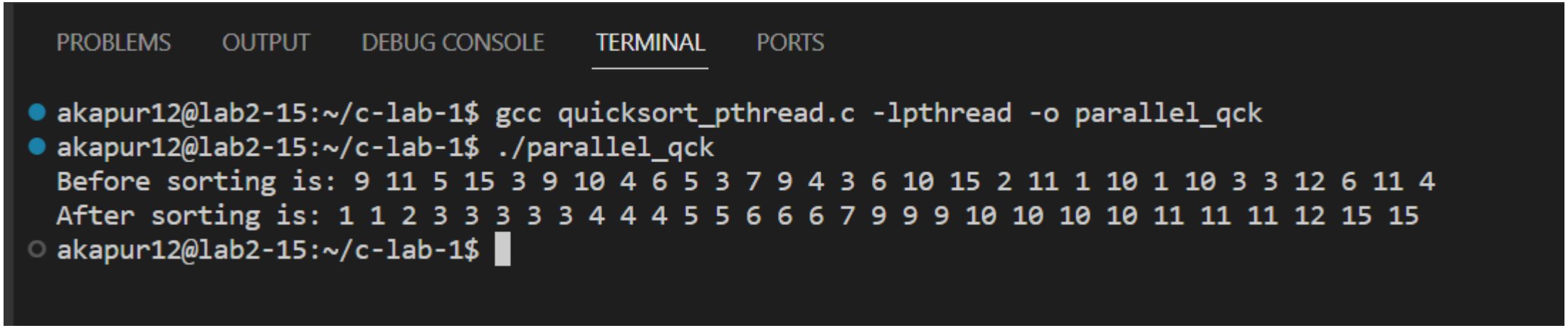
```c
    // int arr[7] = {3, 1, 55, 4, 5, 8, 7};

    int arr[SIZE];
    // initialise an array with random values.
    for (int i=0;i<SIZE;i++){
        arr[i] = 1+ rand() % (30+1-1);
    }

    struct tracker *output = malloc(sizeof(struct tracker));
    output->arr = malloc(sizeof(int)*SIZE);
    output->arr = arr;
    output->evenSum=0;
    output->oddSum=0;

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cond, NULL);
    pthread_t thread[2];

    pthread_create(&thread[0], NULL, evenWorker, output);
    pthread_create(&thread[1], NULL, oddWorker, output);

    void *result[2];

    pthread_join(thread[0], &result[0]);
    pthread_join(thread[1], &result[1]);

    printf("Output from thread 0 (evenWorker) is:%d\n", output->evenSum);
    printf("Output from thread 1 (oddWorker) is:%d\n", output->oddSum);

    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);
    return 0;
}
```

odd-even-threads.c > ...

```c
#define SIZE 20

pthread_mutex_t lock;
pthread_cond_t cond;
int pos = 0;

struct tracker{
    int * arr;
    int evenSum;
    int oddSum;
};

void * evenWorker(void * arg){
    struct tracker * output = arg;

    while(pos < SIZE){
        pthread_mutex_lock(&lock);
        if(pos%2==0){
            printf("Tid %ld even index %d element: %d\n", pthread_self(), pos, output->arr[pos]);
            if(output->arr[pos]%2 != 0){
                output->evenSum = output->evenSum + output->arr[pos];
            }
            pos++;
            pthread_cond_signal(&cond);
        }
        else{
            pthread_cond_wait(&cond, &lock);
        }
        pthread_mutex_unlock(&lock);
    }
    pthread_exit(NULL);
}

```

```c
void * oddWorker(void * arg){
    struct tracker * output = arg;

    while(pos < SIZE){
        pthread_mutex_lock(&lock);
        if(pos%2!=0){
            printf("Tid %ld odd index %d element: %d\n", pthread_self(), pos, output->arr[pos]);
            if(output->arr[pos]%2 == 0){
                output->oddSum = output->oddSum + output->arr[pos];
            }
            pos++;
            pthread_cond_signal(&cond);
        }
        else{
            pthread_cond_wait(&cond, &lock);
        }
        pthread_mutex_unlock(&lock);
    }
    pthread_exit(NULL);
}

```

- Example: make a multi threaded implementation for quicksort.



```
102
103 int main(int argc, char *argv[]){
104
105     srand((unsigned int) time(NULL));
106     int arr[SIZE];
107
108     // initialise an array with random values.
109     for (int i=0;i<SIZE;i++){
110         arr[i] = 1+ rand() % (15+1-1);
111     }
112
113     printf("Before sorting is: ");
114     for (int i=0;i<SIZE;i++){
115         printf("%d ", arr[i]);
116     }
117
118     // we will use a structure to pass the values to thread's functions.
119     struct argument *helper = malloc(sizeof(struct argument));
120     helper->arr = malloc(sizeof(int)*SIZE);
121     helper->arr = arr;
122     helper->lowIndex = 0;
123     helper->highIndex = SIZE-1;
124
125     quicksort(helper);
126
127     printf("\nAfter sorting is: ");
128     for (int i=0;i<SIZE;i++){
129         printf("%d ", arr[i]);
130     }
131
```

```c

void* quicksort(void * args){

    // create two pointers vars to structure because we will launch quicksort on one half in a new thread.
    // we will let this thread continue its execution for quicksort of other half -- to avoid wasting this thread, we
    // will not spawn a new thread for the other half.

    // assign the input pointer to helper_1 only, we will assign values to helper_2 based using helper_1 itself.
    struct argument *helper_1 = args;
    struct argument helper_2;

    int lowIndex = helper_1->lowIndex;
    int highIndex = helper_1->highIndex;
    int *arr = helper_1->arr;

    pthread_t thread_2;

    if (lowIndex >=  highIndex){
        return NULL;
    }

    //randomly assign the pivot
    //int pivotIndex = lowIndex + rand() % (highIndex + 1 - lowIndex);
    //int pivot = arr[pivotIndex];
    int pivot = arr[highIndex];

    //swap(arr, pivotIndex, highIndex);
    int left = partition(arr, lowIndex, highIndex, pivot);


```

quicksort_pthread.c

quicksort_pthread.c > quicksort(void *)

```c
    //randomly assign the pivot
    //int pivotIndex = lowIndex + rand() % (highIndex + 1 - lowIndex);
    //int pivot = arr[pivotIndex];
    int pivot = arr[highIndex];

    //swap(arr, pivotIndex, highIndex);
    int left = partition(arr, lowIndex, highIndex, pivot);


    // helper_1->lowIndex = lowIndex; To show that we want helper_1 to have its original lowIndex value
    // we are readying helper_1 as argument for quicksort call from lowIndex to left-1 on arr.
    helper_1->highIndex = left-1;
    //helper_1->arr = arr; To show that we want helper_1 to have the maintain its original reference to the array.


    // we are readying helper_2 as argument for quicksort call from left+1 to highIndex on arr.
    helper_2.lowIndex = left+1;
    helper_2.highIndex=highIndex;
    // we want helper_2 to take the reference of helper_1's arr so that effects of swaps by threads are visible in final answer.
    helper_2.arr = helper_1->arr;

```

quicksort_pthread.c ●

quicksort_pthread.c > quicksort(void *)

```
59      // spawn a new thread for half the computation of quicksort.
60      if (pthread_create(&thread_2, NULL, quicksort, &helper_2) != 0){
61          printf("There is an error while creating the thread.");
62      }
63
64      quicksort(helper_1);
65
66      if(pthread_join(thread_2, NULL) !=0){
67          printf("There is an error while joining the threads.");
68      }
69  }
70
71  int partition(int arr[], int lowIndex, int highIndex, int pivot){
72      int left = lowIndex;
73      int right = highIndex - 1;
74
75      while (left < right){
76          while(arr[left] <= pivot && left < right){
77              left++;}
78
79          while(arr[right] >= pivot && left < right){
80              right--;}
81          swap(arr, left, right);
82          }
83
84          if (arr[left] > arr[highIndex]){
85              swap(arr, left, highIndex);
86          }
87          else{
88              left = highIndex;
89          }
90          return left;
91  }
```

Talk about

- Launching threads and which function used as routine for pthread_create
- Sending input to threads
- Why two new threads not created?

Sources

- POSIX - https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html
- https://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/threads.htm
- https://www.youtube.com/watch?v=olYdb0DdGtM
- https://www.youtube.com/watch?v=qPhP86HIXgg

- https://en.wikipedia.org/wiki/Pthreads
- https://stackoverflow.com/questions/1780599/what-is-the-meaning-of-posix

- https://man7.org/linux/man-pages/man3/pthread_attr_init.3.html
- https://man7.org/linux/man-pages/man3/pthread_join.3.html
- https://man7.org/linux/man-pages/man3/pthread_exit.3.html
- http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html

- https://www.youtube.com/watch?v=ln3el6PR__Q&list=PLfqABt5AS4FmuQf70psXrsMLEDQXNkLq2&index=6
- https://www.youtube.com/watch?v=1ks-oMotUjc