Day3

Table of Contents

1. Agenda

- Reduction
- Returning from a function
- Barrier

- Error handling
- Busy waiting
- Conditional variables
- Detaching Threads

2. Reduction

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define N 5000000
#define T 16

pthread_mutex_t mutex;
long sum = 0;
int arr[N];

void *hello(void* threadId){
    long tid = (long)threadId;
    long localSum = 0;
    int chunk_size = N / T;
    int start = tid * chunk_size;
    int end = start + chunk_size;
    if (tid == T - 1) {
        end = N;
    }
    for (int i = start; i < end; i++) {
        localSum += (long)arr[i];
    }
    pthread_mutex_lock(&mutex);
    sum += localSum;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main(){
    for(int i = 0; i < N; i++){
        arr[i] = i + 1;
    }
    pthread_t* t;
    pthread_mutex_init(&mutex, NULL);
    t = malloc(sizeof(pthread_t) * T);
    for(long i = 0; i < T; i++)
        pthread_create(&t[i], NULL, hello, (void*)i);
    for(long i = 0; i < T; i++)
        pthread_join(t[i], NULL);
    pthread_mutex_destroy(&mutex);
    printf("Sum using manual reduction: %ld\n", sum);
    printf("Natural Number sum original: %ld\n", ((N * ((N + 1) * 1L)) / 2));
    free(t);
    return 0;
}
```

```
Sum using manual reduction: 12500002500000
Natural Number sum original: 12500002500000
```

3. Reduction Alter

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define N 5000000
#define T 16

long globalArraySum[T];
int arr[N];

void *hello(void* threadId){
    long tid = (long)threadId;
    long localSum = 0;
    int chunk_size = N / T;
    int start = tid * chunk_size;
    int end = (tid + 1) * chunk_size;
    if (tid == T - 1) {
        end = N;
    }
    for (int i = start; i < end; i++) {
        localSum += (long)arr[i];
    }
    globalArraySum[tid] = localSum;
    return NULL;
}

int main(){
```

```c
    for(int i = 0; i < N; i++){
        arr[i] = i + 1;
    }
    pthread_t* t;
    t = malloc(sizeof(pthread_t) * N);

    for(long i = 0; i < T; i++)
        pthread_create(&t[i], NULL, hello, (void*)i);
    for(long i = 0; i < T; i++)
        pthread_join(t[i], NULL);
    long sum = 0;
    for(int i = 0; i < T; i++){
        sum+= globalArraySum[i];
    }
    for(int i = 0; i < T; i++) printf("local sum of thread %d = %ld\n", i, globalArraySum[i]);
    printf("Sum using manual reduction: %ld\n", sum);
    printf("Natural Number sum original: %ld\n", ((N * ((N + 1) * 1L)) / 2));
    free(t);
    return 0;
}
```

```
local sum of thread 0 = 48828281250
local sum of thread 1 = 146484531250
local sum of thread 2 = 244140781250
local sum of thread 3 = 341797031250
local sum of thread 4 = 439453281250
local sum of thread 5 = 537109531250
local sum of thread 6 = 634765781250
local sum of thread 7 = 732422031250
local sum of thread 8 = 830078281250
local sum of thread 9 = 927734531250
local sum of thread 10 = 1025390781250
local sum of thread 11 = 1123047031250
local sum of thread 12 = 1220703281250
local sum of thread 13 = 1318359531250
local sum of thread 14 = 1416015781250
local sum of thread 15 = 1513672031250
Sum using manual reduction: 12500002500000
Natural Number sum original: 12500002500000
```

4. Return from function

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define N 5000000
#define T 16

int arr[N];

void *hello(void* threadId){
    long tid = (long)threadId;
    long localSum = 0;
    int chunk_size = N / T;
    int start = tid * chunk_size;
    int end = (tid + 1) * chunk_size;
    if (tid == T - 1) {
        end = N;
    }
    for (int i = start; i < end; i++) {
        localSum += (long)arr[i];
    }
    return (void*)localSum;
}

int main(){
    for(int i = 0; i < N; i++){
        arr[i] = i + 1;
    }
    pthread_t* t;
    t = malloc(sizeof(pthread_t) * N);

    long sum = 0, localSum;
    for(long i = 0; i < T; i++)
        pthread_create(&t[i], NULL, hello, (void*)i);
    for(long i = 0; i < T; i++){
        pthread_join(t[i], (void**)&localSum);
        sum+= *(long*)&localSum;
    }
    printf("Sum using manual reduction: %ld\n", sum);
    printf("Natural Number sum original: %ld\n", ((N * ((N + 1) * 1L)) / 2));
    free(t);
    return 0;
}
```

```
Sum using manual reduction: 12500002500000
```

```
Natural Number sum original: 12500002500000
```

5. Returning from function

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N 30000
#define T 4

int arr[N];

void *hello(void* threadId) {
    long tid = (long)threadId;
    long *localSum = malloc(sizeof(long)); // Allocate memory for the local sum
    *localSum = 0;
    int chunk_size = N / T;
    int start = tid * chunk_size;
    int end = (tid + 1) * chunk_size;

    // Ensure the last thread processes the remaining elements
    if (tid == T - 1) {
        end = N;
    }

    for (int i = start; i < end; i++) {
        *localSum += arr[i];
    }

    return (void*)localSum;
}

int main() {
    for (int i = 0; i < N; i++) {
        arr[i] = i + 1;
    }

    pthread_t threads[T];
    void *status;
    long sum = 0;

    // Create threads
    for (long i = 0; i < T; i++) {
        pthread_create(&threads[i], NULL, hello, (void*)i);
    }

    // Join threads and aggregate the local sums
    for (long i = 0; i < T; i++) {
        pthread_join(threads[i], &status);
        sum += *(long*)status;
        free(status); // Free the allocated memory for the local sum
    }

    printf("Sum using manual reduction: %ld\n", sum);
    printf("Natural Number sum original: %ld\n", ((N * 1L * (N + 1)) / 2));

    return 0;
}
```

```
Sum using manual reduction: 450015000
Natural Number sum original: 450015000
```

6. pthread_exit

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N 1000000
#define T 20

int arr[N];

void *hello(void* threadId) {
    long tid = (long)threadId;
    long localSum = 0; // Changed to long to match sum type
    int chunk_size = N / T;
    int start = tid * chunk_size;
    int end = (tid + 1) * chunk_size;

    // Ensure the last thread processes the remaining elements
    if (tid == T - 1) {
        end = N;
    }

    for (int i = start; i < end; i++) {
        localSum += arr[i];
    }

    pthread_exit((void*) localSum);
}

int main() {
    for (int i = 0; i < N; i++) {
        arr[i] = i + 1;
    }

    pthread_t threads[T];
    void *status;
    long sum = 0; // Changed to long to match localSum type

    // Create threads
    for (long i = 0; i < T; i++) {
        pthread_create(&threads[i], NULL, hello, (void*)i);
    }

    // Join threads and aggregate the local sums
    for (long i = 0; i < T; i++) {
        pthread_join(threads[i], &status);
        sum += (long)status;
    }

    printf("Sum using manual reduction: %ld\n", sum);
    printf("Natural Number sum original: %ld\n", ((N * 1L * (N + 1)) / 2));

    return 0;
}
```

```
Sum using manual reduction: 500000500000
Natural Number sum original: 500000500000
```

7. Conditional Variable

A conditional variable in Pthreads is a synchronization primitive that allows threads to wait until a certain condition is true. It is used to block a thread until another thread signals that the condition is met. Conditional variables are usually used in conjunction with a mutex to avoid race conditions.

- pthread_cond_wait: Releases the mutex and waits for the condition variable to be signaled.
- pthread_cond_signal: Wakes up one thread waiting on the condition variable.
- pthread_cond_broadcast: Wakes up all threads waiting on the condition variable.

7.1. Code

In this code we are trying to implement barrier using conditional variable.

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

#define N 3000000
#define T 16

int arr[N];
pthread_mutex_t mutex;
pthread_cond_t cond;
int data_ready = 0; // Condition to indicate if the data is ready

void *initialize_and_sum(void* threadId) {
    long tid = (long)threadId;
    long *localSum = malloc(sizeof(long)); // Allocate memory for the local sum
    *localSum = 0;
    int chunk_size = N / T;
    int start = tid * chunk_size;
    int end = (tid + 1) * chunk_size;

    // Ensure the last thread processes the remaining elements
    if (tid == T - 1) {
        end = N;
    }

    if (tid == 0) {
        // Thread 0 initializes the array
        for (int i = 0; i < N; i++) {
            arr[i] = i + 1;
        }

        // Signal all other threads that data is ready
        pthread_mutex_lock(&mutex);
        data_ready = 1;
        pthread_cond_broadcast(&cond);
        pthread_mutex_unlock(&mutex);
    } else {
        // Other threads wait until the data is initialized
        pthread_mutex_lock(&mutex);
        while (data_ready != 1) {
            pthread_cond_wait(&cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);
    }

    // Compute the local sum
    for (int i = start; i < end; i++) {
        *localSum += arr[i];
    }
    return (void*)localSum;
}

int main() {
    pthread_t threads[T];
    void *status;
    long sum = 0;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    // Create threads for initialization and summing
    for (long i = 0; i < T; i++) {
        pthread_create(&threads[i], NULL, initialize_and_sum, (void*)i);
    }

    // Join threads and aggregate the local sums
    for (long i = 0; i < T; i++) {
        pthread_join(threads[i], &status);
        sum += *(long*)status;
        free(status); // Free the allocated memory for the local sum
    }

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);

    printf("Sum using manual reduction: %ld\n", sum);
    printf("Natural Number sum original: %ld\n", ((N * 1L * (N + 1)) / 2));

    return 0;
}
```

```
Sum using manual reduction: 4500001500000
Natural Number sum original: 4500001500000
```

8. pthread_barrier

In this code only one thread (say 0) is allowed to create the whole data. After then we have to computer the result using all those available threads. Using barrier in this code will make sure that 0 will finish the data and go to the barrier then only all those threads will move to next line of the code. Means until 0 is doing the data every threads will have to wait for 0 to come to the barrier.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N 1000000
#define T 16

int arr[N];
pthread_barrier_t barrier;

void *hello(void* threadId) {
    long tid = (long)threadId;
    long *localSum = malloc(sizeof(long)); // Allocate memory for the local sum
    *localSum = 0;
    int chunk_size = N / T;
    int start = tid * chunk_size;
    int end = (tid + 1) * chunk_size;

    // Ensure the last thread processes the remaining elements
    if (tid == T - 1) {
        end = N;
    }

    // Initialize the chunk of the array
    if(tid == 0){
        for (int i = 0; i < N; i++) {
                arr[i] = i + 1;
        }
    }

    // Wait for all threads to finish initialization
    pthread_barrier_wait(&barrier);

    // Compute the local sum
    for (int i = start; i < end; i++) {
        *localSum += arr[i];
    }

    return (void*)localSum;
}

int main() {
    pthread_t threads[T];
    void *status;
    long sum = 0;

    // Initialize the barrier
    pthread_barrier_init(&barrier, NULL, T);

    // Create threads
    for (long i = 0; i < T; i++) {
        pthread_create(&threads[i], NULL, hello, (void*)i);
    }

    // Join threads and aggregate the local sums
    for (long i = 0; i < T; i++) {
        pthread_join(threads[i], &status);
        sum += *(long*)status;
        free(status); // Free the allocated memory for the local sum
    }

    // Destroy the barrier
    pthread_barrier_destroy(&barrier);

    printf("Sum using manual reduction: %ld\n", sum);
    printf("Natural Number sum original: %ld\n", ((N * 1L * (N + 1)) / 2));

    return 0;
}
```

```
Sum using manual reduction: 500000500000
Natural Number sum original: 500000500000
```

9. Detached thread

This program demonstrate detached threads. Here sometimes you'll find data is not fully initialized by detached threads which leads to segmentation fault. You can use sleep or wait there for some time to make sure the data is fully initialized.

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

#define N 10000  // Size of the array
#define T 4      // Number of threads

int arr[N];

void *init_array(void *arg) {
    int thread_id = *(int *)arg;
    int chunk_size = N / T;
    int start = thread_id * chunk_size;
    int end = (thread_id + 1) * chunk_size;

    if (thread_id == T - 1) {
        end = N;
    }

    for (int i = start; i < end; ++i) {
        arr[i] = i + 1;
    }

}

int main() {
    pthread_t threads[T];
    pthread_attr_t attr;
    int thread_args[T];

    // Initialize thread attributes
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    // Create detached threads to initialize array
    for (int i = 0; i < T; ++i) {
        thread_args[i] = i;
        pthread_create(&threads[i], &attr, init_array, (void *)&thread_args[i]);
    }

    // Destroy thread attributes
    pthread_attr_destroy(&attr);

    // Optional: Main thread can perform other tasks or wait
    // e.g., usleep(1000); // Wait for threads to complete if necessary

    printf("Array initialization using detached threads...\n");

    // Main thread continues execution
    // Print or use initialized array if needed

    // Example: Print a few initialized array elements
    printf("Initialized array elements:\n");
    for (int i = 0; i < 100; ++i) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    pthread_exit(0);

    return 0;
}
```

```
Array initialization using detached threads...
Initialized array elements:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 4
```

```
Array initialization using detached threads...
Initialized array elements:
0 0 0 0 0 0 0 0 0 0
```

```
Array initialization using detached threads...
Initialized array elements:
1 2 3 4 5 6 7 8 9 10
```

```
Array initialization using detached threads...
Initialized array elements:
1 2 3 4 5 6 7 8 9 10
```

10. Addition of two array

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define N 10000
#define T 20

int *arr1;
int *arr2;
int *arr3;

void *hello(void* threadId) {
    long tid = (long)threadId;
    long localSum = 0; // Changed to long to match sum type
    int chunk_size = N / T;
    int start = tid * chunk_size;
    int end = (tid + 1) * chunk_size;

    // Ensure the last thread processes the remaining elements
    if (tid == T - 1) {
        end = N;
    }

    for (int i = start; i < end; i++) {
        arr3[i] = arr1[i] + arr2[i];
    }

    return NULL;
}

int main() {
    arr1 = (int*)malloc(sizeof(int) * N);
    arr2 = (int*)malloc(sizeof(int) * N);
    arr3 = (int*)malloc(sizeof(int) * N);
    for (int i = 0; i < N; i++) {
        arr1[i] = i + 1;
        arr2[i] = i + 1;
        arr3[i] = 0;
    }

    pthread_t threads[T];

    // Create threads
    for (long i = 0; i < T; i++) {
        pthread_create(&threads[i], NULL, hello, (void*)i);
    }

    // Join threads and aggregate the local sums
    for (long i = 0; i < T; i++) {
        pthread_join(threads[i], NULL);
    }

    for(int i = 0; i < N; i++){
        printf("%d ",arr3[i]);
    }

    return 0;
}
```

```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92
```

```
gcc twoArraySum.c -lpthread
```

```
./a.out > output.txt
echo "Check output.txt"
```

```
Check output.txt
```

11. Assignments: PThreads

11.1. Create a serial matrix addition code and parallelize it using pthreads.

11.2. Create a serial matrix addition code and parallelize it using pthreads.

11.3. Create a prime number calculator.

- Your code should calculate the numbers of prime between 0 and N.

- Serial code is available on github. You can copy and parallelize it.

12. Problem

Create an array of size N. Initialialize that array with some elements. You have to create T number of threads and then you have to devide the data between those number of threads. After division you have to calculate sum of all the elements of the array by those threads.

13. Return from function demo2

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define T 7

void* hello(){
    long* localValue = malloc(sizeof(long));
    *localValue = 123;
    return (void*)*localValue;
}

int main(){
    pthread_t* t;
    t = malloc(sizeof(pthread_t) * T);

    long localValue;
    for(long i = 0; i < T; i++)
        pthread_create(&t[i], NULL, hello, NULL);
    for(long i = 0; i < T; i++){
        pthread_join(t[i], (void**)&localValue);
        printf("Local Value from each thread = %ld\n", localValue);
    }
    free(t);
    return 0;
}
```

```
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
```

14. Return from function demo3

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define T 7

void* hello(){
    int* a;
    a = (int*) malloc(sizeof(int));
    *a = 5;
    return (void*)a;
}

int main(){
    pthread_t* t;
    t = malloc(sizeof(pthread_t) * T);

    int* b;
    for(long i = 0; i < T; i++)
        pthread_create(&t[i], NULL, hello, NULL);
    for(long i = 0; i < T; i++){
        pthread_join(t[i], (void**)&b);
        printf("Local Value from each thread = %d\n", *b);
    }
    free(t);
    return 0;
}
```

```
Local Value from each thread = 5
Local Value from each thread = 5
Local Value from each thread = 5
Local Value from each thread = 5
Local Value from each thread = 5
Local Value from each thread = 5
Local Value from each thread = 5
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
Local Value from each thread = 123
```

15. Serial array addition code

```c
#include<stdio.h>
#include<stdlib.h>
#define N 100000
int main(){
    int* a, *b, *c;
    a = (int*) malloc(sizeof(int) * N);
    b = (int*) malloc(sizeof(int) * N);
    c = (int*) malloc(sizeof(int) * N);
    for(int i = 0; i < N; i++){
        a[i] = i + 1;
        b[i] = i + 1;
        c[i] = 0;
    }

    for(int i = 0; i < N; i++){
        c[i] = a[i] + b[i];
    }
    for(int i = 0; i < N; i++){
        printf("%d ", c[i]);
    }
    return 0;
}
```

```
gcc serialArrayAddition.c -o serialArrayAddition.out
```

```
./serialArrayAddition.out > output.txt
echo "check output.txt"
```

```
check output.txt
```

16. Parallel array addition code

```c
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#define N 100000
#define T 10

int* a, *b, *c;

void* task(void* threadId){
    int tid = *(int*)threadId;
    int chunksize = N / T;
    int start = tid * chunksize;
    int end = start + chunksize;
    if(tid == T - 1) end = N;

    for(int i = start; i < end; i++){
        c[i] = a[i] + b[i];
    }
}

int main(){
    a = (int*) malloc(sizeof(int) * N);
    b = (int*) malloc(sizeof(int) * N);
    c = (int*) malloc(sizeof(int) * N);
    for(int i = 0; i < N; i++){
        a[i] = i + 1;
        b[i] = i + 1;
        c[i] = 0;
    }

    pthread_t t[T];
    for(int i = 0; i < T; i++){
        int* a = (int*) malloc(sizeof(int));
        *a = i;
        pthread_create(&t[i], NULL, task, (void*)a);
    }

    for(int i = 0; i < T; i++){
        pthread_join(t[i], NULL);
    }

    for(int i = 0; i < N; i++){
        printf("%d ", c[i]);
    }
    return 0;
}
```

```
gcc parallelArrayAddition.c -o parallelArrayAddition.out -lpthread
```

```
./parallelArrayAddition.out > output_parallel.txt
echo "check output_parallel.txt"
```

```
check output_parallel.txt
```

17. Parallel data creation and computation

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#define N 100000
#define T 10

int* a, *b, *c;

void* task(void* threadId){
    int tid = *(int*)threadId;
    int chunksize = N / T;
    int start = tid * chunksize;
    int end = start + chunksize;
    if(tid == T - 1) end = N;

    for(int i = start; i < end; i++){
        a[i] = i + 1;
        b[i] = i + 1;
        c[i] = 0;
    }
    for(int i = start; i < end; i++){
        c[i] = a[i] + b[i];
    }
}

int main(){
    a = (int*) malloc(sizeof(int) * N);
    b = (int*) malloc(sizeof(int) * N);
    c = (int*) malloc(sizeof(int) * N);

    pthread_t t[T];
    for(int i = 0; i < T; i++){
        int* a = (int*) malloc(sizeof(int));
        *a = i;
        pthread_create(&t[i], NULL, task, (void*)a);
    }

    for(int i = 0; i < T; i++){
        pthread_join(t[i], NULL);
    }

    for(int i = 0; i < N; i++){
        printf("%d ", c[i]);
    }
    return 0;
}
```

```
gcc parallelDataWithComputation.c -lpthread
```

```
./a.out > output1.txt
echo "check output1.txt"
```

```
check output1.txt
```

18. Data creation by a particular thread

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#define N 100000
```

```c
#define T 10

int* a, *b, *c;
pthread_barrier_t barrier;

void* task(void* threadId){
    int tid = *(int*)threadId;
    int chunksize = N / T;
    int start = tid * chunksize;
    int end = start + chunksize;
    if(tid == T - 1) end = N;

    if(tid == 0){
        for(int i = 0; i < N; i++){
                a[i] = i + 1;
                b[i] = i + 1;
                c[i] = 0;
        }
    }

    pthread_barrier_wait(&barrier);

    for(int i = start; i < end; i++){
        c[i] = a[i] + b[i];
    }
}

int main(){
    a = (int*) malloc(sizeof(int) * N);
    b = (int*) malloc(sizeof(int) * N);
    c = (int*) malloc(sizeof(int) * N);

    pthread_t t[T];
    pthread_barrier_init(&barrier, NULL, T);
    for(int i = 0; i < T; i++){
        int* a = (int*) malloc(sizeof(int));
        *a = i;
        pthread_create(&t[i], NULL, task, (void*)a);
    }

    for(int i = 0; i < T; i++){
        pthread_join(t[i], NULL);
    }

    pthread_barrier_destroy(&barrier);
    for(int i = 0; i < N; i++){
        printf("%d ", c[i]);
    }
    return 0;
}
```

```
gcc task9.c -lpthread
```

```
./a.out > output2.txt
echo "check output2.txt"
```

```
check output2.txt
```

Author: Abhishek Raj

Created: 2024-12-18 Wed 17:06