# TASK 1:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int n, m, p;

int MAT\_A[10][10], MAT\_B[10][10], MAT\_C[10][10];

pthread\_t thread[10];

void\* multiply(void\* arg)

{

int row = \*(int\*) arg;

for (int i = 0; i < p; i++)

{

MAT\_C[row][i] = 0;

for (int j = 0; j < m; j++)

{

MAT\_C[row][i] += MAT\_A[row][j] \* MAT\_B[j][i];

}

}

pthread\_exit(NULL);

}

int main()

{

int i, j;

printf("Enter size of matrix A (less than 10, n x m): ");

scanf("%d %d", &n, &m);

printf("Enter size of matrix B (less than 10, m x k): ");

scanf("%d %d", &m, &p);

printf("Enter values for matrix A:\n");

for (int i = 0; i < n; i++)

{

for (int j = 0; j < m; j++)

{

scanf("%d", &MAT\_A[i][j]);

}

}

printf("Enter values for matrix B:\n");

for (int i = 0; i < m; i++)

{

for (int j = 0; j < p; j++)

{

scanf("%d", &MAT\_B[i][j]);

}

}

for (int i = 0; i < n; i++)

{

pthread\_create(&thread[i], NULL, multiply, (void\*) &i);

}

for (int i = 0; i < n; i++)

{

pthread\_join(thread[i], NULL);

}

printf("\nResultant matrix C:\n");

for (int i = 0; i < n; i++)

{

for (int j = 0; j < p; j++)

{

printf("%d ", MAT\_C[i][j]);

}

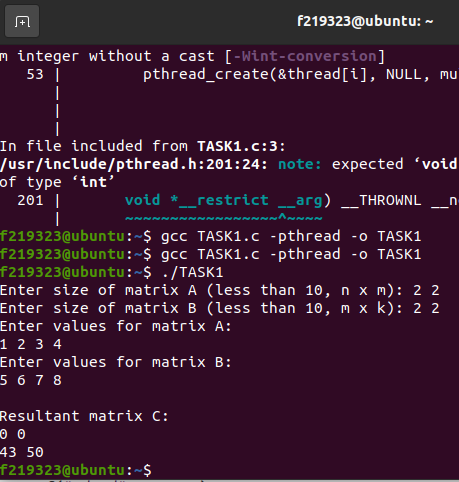
printf("\n");

}

return 0;

}

# OUTPUT:



TASK 3:  
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

struct Node

{

int data;

struct Node\* next;

};

struct Queue

{

struct Node\* front;

struct Node\* rear;

};

int globalValue;

struct Queue\* createQueue()

{

struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue));

q->front = q->rear = NULL;

return q;

}

void enqueue(struct Queue\* q, int value)

{

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = NULL;

if (q->rear == NULL)

{

q->front = q->rear = newNode;

return;

}

q->rear->next = newNode;

q->rear = newNode;

}

int dequeue(struct Queue\* q)

{

if (q->front == NULL)

{

return -1;

}

int value = q->front->data;

struct Node\* temp = q->front;

q->front = q->front->next;

if (q->front == NULL)

{

q->rear = NULL;

}

free(temp);

return value;

}

void\* readIntegers()

{

FILE\* obj = fopen("input.txt", "r");

int value;

while (fscanf(obj, "%d", &value) != EOF)

{

globalValue = value;

}

fclose(obj);

pthread\_exit(NULL);

}

void\* insertIntoQueue(void\* arg)

{

struct Queue\* q = (struct Queue\*)arg;

while (globalValue != -1)

{

if (globalValue != 0)

{

enqueue(q, globalValue);

globalValue = 0;

}

}

pthread\_exit(NULL);

}

int main()

{

struct Queue\* q = createQueue();

pthread\_t readThread, insertThread;

pthread\_create(&readThread, NULL, readIntegers, NULL);

pthread\_create(&insertThread, NULL, insertIntoQueue, q);

pthread\_join(readThread, NULL);

pthread\_join(insertThread, NULL);

struct Node\* temp = q->front;

while (temp != NULL)

{

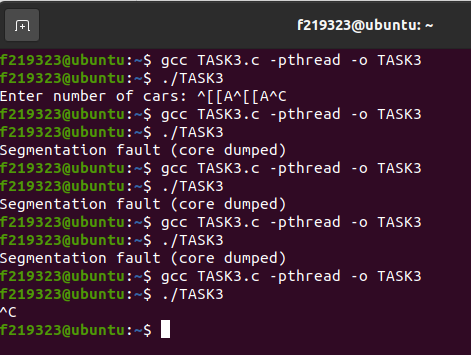
printf("%d ", temp->data);

temp = temp->next;

}

return 0;

}

OUTPUT:  


TASK 4:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <time.h>

#define ARRAY\_SIZE 1000000

#define CHUNK\_SIZE 250000

void generate\_integers(int\* array, int size)

{

srand(time(NULL));

for (int i = 0; i < size; i++)

{

array[i] = rand() % 10000;

}

}

void print\_array\_to\_file(int\* array, int size, char\* filename)

{

FILE\* obj = fopen(filename, "w");

for (int i = 0; i < size; i++)

{

fprintf(obj, "%d\n", array[i]);

}

fclose(obj);

}

void read\_array\_from\_file(int\* array, int size, char\* filename)

{

FILE\* obj = fopen(filename, "r");

for (int i = 0; i < size; i++)

{

fscanf(obj, "%d", &array[i]);

}

fclose(obj);

}

void print\_array(int\* array, int size)

{

for (int i = 0; i < size; i++)

{

printf("%d ", array[i]);

}

printf("\n");

}

void merge(int\* left, int left\_size, int\* right, int right\_size, int\* result)

{

int i = 0, j = 0, k = 0;

while (i < left\_size && j < right\_size)

{

if (left[i] <= right[j])

{

result[k++] = left[i++];

}

else

{

result[k++] = right[j++];

}

}

while (i < left\_size)

{

result[k++] = left[i++];

}

while (j < right\_size)

{

result[k++] = right[j++];

}

}

void merge\_sort\_rec(int\* array, int\* temp, int left, int right) ;

void\* merge\_sort(void\* arg)

{

int\* array = (int\*)arg;

int size = CHUNK\_SIZE;

int temp[size];

clock\_t start = clock();

merge\_sort\_rec(array, temp, 0, size-1);

clock\_t end = clock();

double time\_taken = ((double)(end - start))/CLOCKS\_PER\_SEC;

printf("Merge Sort Thread %lu finished in %lf seconds\n", pthread\_self(), time\_taken);

pthread\_exit(NULL);

}

void merge\_sort\_rec(int\* array, int\* temp, int left, int right)

{

if (left < right) {

int mid = (left + right) / 2;

merge\_sort\_rec(array, temp, left, mid);

merge\_sort\_rec(array, temp, mid+1, right);

merge(array+left, mid-left+1, array+mid+1, right-mid, temp+left);

for (int i = left; i <= right; i++)

{

array[i] = temp[i];

}

}

}

void selection\_sort(int\* array, int size)

{

for (int i = 0; i < size-1; i++)

{

for (int j = i+1; j < size; j++)

{

if (array[i] > array[j])

{

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

}

}

int main()

{

int array[ARRAY\_SIZE];

generate\_integers(array, ARRAY\_SIZE);

print\_array\_to\_file(array, ARRAY\_SIZE, "array.txt");

pthread\_t threads[4];

for (int i = 0; i < 4; i++)

{

int\* chunk = &array[i\*CHUNK\_SIZE];

pthread\_create(&threads[i], NULL, merge\_sort, chunk);

}

for (int i = 0; i < 4; i++)

{

pthread\_join(threads[i], NULL);

}

int temp[ARRAY\_SIZE];

for (int i = 0; i < ARRAY\_SIZE; i += 2\*CHUNK\_SIZE)

{

int left\_chunk\_start = i;

int left\_chunk\_end = i + CHUNK\_SIZE - 1;

int right\_chunk\_start = i + CHUNK\_SIZE;

int right\_chunk\_end = i + 2\*CHUNK\_SIZE - 1;

if (right\_chunk\_end >= ARRAY\_SIZE)

{

right\_chunk\_end = ARRAY\_SIZE-1;

}

merge(array+left\_chunk\_start, CHUNK\_SIZE, array+right\_chunk\_start, right\_chunk\_end-right\_chunk\_start+1, temp+left\_chunk\_start);

}

merge(array, 2\*CHUNK\_SIZE, array+2\*CHUNK\_SIZE, 2\*CHUNK\_SIZE, temp);

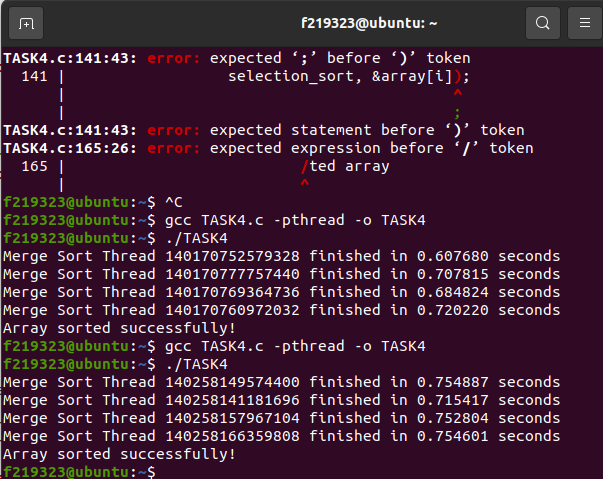
print\_array\_to\_file(temp, ARRAY\_SIZE, "sorted\_array.txt");

printf("Array sorted successfully!\n");

return 0;

}

OUTPUT:



TASK 5:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include <sys/wait.h>

//READ\_END 0

//WRITE\_END 1

int main()

{

int arr[10];

int i, left\_max, right\_max, max;

int left\_pipe[2], right\_pipe[2];

pid\_t left\_pid, right\_pid;

srand(time(NULL));

for (i = 0; i < 10; i++)

{

arr[i] = rand() % 1000;

}

int left[5], right[5];

for (i = 0; i < 5; i++)

{

left[i] = arr[i];

right[i] = arr[5 + i];

}

pipe(left\_pipe);

pipe(right\_pipe);

left\_pid = fork();

if (left\_pid == -1)

{

printf("Error: failed to fork\n");

exit(EXIT\_FAILURE);

}

else if (left\_pid == 0)

{

close(left\_pipe[0]);

left\_max = left[0];

for (i = 1; i < 5; i++)

{

if (left[i] > left\_max)

{

left\_max = left[i];

}

}

write(left\_pipe[1], &left\_max, sizeof(int));

close(left\_pipe[1]);

exit(EXIT\_SUCCESS);

}

else

{

right\_pid = fork();

if (right\_pid == -1)

{

printf("Error: failed to fork\n");

exit(EXIT\_FAILURE);

}

else if (right\_pid == 0)

{

close(right\_pipe[0]);

right\_max = right[0];

for (i = 1; i < 5; i++)

{

if (right[i] > right\_max)

{

right\_max = right[i];

}

}

write(right\_pipe[1], &right\_max, sizeof(int));

close(right\_pipe[1]);

exit(EXIT\_SUCCESS);

}

else

{

wait(NULL);

wait(NULL);

close(left\_pipe[1]);

close(right\_pipe[1]);

read(left\_pipe[0], &left\_max, sizeof(int));

read(right\_pipe[0], &right\_max, sizeof(int));

close(left\_pipe[0]);

close(right\_pipe[0]);

if (left\_max > right\_max)

{

max = left\_max;

}

else

{

max = right\_max;

}

printf("Max value: %d\n", max);

}

}

return 0;

}

# OUTPUT:

