GUC
German University in Cairo
الجامعة الألمانية بالقاهرة

Operating System Project
Milestone 1 Report

Team Members

Daniel Samer Morris 58-4182

Karl George 58-11295

Youssef Tamer 58-5378

Mohamed tamer58-5384

Mohamed Elsayed Hanafy 58-16392

Samuel Atef 58-4285

Objectives

The aim of the project is to run multiple threads on a single core each executing a function over two scheduling algorithms: First in first out and round robin scheduling algorithm.

We are going to be measuring the Execution time, release time (inputs at the beginning of the schedulers), start time, finish time, waiting time, response time, turnaround time, CPU useful work, CPU Utilization, memory consumption for both scheduling algorithms for each thread.

Thread 1: A function that takes two input alphabet characters through scanf() and prints characters in between.

Thread2: Outputs three messages.

Thread 3: Takes two integers and outputs the sum, product and of all integers in between.

Performance Analysis

Start Time

Start time is the moment when a process begins execution on the CPU

Finish Time

Finish time indicates when a process completes its execution. It is crucial for calculating turnaround time and evaluating overall system performance.

Execution time

Execution time is the total time taken by the scheduling algorithm to process all jobs from the input queue until completion. It includes the overhead of decision-making and context switching. The lower the execution time, the more efficient the algorithm

$$Execution\ Time = Finish\ Time - Start\ Time$$

Waiting time

Waiting time is the total time a process spends in the ready queue before getting CPU time. A lower waiting time leads to better performance and user experience.

$$Waiting\ Time = Turnaround\ Time - Execution\ Time$$

Release time

Release time refers to the moment when a process is made available for scheduling. It is an important metric that affects the scheduling decisions, particularly in preemptive and real-time scheduling algorithms.

Turnaround time

Turnaround time is the total time taken for a process to complete execution from the time it was submitted.

$$Turnaround\ Time = Finish\ Time - Release\ Time$$

CPU Utilization (per thread)

CPU utilization represents the percentage of time the CPU is actively executing processes. High CPU utilization is desirable as it ensures maximum efficiency.

$$CPU\ Utilization = Execution\ time\ /\ (execution\ time + waiting\ time)$$

Response

Response time is the time when a process is submitted to when it starts execution for the first time. It is a key metric in interactive systems where quick feedback is required.

$$Response\ time = Start\ time - release\ time$$

Results

FIFO:

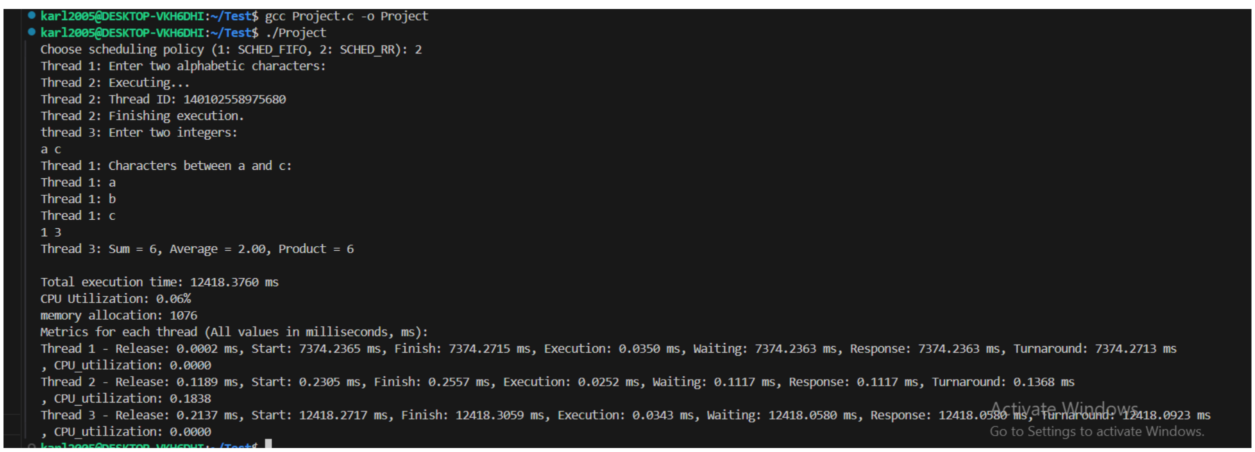
FIFO is a non-preemptive scheduling algorithm. Once a process starts execution, it runs until completion without being interrupted.so the order of execution should be thread 1 then thread 2 then thread 3.So the printf("Thread 1: Enter two alphabetic characters: \n"); in thread1 got executed then we encountered a scanf() which blocks thread1 until the input in thread1 is received so we go to thread2 and execute these print statements: printf("Thread 2: Executing...\n");printf("Thread 2: Thread ID: %lu\n", id);printf("Thread 2: Finishing execution.\n");. Then we go to thread 3 and execute the first print statement: printf("thread 3: Enter two integers:\n");.then the next statement in thread 3 is a scanf() so we block thread 3 till its input is received. Then we go back to thread 1 to take the input, then we continue the flow of thread1 then we go to thread 3 to take its input and continue the flow of thread 3 then we print the values of time metrices.

```
karl2005@DESKTOP-VKH6DHI:~/Test$ gcc Project.c -o Project
karl2005@DESKTOP-VKH6DHI:~/Test$ ./Project
Choose scheduling policy (1: SCHED_FIFO, 2: SCHED_RR): 1
Thread 1: Enter two alphabetic characters:
Thread 2: Executing...
Thread 2: Thread ID: 139700050986688
Thread 2: Finishing execution.
thread 3: Enter two integers:
a c
Thread 1: Characters between a and c:
Thread 1: a
Thread 1: b
Thread 1: c
1 3
Thread 3: Sum = 6, Average = 2.00, Product = 6

Total execution time: 13210.2203 ms
CPU Utilization: 0.07%
memory allocation: 1100
Metrics for each thread (All values in milliseconds, ms):
Thread 1 - Release: 0.0001 ms, Start: 8280.9289 ms, Finish: 8280.9751 ms, Execution: 0.0462 ms, Waiting: 8280.9288 ms, Response: 8280.9288 ms, Turnaround: 8280.9749 ms
, CPU_utilization: 0.0000
Thread 2 - Release: 0.1107 ms, Start: 0.2200 ms, Finish: 0.2481 ms, Execution: 0.0281 ms, Waiting: 0.1092 ms, Response: 0.1092 ms, Turnaround: 0.1373 ms
, CPU_utilization: 0.2045
Thread 3 - Release: 0.1984 ms, Start: 13210.1415 ms, Finish: 13210.1556 ms, Execution: 0.0142 ms, Waiting: 13209.9431 ms, Response: 13209.9431 ms, Turnaround: 13209.9573 ms
, CPU_utilization: 0.0000
```

Round Robin:

RR is preemptive, so each thread gets a fixed time slice (quantum). If a thread performs I/O within its time slice, it gets blocked, and the CPU moves to the next thread in the queue. Once the I/O operation is complete, the thread re-enters the ready queue and waits for its turn again.so at the beginning thread 1,2 and 3 are created and they are put in the ready queue. Now it’s the turn of thread 1 so we execute the 1rst print statement then the next statement is a scanf() so the thread gets blocked and then we move to thread 2 to execute the 3 print statements and now thread 2 is executed completely so it does not re-enter the ready queue then we go to the next thread in the queue which is thread 3 we get it out of the queue and execute the 1rst print statement and then we have a scanf() so thread 3 gets blocked. Now we return back to thread 1 and take the input then thread 1 re-enters the ready queue and it’s the 1rst in the queue so that’s its turn to get executed so we execute the rest of the thread and print the characters and then we take the input of thread 3 and then it re-enters the queue and now that’s its turn to get executed so we execute the rest of the thread and print the sum average and product. Then after we finish the 3 threads we print the time metricies. The results look similar to FIFO because the quantum was too large for threads 1 and 2 and 3 so each threads was able to completely finish execution in the fixed time slice available for it unless an IO operation happens.



```
karl2005@DESKTOP-VKH6DHI:~/Test$ gcc Project.c -o Project
karl2005@DESKTOP-VKH6DHI:~/Test$ ./Project
Choose scheduling policy (1: SCHED_FIFO, 2: SCHED_RR): 2
Thread 1: Enter two alphabetic characters:
Thread 2: Executing...
Thread 2: Thread ID: 140102558975680
Thread 2: Finishing execution.
thread 3: Enter two integers:
a c
Thread 1: Characters between a and c:
Thread 1: a
Thread 1: b
Thread 1: c
1 3
Thread 3: Sum = 6, Average = 2.00, Product = 6

Total execution time: 12418.3760 ms
CPU Utilization: 0.06%
memory allocation: 1076
Metrics for each thread (All values in milliseconds, ms):
Thread 1 - Release: 0.0002 ms, Start: 7374.2365 ms, Finish: 7374.2715 ms, Execution: 0.0350 ms, Waiting: 7374.2363 ms, Response: 7374.2363 ms, Turnaround: 7374.2713 ms
, CPU_utilization: 0.0000
Thread 2 - Release: 0.1189 ms, Start: 0.2305 ms, Finish: 0.2557 ms, Execution: 0.0252 ms, Waiting: 0.1117 ms, Response: 0.1117 ms, Turnaround: 0.1368 ms
, CPU_utilization: 0.1838
Thread 3 - Release: 0.2137 ms, Start: 12418.2717 ms, Finish: 12418.3059 ms, Execution: 0.0343 ms, Waiting: 12418.0580 ms, Response: 12418.0580 ms, Turnaround: 12418.0923 ms
, CPU_utilization: 0.0000
```

Code Explanation

milestone1.c > _GNU_SOURCE

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

// Function to get the current time in milliseconds
double get_time_ms() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000.0 + ts.tv_nsec / 1.0e6; // Convert to milliseconds
}

struct thread_metrics {
    double release_time;
    double start_time;
    double finish_time;
    double execution_time;
    double waiting_time;
    double response_time;
    double turnaround_time;
    double utilization_time;
};

struct thread_metrics t1_metrics, t2_metrics, t3_metrics;

// Struct to hold input data
typedef struct {
    char ch1, ch2;
    int num1, num2;
} InputData;

InputData input_data;
```

The first parts included the libraries we used.

Then we created a function that gets the current time and converted it to ms.

Then we defined structure and included the metrics that we are going to need.

Then we created three instances for the three threads we are going to use.

```c
void *thread1_func(void *arg) {
  printf("Thread 1: Enter two alphabetic characters: \n");
  scanf(" %c %c", &input_data.ch1, &input_data.ch2);

    t1_metrics.start_time = get_time_ms();

    char ch1 = input_data.ch1, ch2 = input_data.ch2;
    if (ch1 > ch2) {
        char temp = ch1;
        ch1 = ch2;
        ch2 = temp;
    }


    printf("Thread 1: Characters between %c and %c: \n", ch1, ch2);
    for (char c = ch1; c <= ch2; c++) {
        printf("Thread 1: %c \n", c);

    }


    t1_metrics.finish_time = get_time_ms();
    return NULL;
}

void *thread2_func(void *arg) {
    t2_metrics.start_time = get_time_ms();

    pthread_t id = pthread_self();
    printf("Thread 2: Executing...\n");
    printf("Thread 2: Thread ID: %lu\n", id);
    printf("Thread 2: Finishing execution.\n");

    t2_metrics.finish_time = get_time_ms();
    return NULL;
}

void *thread3_func(void *arg) {
  printf("thread 3: Enter two integers:\n");
  scanf("%d %d", &input_data.num1, &input_data.num2);
    t3_metrics.start_time = get_time_ms();

    int num1 = input_data.num1, num2 = input_data.num2;
    if (num1 > num2) {
        int temp = num1;
        num1 = num2;
        num2 = temp;
    }

    int sum = 0, count = 0;
    long long product = 1; // Use long long to prevent overflow

    for (int i = num1; i <= num2; i++) {
        sum += i;
        product *= i;
        count++;
    }
    float avg = (float)sum / count;

    printf("Thread 3: Sum = %d, Average = %.2f, Product = %lld\n", sum, avg, product);

    t3_metrics.finish_time = get_time_ms();
    return NULL;
}
```

This part contains the functions that each thread will perform.

```c
//memory allocation
long get_memory_usage() {
    FILE *file = fopen("/proc/self/status", "r");
    if (!file) {
        perror("Failed to open memory status file");
        return -1;
    }

    char line[256];
    long memory_kb = -1;

    while (fgets(line, sizeof(line), file)) {
        if (strncmp(line, "VmRSS:", 6) == 0) {
            sscanf(line, "VmRSS: %ld kB", &memory_kb);
            break;
        }
    }

    fclose(file);
    return memory_kb;
}
```

This part contains the function that calulate the memory usage in KB.

```c
int main() {
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(0, &cpuset); // Bind to core 0

    pthread_t t1, t2, t3;
    struct sched_param param;
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    //memory allocation
    long memoryAllocation=get_memory_usage();


    // Set thread CPU affinity
    pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpuset);

    int policy;
    printf("Choose scheduling policy (1: SCHED_FIFO, 2: SCHED_RR): ");
    scanf("%d", &policy);

    switch (policy) {
        case 1:
            pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
            param.sched_priority = sched_get_priority_max(SCHED_FIFO);
            break;
        case 2:
            pthread_attr_setschedpolicy(&attr, SCHED_RR);
            param.sched_priority = sched_get_priority_max(SCHED_RR);
            break;
        default:
            pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
            param.sched_priority = sched_get_priority_max(SCHED_FIFO);
            break;
    }

    if (pthread_attr_setschedparam(&attr, &param) != 0) {
        printf("Warning: Failed to set thread scheduling parameters. Run as root for SCHED_FIFO/SCHED_RR.\n");
    }
```

The first part in the main function makes it run on a single core.

Then we create 3 threads for the three functions.

Then it prompts the user to choose a scheduling algorithm and generate a warning message if he didn't choose a correct number(1,2).

```c
 t1_metrics.release_time = get_time_ms();
 pthread_create(&t1, &attr, thread1_func, NULL);
 pthread_setaffinity_np(t1, sizeof(cpu_set_t), &cpuset);

 t2_metrics.release_time = get_time_ms();
 pthread_create(&t2, &attr, thread2_func, NULL);
 pthread_setaffinity_np(t2, sizeof(cpu_set_t), &cpuset);

 t3_metrics.release_time = get_time_ms();
 pthread_create(&t3, &attr, thread3_func, NULL);
 pthread_setaffinity_np(t3, sizeof(cpu_set_t), &cpuset);

 // Wait for all threads to finish
 pthread_join(t1, NULL);
 pthread_join(t2, NULL);
 pthread_join(t3, NULL);

 double global_end_time = get_time_ms();
 double execution_time = global_end_time - global_start_time;
Calculate metrics for Thread 1
_metrics.execution_time = t1_metrics.finish_time - t1_metrics.start_time;
_metrics.turnaround_time = t1_metrics.finish_time - t1_metrics.release_time;
_metrics.waiting_time = t1_metrics.turnaround_time - t1_metrics.execution_time;
_metrics.response_time = t1_metrics.start_time - t1_metrics.release_time;
_metrics.utilization_time=t1_metrics.execution_time/(t1_metrics.execution_time+t1_metrics.waiting_time);
hahseb cpu utilization exec/turnaround
Calculate metrics for Thread 2
_metrics.execution_time = t2_metrics.finish_time - t2_metrics.start_time;
_metrics.turnaround_time = t2_metrics.finish_time - t2_metrics.release_time;
_metrics.waiting_time = t2_metrics.turnaround_time - t2_metrics.execution_time;
_metrics.response_time = t2_metrics.start_time - t2_metrics.release_time;
_metrics.utilization_time=t2_metrics.execution_time/(t2_metrics.execution_time+t2_metrics.waiting_time);
Calculate metrics for Thread 3
_metrics.execution_time = t3_metrics.finish_time - t3_metrics.start_time;
_metrics.turnaround_time = t3_metrics.finish_time - t3_metrics.release_time;
_metrics.waiting_time = t3_metrics.turnaround_time - t3_metrics.execution_time;
_metrics.response_time = t3_metrics.start_time - t3_metrics.release_time;
_metrics.utilization_time=t3_metrics.execution_time/(t3_metrics.execution_time+t3_metrics.waiting_time);

/ double cpu_useful_work = (t1_metrics.execution_time + t2_metrics.execution_time + t3_metrics.execution_time)/3;
ouble cpu_utilization =(t1_metrics.utilization_time + t2_metrics.utilization_time + t3_metrics.utilization_time)/3;
built in cpu usage
 printf("\nTotal execution time: %.4f ms\n", execution_time);
 //printf("CPU Useful Work: %.4f ms\n", cpu_useful_work);
 printf("CPU Utilization: %.2f%%\n", cpu_utilization);
 printf("memory allocation: %ld",memoryAllocation);
 printf("\nMetrics for each thread (All values in milliseconds, ms):\n");
```

This part we started the timer and created the threads for each function and we waited for all threads to finish using the pthread_join fubnction.

Then we calculated the metrics using the formulas mentioned above for each thread.

We then printed the results.