Operating Systems MileStone 1 Report

Team: 20

Names:

- Zeina Mohamed Fadel 55-1095
- Habiba Hesham 55-0966
- Ehab Medhat 55-4571
- Abdelrahman ElAby 55-0990
- Abdelrahman ElSamalouty 55-0842
- Hazem Mansour 55-1036

Libraries that we imported

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>
```

stdio: for inputs/outputs
stdlib: standard library for C code
pthread: for threads
sched: for scheduling
unistd: for accessing processes' attributes
sys/time and sys/resource: for getting our KPIs

Global Attributes

```
cpu_set_t cpu_set;
```

cpu_set is of type cpu_set_t, used as a global variable and will be assigned later with the specific processor that we want all our threads to run on

```
//Create a cpu set for the threads to use
CPU_ZERO(&cpu_set);
CPU_SET(0, &cpu_set);
```

In the main method, we first empty our set and then adds processor 0 to the set in order to run all the threads on that single processor

The Main Method's Attributes

```
pthread_attr_t attr;
pthread_attr_init(&attr);
struct sched_param param;
param.sched_priority = 0;
struct timespec start, end;
long long execution_time_ns;
```

attr: carries all details of the thread's behavior
param: structure that contains all the attributes details including the priority that is set to 0 if we're using the 'Other' scheduling policy and set to a number from 1-99 if we're using FIFO or Round Robin
timespec / execution_time_ns: used for getting our KPIs

Creating the Threads:

```
pthread_t threads[4];
int thread_ids[4] = {1, 2, 3, 4};
```

```
for (int i = 0; i < 4; i++) {
    pthread_create(&threads[i], &attr, thread_function, (void *)&thread_ids[i]);
}
for (int i = 0; i < 4; i++) {
    pthread_join(threads[i], NULL);
}
```

- First, we created an array of type pthread_t that carries our 4 threads
- We then loop to create all 4 threads in order (1,2,3,4) with the scheduling policy and parameter attr where these threads calls the “thread_function” (explained later below) method passing to it the threads’ ID
- After creating all 4 threads, we join them with the main method to ensure that all the threads will finish executing before the main method is finished

Setting The Scheduling Policies:

```
    // Set scheduling policy and priority for the main thread

    if(pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpu_set)!=0){
        printf("Error setting main thread affinity\n");
    }

    if(pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED)!=0){
        printf("Error setting thread inheritance\n\n");
    }

// Set scheduling policy and priority for worker threads
    if(pthread_setschedparam(pthread_self(),SCHED_OTHER,&param)!=0){
        printf("Error setting thread attribute policy\n");
    }


    if(pthread_attr_setschedpolicy(&attr, SCHED_OTHER)!=0){
        printf("Error setting thread attribute policy\n");
    }
```

- First, we set the affinity of the main method to the processor assigned in the cpu_set and print an error message if it doesn't succeed
- Second, allows the attribute to work with its own attributes and not inherit from the main method by setting the inheritance mode to "EXPLICIT" and print an error message if it doesn't succeed
- Third, Setting the scheduling policy and param of the main method to the specified scheduling policy (SCHED_OTHER, SCHED_FIFO, SCHED_RR) and print an error message if it doesn't succeed
- Fourth, Setting the scheduling policy of the attr to the specified scheduling policy (SCHED_OTHER, SCHED_FIFO, SCHED_RR) and print an error message if it doesn't succeed

KPIs:

```
// Get the start time
    clock_gettime(CLOCK_MONOTONIC, &start);
```

```
    // Measure CPU Utilization
 struct rusage usage;
 getrusage(RUSAGE_SELF, &usage);
 printf("CPU Utilization: User Time = %ld.%06ld s, System Time = %ld.%06ld s\n",
        usage.ru_utime.tv_sec, usage.ru_utime.tv_usec,
        usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
```

```
// Get the end time
clock_gettime(CLOCK_MONOTONIC, &end);

// Calculate the execution time
execution_time_ns = (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec);

printf("Thread execution time: %lld ns\n", execution_time_ns);
```

- Start time records the beginning of the main's execution, the end time records the ending of the main's execution and the difference records the total execution time
- We also get CPU usage time to determine which policy is more efficient

The function that our threads will execute

```c
void *thread_function(void *arg) {
    // Set thread affinity to a specific CPU core

    if(pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpu_set)!=0){
        printf("affinity errorrrr\n");
    }

    int thread_id = *((int *) arg); // Extract thread ID from argument
    printf("Thread %d: Starting\n", thread_id);

    // Perform some work (iterations)

    printf("Thread %d: Executing iteration %d\n", thread_id, 1);
    for(int i=0;i<80000000;i++){

    }
    printf("Thread %d: Executing iteration %d\n", thread_id, 2);
    for(int i=0;i<80000000;i++){

    }
    printf("Thread %d: Executing iteration %d\n", thread_id, 3);
    for(int i=0;i<80000000;i++){

    }
    printf("Thread %d: Executing iteration %d\n", thread_id, 4);
    for(int i=0;i<80000000;i++){

    }

    printf("Thread %d: Exiting\n", thread_id);
    pthread_exit(NULL);
}
```

- First, we set affinity of our threads to a specific CPU core that is stored in cpu_set and print an error message in case it does not succeed
- Next, we extract the thread ID from the function's argument
- Then, the first print statement is executed printing "starting" when the thread starts execution
- Then, 4 print statements are written
- Between those 4 print statements, we loop for 80000000 iterations in order to increase the execution time of the function so that it would be greater than the quantum
- Lastly, we print "exiting" when the thread is done executing

The output of this function for thread 1 for example should look like this:

```
Thread 1: Starting
Thread 1: Executing iteration 1
Thread 1: Executing iteration 2
Thread 1: Executing iteration 3
Thread 1: Executing iteration 4
Thread 1: Exiting
```

Scheduling Policies Outputs:

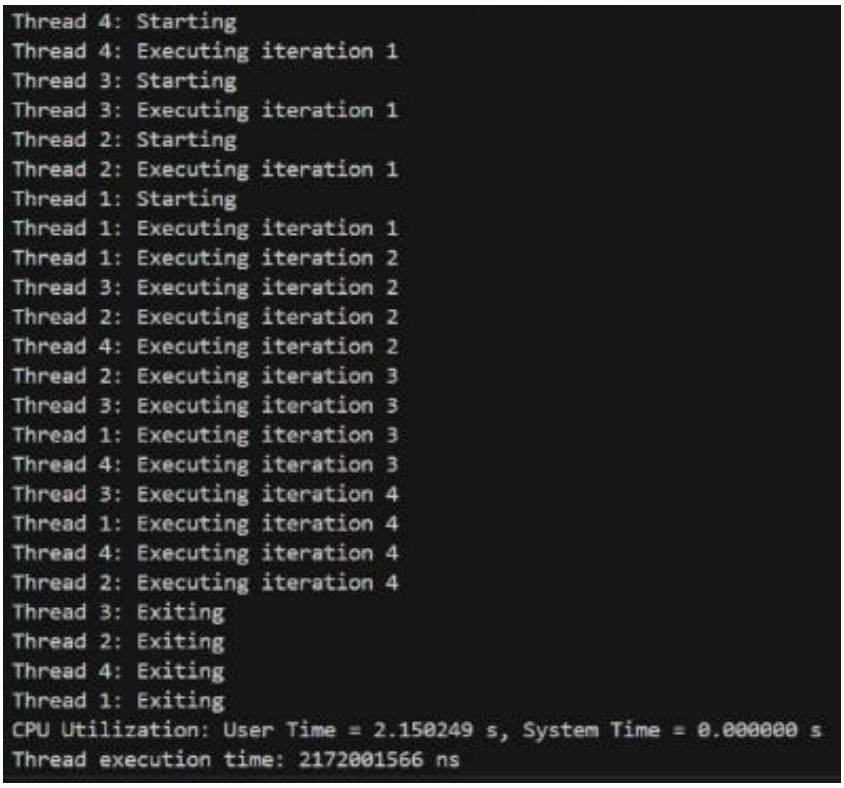
1. FIFO:

```
Thread 1: Starting
Thread 1: Executing iteration 1
Thread 1: Executing iteration 2
Thread 1: Executing iteration 3
Thread 1: Executing iteration 4
Thread 1: Exiting
Thread 2: Starting
Thread 2: Executing iteration 1
Thread 2: Executing iteration 2
Thread 2: Executing iteration 3
Thread 2: Executing iteration 4
Thread 2: Exiting
Thread 3: Starting
Thread 3: Executing iteration 1
Thread 3: Executing iteration 2
Thread 3: Executing iteration 3
Thread 3: Executing iteration 4
Thread 3: Exiting
Thread 4: Starting
Thread 4: Executing iteration 1
Thread 4: Executing iteration 2
Thread 4: Executing iteration 3
Thread 4: Executing iteration 4
Thread 4: Exiting
CPU Utilization: User Time = 2.364122 s, System Time = 0.000000 s
Thread execution time: 2473331005 ns
```

2. Round Robin:

```
Thread 1: Starting
Thread 1: Executing iteration 1
Thread 2: Starting
Thread 2: Executing iteration 1
Thread 3: Starting
Thread 3: Executing iteration 1
Thread 4: Starting
Thread 4: Executing iteration 1
Thread 1: Executing iteration 2
Thread 2: Executing iteration 2
Thread 3: Executing iteration 2
Thread 4: Executing iteration 2
Thread 1: Executing iteration 3
Thread 2: Executing iteration 3
Thread 3: Executing iteration 3
Thread 4: Executing iteration 3
Thread 1: Executing iteration 4
Thread 2: Executing iteration 4
Thread 3: Executing iteration 4
Thread 4: Executing iteration 4
Thread 1: Exiting
Thread 2: Exiting
Thread 3: Exiting
Thread 4: Exiting
CPU Utilization: User Time = 2.282418 s, System Time = 0.009943 s
Thread execution time: 2403010859 ns
```

3. Other:



Comparing between different scheduling policies:

All scheduling policies have approximately the same time for execution with minor differences where the ‘Other’ is the fastest and most efficient then comes the ‘Round Robin’ and lastly ‘FIFO’