Source Code - lab04_ex1.c

```
/** Riya Patel
CSC 345-01
Lab 4 Exercise 1
**/
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char** argv)
{
    int n = atoi(argv[1]);
    int i, j;
    int count = 0;
    time_t begin = time(NULL);
    pid_t id = getpid();

    for (i=1; i <= n; ++i)
    {
        for (j=2;j<i;++j)
        {
            if (i % j == 0) {
                break;
            }
        }
        if (j == i)
        {
            ++count;
        }
    }

    printf("\n");
    printf("* Process %d found %d primes within [1, %d] in %ld
seconds\n", id, count, n, time(NULL) - begin);

    return 0;
}
```

Discussion - lab04_ex1.c

```
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$ ./lab04_ex1 10

* Process 27051 found 4 primes within [1, 10] in 0 seconds
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$ ./lab04_ex1 1000

* Process 27052 found 168 primes within [1, 1000] in 0 seconds
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$ ./lab04_ex1 100000

* Process 27053 found 9592 primes within [1, 100000] in 1 seconds
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$ ./lab04_ex1 1000000

* Process 27066 found 78498 primes within [1, 1000000] in 117 seconds
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$
```

Figure 1: ./lab04_ex1 Test Case Results

```
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$ nice -n 19 ./lab04_ex1 300000 &
[1] 27621
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$ ./lab04_ex1 300000 &
[2] 27630
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$
* Process 27621 found 25997 primes within [1, 300000] in 12 seconds

* Process 27630 found 25997 primes within [1, 300000] in 12 seconds

[1]-  Done                    nice -n 19 ./lab04_ex1 300000
[2]+  Done                    ./lab04_ex1 300000
osc@osc-VirtualBox:~/Labs/csc345-OS-main/Lab4$
```

Figure 2: ./lab04_ex1 Test Case Results with Priority-Based Scheduling

Exercise 1 calculates the amount of prime numbers within the range of 1 and the given user input. As expected and shown in Figure 1, the larger the range, the longer it takes for the program to count the total number of primes. As the input increases, the time it takes for the program to complete increases exponentially as a result of the modular calculations function time in Linux. Figure 2 represents priority scheduling. The 19 indicates that the program is being run with high priority (highest priority is 20) and as a result the process in the background gets completed first as it is the higher priority.

Source Code - lab04_ex2.c

```
/** Riya Patel
CSC 345-01
Lab 4 Exercise 2
**/

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void* runner (void* param){
    pthread_exit(0);
}

int main(int argc, char** argv){
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    pthread_attr_init(&attr);

    if(pthread_attr_getschedpolicy(&attr, &policy) != 0){
        fprintf(stderr, "Unable to get policy\n");
    } else {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)
        fprintf(stderr, "Unable to set policy.\n");

    if(pthread_attr_getschedpolicy(&attr, &policy) != 0){
        fprintf(stderr, "Unable to get policy\n");
    } else {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }
```

```
    for (i=0;i<NUM_THREADS;i++)
        pthread_create(&tid[i],&attr,runner,NULL);

    for (i=0;i<NUM_THREADS;i++)
        pthread_join(tid[i],NULL);
}
```

Discussion - lab04_ex2.c

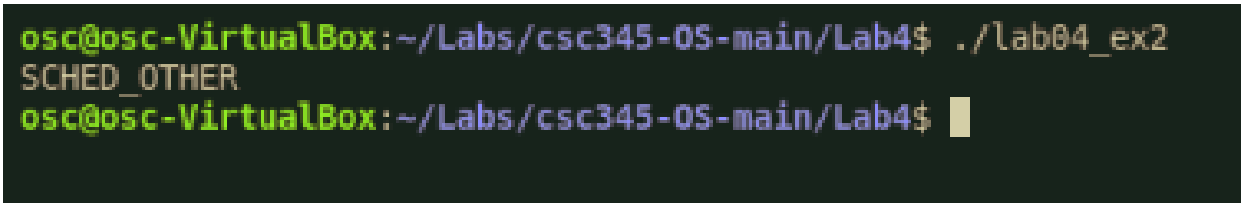

Figure 3: ./lab04_ex2 Results (No Threads)

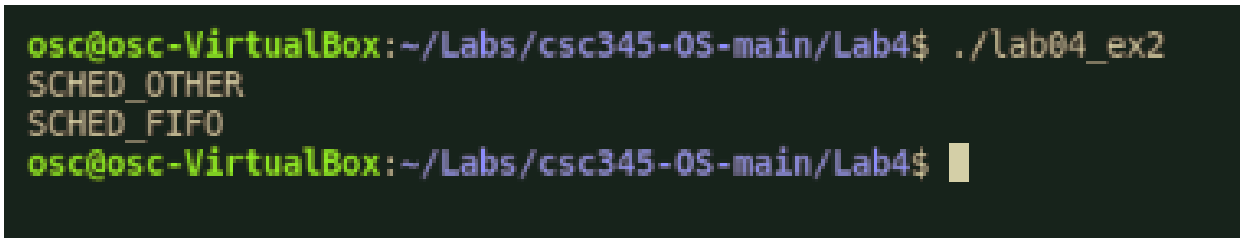

Figure 4: ./lab04_ex2 Results with FIFO Scheduling Policy (with Threads)

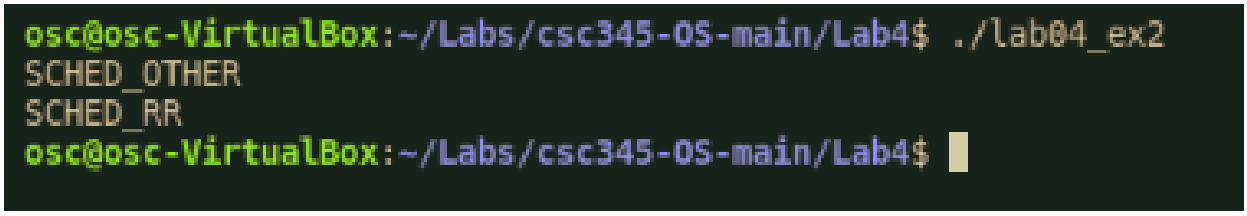

Figure 5: ./lab04_ex2 Results with RR Scheduling Policy (with Threads)

Exercise 2 is a mechanism to see the effect of different scheduling policies and evaluate when it is and isn't suitable. Figure 3 shows the output when there are no threads present in the program. As a result, the default scheduling policy (SCHED_OTHER) is only outputted once. When threads are present, as shown in Figure 4, there are two outputs and in this case the scheduling policy is set to SCHED_FIFO. In Figure 5, there are still threads present but the scheduling policy is set to SCHED_RR.