Jordan Sinoway

CSC345-02

Lab 4 – Scheduler

March 3, 2021

Code – lab04_ex1

```
// Jordan Sinoway
// CSC345-02
// Lab 4 Ex 1
// 2/25/2021

//priority based scheduling
//finds time it takes to calculate prime numbers of user input val

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char** argv)
{
    int n = atoi(argv[1]); //converting 2nd arg to integer val
    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)
        {
            // printf("%d ", j);
            ++count;
        }
    }

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

    return 0;
}
```

Explanation & Screenshots – lab04_ex1

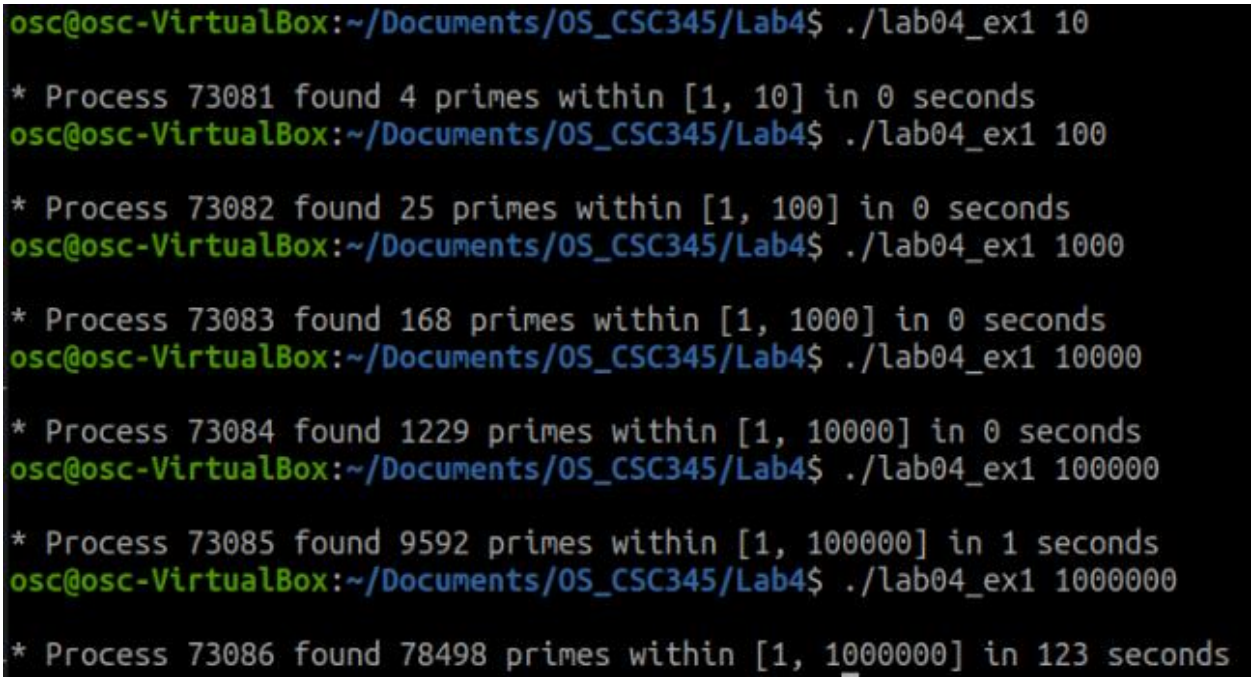

Figure 1

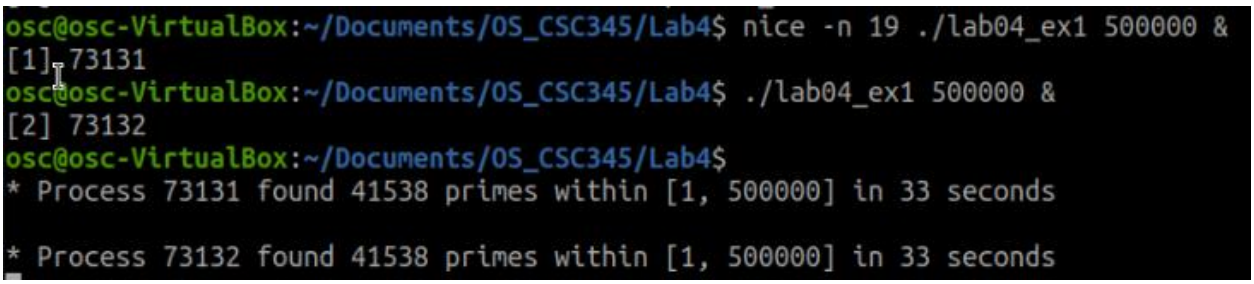

Figure 2

Example 1 calculates the number of prime numbers in a given user input value. This calculation is completed with the simple remainder calculation shown in the code above. The number of primes is outputted along with the how long it takes the program to run, which can be seen in Figure 1. Figure 1 also shows the exponential increase in processing time as the input size increases, which is due to how the modulo operator, and division by extension, works in Linux. The program also shows how priority-based scheduling works using the “nice” command, where the priority of a program can be set when running the program in console, as shown in Figure 2. The “NICE value” that is set is on a range from 20 (highest) to 10 (lowest) priority.

Code – lab04_ex2

```
// Jordan Sinoway
// CSC345-02
// Lab 4 Ex 2
// 2/25/2021

#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

// Each thread will begin control in this function
void *runner(void *param)
{
    // do some work ...
    //printf("nice\n");
    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); // get default attributes

    // get the current scheduling policy
    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");
    }

    // set the scheduling policy - FIFO, RR, or OTHER
    if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
        fprintf(stderr, "Unable to set policy.\n");

    // create the threads
    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);

    return 0;
}
```

Explanation & Screenshots – lab04_ex2

```
osc@osc-VirtualBox:~/Documents/OS_CSC345/Lab4$ ./lab04_ex2
SCHED_OTHER
```

Figure 3

```
osc@osc-VirtualBox:~/Documents/OS_CSC345/Lab4$ ./lab04_ex2
SCHED_OTHER
SCHED_FIFO
```

Figure 4

```
osc@osc-VirtualBox:~/Documents/OS_CSC345/Lab4$ ./lab04_ex2
SCHED_OTHER
SCHED_RR
```

Figure 5

Example 2 shows how the different scheduling policies can be set within a program, and which one is best suited for a given task. Figure 4 shows the output of the program when there are no threads created, and the default scheduling policy, SCHED_OTHER is shown. Figure 5 shows the output of the program before and after threads have been created, with the scheduling policy set to SCHED_FIFO. Figure 6 shows the output of the program, again after threads have been created, but this time with the scheduling policy set to SCHED_RR. SCHED_OTHER is the default policy, while SCHED_FIFO (first-in first-out) and SCHED_RR (round robin) are the real-time scheduling policies.