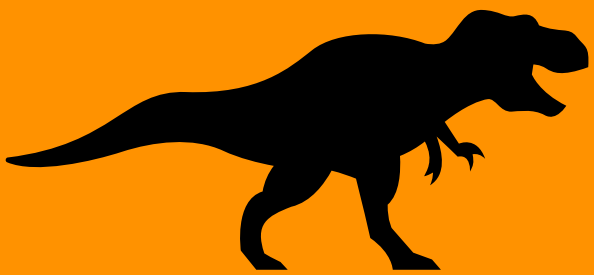
# THREADS AND PROCESSES

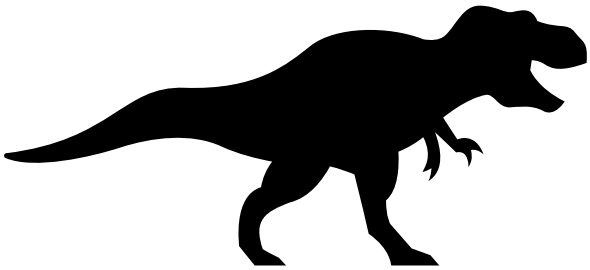
# OPERATING SYSTEM



COLLABORATORS NAME & ID :
MARYAM ALIKARAMI 9731045
MAHAN AHMADVAND 9731071

## THREADS AND PROCESS

# OPERATING SYSTEM




COLLABORATORS NAME & ID :
MARYAM ALIKARAMI 9731045
MAHAN AHMADVAND 9731071

# OPERATING SYSTEM LABORATORY

## THREADS AND PROCESSES


INSTRUCTOR :

ARMAN GHEISARI

COLLABORATORS :

MARYAM ALIKARAMI

MAHAN AHMADVAND

# QUESTION 1

**در گام اول از این مساله قصد داریم که نمونه‌برداری مساله را از روی تکه کدی که به صورت سریال نوشته شده‌است، انجام بدهیم. به این منظور، تکه کد زیر را نوشته‌ایم:**

serial_sampling.c

```c
int main(){

    int iteration = 0;
    printf("Enter iteration: ");
    scanf("%d", &iteration);

    srand(time(0));
    clock_t start = clock();

    int hist[25] = {0}; //0 to 11 : -12 to -1 , 12 to 24 : 0 to 12 => index + 12

    for (int i = 0; i < iteration; ++i) {

        int counter = 0;
        for (int j = 0; j < 12; ++j){

            int random = rand()%(100 - 0 + 1);
            if (random >= 49)
                counter++;
            else
                counter--;
        }
        hist[counter + 12] += 1;
    }

    // time
    clock_t end = clock();
    double time_spent = (double)(end - start)/CLOCKS_PER_SEC;
    printf("\nTime Spent %lf\n", time_spent);

    int sum = 0;
    // print hist
    for (int i = 0; i < 25; ++i){
        sum += hist[i];
        printf(" %d : %d |", i - 12, hist[i]);
    }
    // must be equal to iteration
    printf("\n sum of the cells is : %d \n", sum);

    // print Histogram
    for (int i = 0; i < 25; ++i){

        printf("hist[%d] : ", i - 12);

        for (int j = 0; j < (int)(hist[i]/(iteration/100)); ++j)
            printf("*");
        printf("\n");
    }
```

همانطوری که مشخص است، عناصر ۰ تا ۱۱ آرایه hist را نمایندگان اعداد ۱۲- تا ۱- در نظر گرفته‌ایم و عناصر ۱۲ تا ۲۴ نیز، نمایندگان اعداد ۰ تا ۱۲ هستند به همین دلیل زمانی که با استفاده از counter می‌خواهیم به یک خانه از آرایه‌ی hist که نماینده‌ی مقدار counter است مقداردهی کنیم، کافی است که به خانه‌ی counter + 12 دسترسی پیدا کنیم.

با استفاده از تابع clock() زمان سپری شده برای پردازش این برنامه را محاسبه می‌کنیم و در انتها نمایش می‌دهیم.

حاصل جمع عناصر hist را نیز محاسبه می‌کنیم تا مطمئن شویم که این مقدار برابر iteration است.

از این کد برای iterationهای 5000، 50000 و 500000 ران می‌گیریم تا جدول خواسته شده در گام اول را تکمیل کنیم:

5000:

```
oslab@OSLab-VirtualBox:~/Desktop/Lab 5$ ./serial
Enter iteration: 5000

Time Spent 0.002919
 -12 : 1 | -11 : 0 | -10 : 14 | -9 : 0 | -8 : 59 | -7 : 0 | -6 : 200 | -5 : 0 |
-4 : 522 | -3 : 0 | -2 : 941 | -1 : 0 | 0 : 1114 | 1 : 0 | 2 : 1022 | 3 : 0 | 4
: 683 | 5 : 0 | 6 : 309 | 7 : 0 | 8 : 111 | 9 : 0 | 10 : 23 | 11 : 0 | 12 : 1 |
 sum of the cells is : 5000
hist[-12] :
hist[-11] :
hist[-10] :
hist[-9] :
hist[-8] : *
hist[-7] :
hist[-6] : ****
hist[-5] :
hist[-4] : **********
hist[-3] :
hist[-2] : ******************
hist[-1] :
hist[0] : **********************
hist[1] :
hist[2] : ********************
hist[3] :
hist[4] : *************
hist[5] :
hist[6] : ******
hist[7] :
hist[8] : **
hist[9] :
hist[10] :
hist[11] :
hist[12] :
```

همانطوری که مشاهده می‌کنید، مقدار جمع عناصر ۵۰۰۰ است که برابر iteration میباشد، این یعنی سلول‌ها به درستی مقداردهی شده‌اند. بیشترین فراوانی متعلق به عدد ۰ است یعنی اینکه در بیشتر iterationهای انجام شده (که مقدار دقیق آن برابر ۱۱۱۴ مرتبه است) مقدار counter برابر ۰ شده است.

زمان صرف شده برای پردازش این تعداد نمونه، ۲.۹۱۹ میلی ثانیه است.

50000:

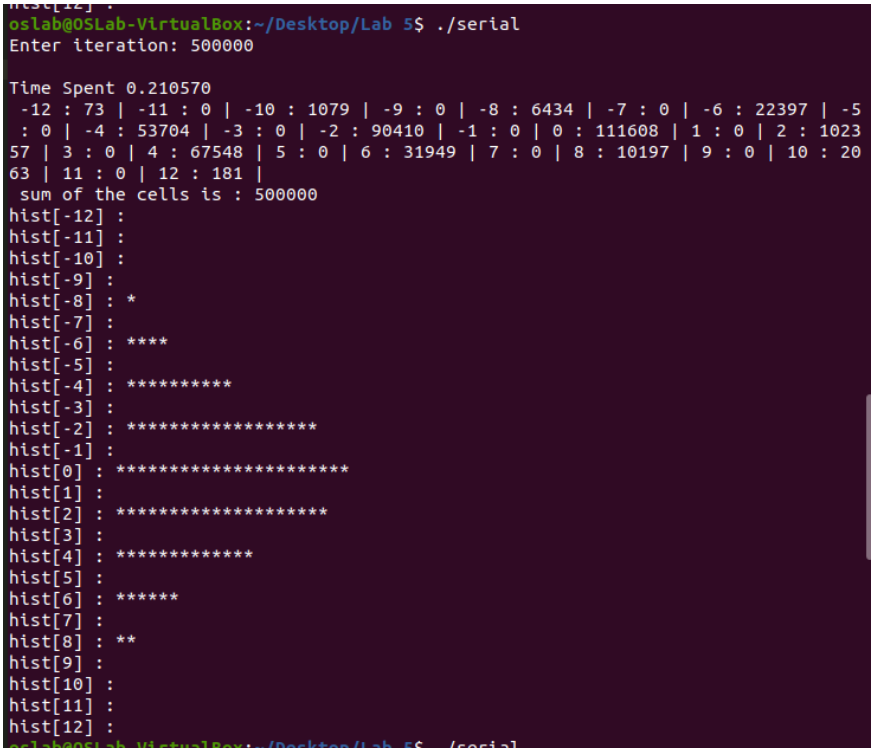
```
oslab@OSLab-VirtualBox:~/Desktop/Lab 5$ ./serial
Enter iteration: 50000

Time Spent 0.022952
 -12 : 7 | -11 : 0 | -10 : 102 | -9 : 0 | -8 : 630 | -7 : 0 | -6 : 2266 | -5 : 0
 | -4 : 5324 | -3 : 0 | -2 : 9083 | -1 : 0 | 0 : 11422 | 1 : 0 | 2 : 10106 | 3 :
 0 | 4 : 6761 | 5 : 0 | 6 : 3157 | 7 : 0 | 8 : 951 | 9 : 0 | 10 : 176 | 11 : 0 |
 12 : 15 |
 sum of the cells is : 50000
hist[-12] :
hist[-11] :
hist[-10] :
hist[-9] :
hist[-8] : *
hist[-7] :
hist[-6] : ****
hist[-5] :
hist[-4] : **********
hist[-3] :
hist[-2] : ******************
hist[-1] :
hist[0] : **********************
hist[1] :
hist[2] : ********************
hist[3] :
hist[4] : ************
hist[5] :
hist[6] : ******
hist[7] :
hist[8] : *
hist[9] :
hist[10] :
hist[11] :
hist[12] :
```

مجددا فراوانی عدد ۰ به عنوان مقدار counter در ۵۰۰۰۰ iterationای که انجام شده‌است، از باقی مقادیر این متغیر، بیشتر می‌باشد. مجموع تکرارهای counter (مجموع مقادیر عناصر hist) هم برابر ۵۰۰۰۰ است که برابر با تعداد iterationها و مقدار مورد انتظار ماست.

زمان صرف شده برای پردازش این تعداد نمونه، ۲۲.۹۵۲ میلی ثانیه است.

**500000:**




**مجددا فراوانی‌ها به همان صورت است. مجموع عناصر hist نیز با تعداد iterationها برابر است.**

**زمان صرف شده برای پردازش این تعداد نمونه، ۲۱۰.۵۷ میلی ثانیه است.**

**با استفاده از مقادیر به دست آمده، جدول زیر را تکمیل می‌کنیم:**

| تعداد نمونه | ۵۰۰۰ | ۵۰۰۰۰ | ۵۰۰۰۰۰ |
|---|---|---|---|
| زمان اجرا (ms) | ۲.۹۱۹ | ۲۲.۹۵۲ | ۲۱۰.۵۷ |

# QUESTION 2

حال برنامه ای بنویسید که با استفاده از fork() و یا exec() تعدادی فرآیند فرزند ایجاد شود و کارها را پخش کند.
ابتدا یک نگاهی به کد می اندازیم :

```c
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <string.h>
#include <pthread.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>

int const num = 10;
int id;
int root_pid;
int pid;

typedef struct
{
    int histData[25];
} Hist;

Hist *hist;

main()
{
    int iteration;

    scanf("%d", &iteration);

    clock_t start = clock();

    srand(time(0));

    id = shmget(IPC_PRIVATE, sizeof(Hist), IPC_CREAT | 0666);

    root_pid = getpid();

    for (int i = 0; i < num; i++)
    {
        if (getpid() == root_pid)
            pid = fork();
        else
            break;
    }

    hist = (Hist *)shmat(id, NULL, 0);


    if (getpid() == root_pid)
    {
        for (int i = 0; i < num; i++)
        {
            wait(NULL);
        }
    }
    else
    {
        // generating count
        int count, random;
        for (int i = 0; i < iteration / num; i++)
        {
            count = 0;
            for (int j = 0; j < 12; j++)
            {
                random = rand() % 101;
                if (random >= 49)
                {
                    count++;
```

```
                else
                {
                    count--;
                }
            }
            hist->histData[count + 12]++;
        }

        exit(0);
    }

    clock_t end = clock();
    double time_spent = (double)(end - start)/CLOCKS_PER_SEC;
    printf("\nTime Spent %lf\n", time_spent);
    // print result
     for (int i = 0; i < 25; i++){
         printf("%d ", i - 12);
          for(int j = 0; j < (int)(hist->histData[i]/(iteration/100)); j++) {
           printf("*");
         }
         printf("\n");
        }

    return 0;
}
```

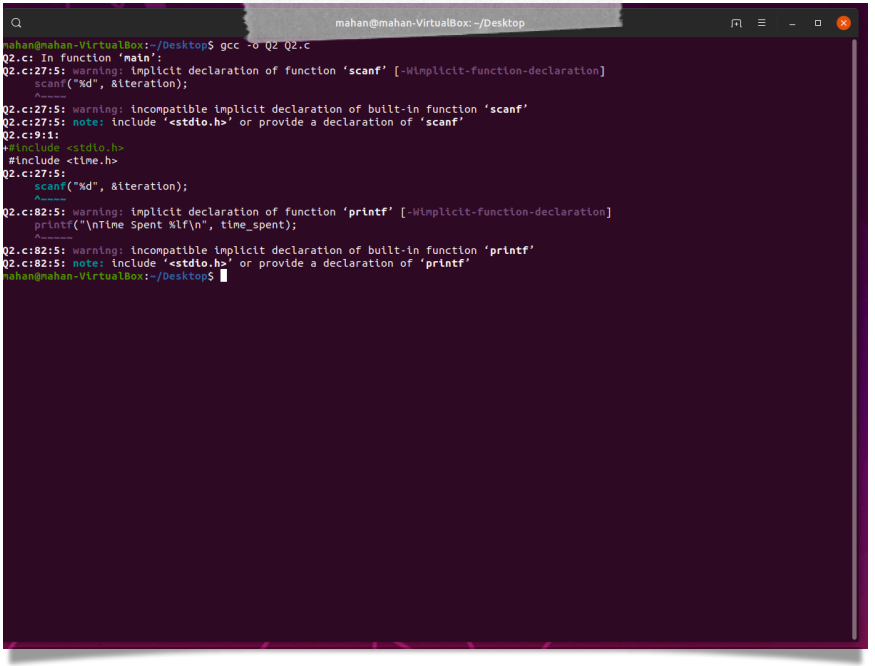
همانطور که می بینیم از مفهوم fork برای پیاده سازی این قسمت استفاده کرده ایم.

در اینجا پدر را ریشه ی درخت درنظر گرفته ایم و PID آن را داخل متغیر root_pid ریخته ایم.

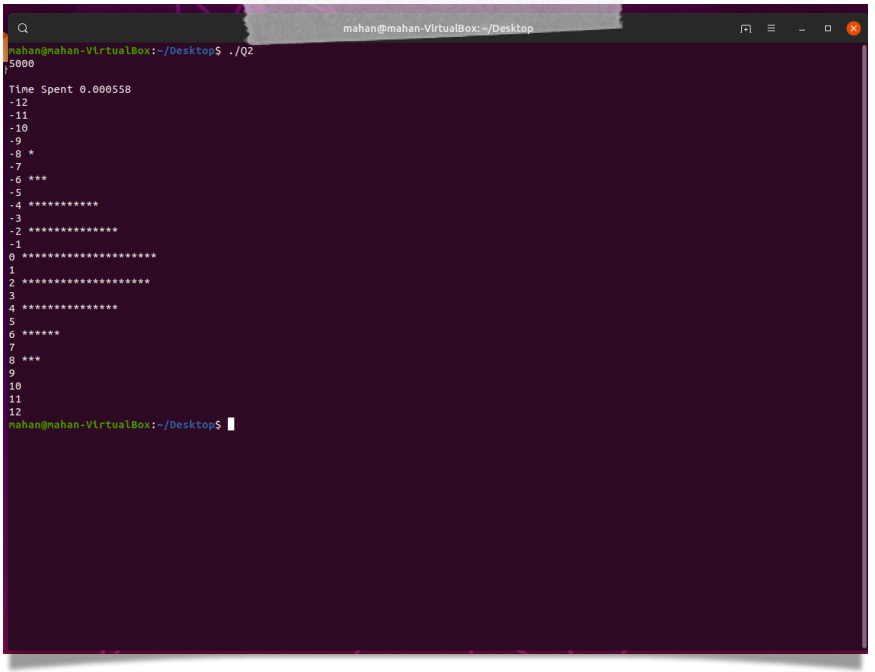
روند کار به این صورت است که root تعدادی دلخواه مثلا در اینجا 10 فرزند تولید می کند(تعداد فرزندان با متغیر num مشخص شده است) و این 10 فرزند هر کدام به اندازه ی iteration/num عملیات را انجام می دهند، که در اینجا iteration تعداد نمونه و num تعداد process های فرزند است.

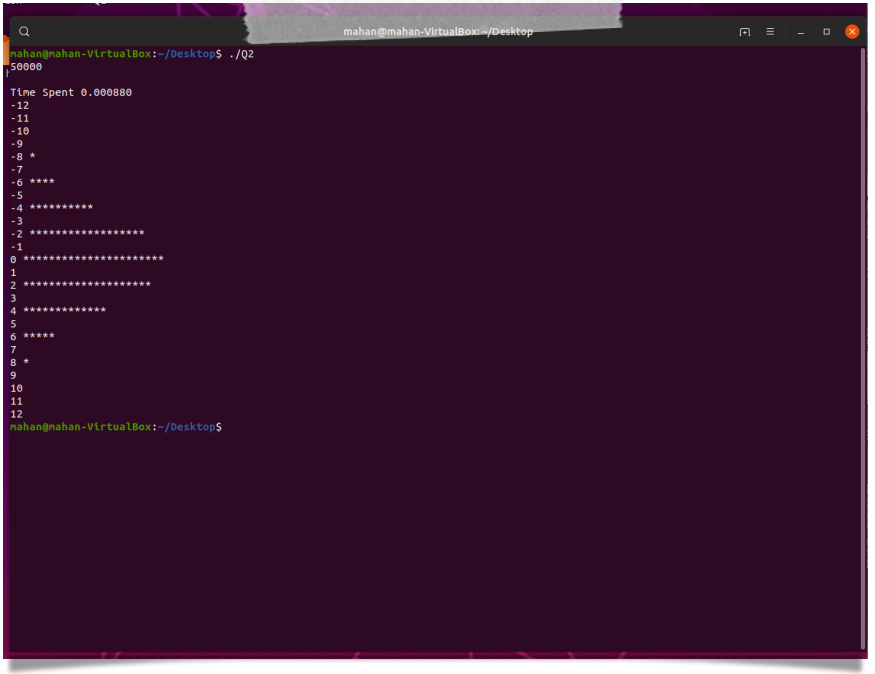
همچنین از مفهموم shared memory برای پیاده سازی استفاده کرده ایم، و process ها همگی با این حافظه ی اشتراکی کار می کنند(عملیات های موردنیاز برای ایجاد حافظه ی اشتراکی مطابق کد بصورت کامل انجام شده است).

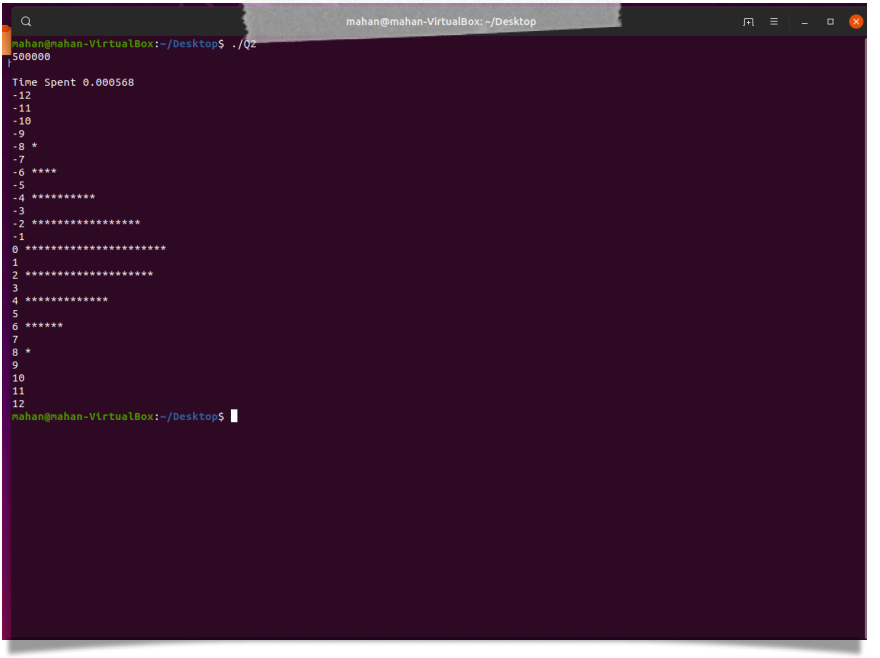
حال کد نشان داده شده را کامپایل می کنیم :




سپس به ازای ورودی های مختلف 5000 ، 50000 و 500000 آن را ران می کنیم :


mahan@mahan-VirtualBox: ~/Desktop
mahan@mahan-VirtualBox:~/Desktop$ ./Q2
5000
Time Spent 0.000558
-12
-11
-10
-9
-8 *
-7
-6 ***
-5
-4 **********
-3
-2 **********
-1
0 **********
1
2 **********
3
4 **********
5
6 ******
7
8 ***
9
10
11
12
mahan@mahan-VirtualBox:~/Desktop$


mahan@mahan-VirtualBox: ~/Desktop
mahan@mahan-VirtualBox:~/Desktop$ ./Q2
50000
Time Spent 0.000880
-12
-11
-10
-9
-8 *
-7
-6 ****
-5
-4 **********
-3
-2 **********
-1
0 **********
1
2 **********
3
4 **********
5
6 *****
7
8 *
9
10
11
12
mahan@mahan-VirtualBox:~/Desktop$




حال جدول موردنظر را کامل می کنیم :

| تعداد نمونه | 5000 | 50000 | 500000 |
|---|---|---|---|
| زمان اجرا (ms) | 0.558 | 0.88 | 0.568 |

# QUESTION 3

آیا این برنامه درگیر شرایط مسابقه می شود؟چگونه؟ اگر جوابتان مثبت بود راه حلی برای آن بیابید.

بله می تواند درگیر شرایط مسابقه شود، زیرا این برنامه دارای Process های مختلفی است که هر کدام می توانند به متغیر مشترک دسترسی داشته باشند و آن را تغییر دهند(hist) و در واقع این ناحیه ی مشترک را Critical Section می نامیم که در واقع برای حل این مشکل باید مسئله Critical Section را حل کنیم.

برای حل این مسئله، پاسخ ما به مسئله باید سه شرط Mutual Exclusion، Progress و Bounded waiting را داشته باشد.

می توانیم این مشکل را با سمافور حل کنیم.

اما این راه حل چگونه است؟

سمافور یک متغیر نامنفی است و بین تردها به اشتراک گذاشته می شود و در واقع یک مکانیزم سیگنالی است.

سمافور از دو عملیات اتمیک استفاده می کند wait و signal.

Semaphore در واقع یک متغیری است که سیستم عامل به ما می دهد و به ما می گوید که فقط حق داری که دوتا تابع wait و signal را روی آن پیاده سازی کنی.

هنگامی که تابع wait را روی سمافور فراخوانی بکنیم مقدار آن یک واحد کم می شود و هنگامی که تابع signal را روی آن فراخوانی بکنیم مقدار آن یکی افزایش می یابد.

هنگامی که سمافور به صفر رسید دیگر تابع wait نمی تواند مقدار آن را کمتر کند و پردازه ای که wait را فراخوانی کرده است، ادامه اش اجرا نمی شود و صبر می کند تا پردازه ی دیگری signal را فراخوانی کند، هنگامی که signal توسط پردازه ای صدا زده شد مقدار سمافور همچنان صفر می ماند اما پردازه ای که برای signal منتظر مانده است، از حالت بلاک خارج شده و ادامه اش اجرا می شود و بعد از آن هر پردازه ای که signal را فراخوانی کند مقدار آن افزایش می یابد و از این به بعد پردازه های دیگر حق دارند که سمافور را wait کنند

# QUESTION 4

نتایج قسمت اول و دوم را مقایسه کنید و میزان افزایش سرعت را در جدول زیر گزارش دهید.

| تعداد نمونه | 5000 | 50000 | 500000 |
|---|---|---|---|
| افزایش سرعت | 80% | 96% | 97% |

همانطور که مشاهده می کنیم سرعت به طور چشم گیری با استفاده از fork() افزایش یافته است و این به این دلیل است که موازی سازی و همروندی در برنامه بیشتر شده است و کارها تقسیم شده اند.


THE END