Operating System (ECON F372)

Assignment 2

Group Number: 23

Name	ID Number
RHYTHM SETHI	2019B3A71306H
ISHAN RASTOGI	2019B3A71305H
ABHINAV BANSAL	2019B3A71293H
SIDDHANT GARG	2019B5A70760H
GAURAV KUMAR	2019B3A71324H
SATYAM GUPTA	2019B3A71277H

Date: December 1, 2022

Using the parallelism offered by the Linux process and threads libraries, we have multiplied arbitrary large sized matrices.

As command line parameters, P1 expects the filenames in1.txt, in2.txt, and out.txt to be provided. The first two text files, in1.txt and in2.txt, each contain matrices that are of arbitrary sizes but yet meet the requirements for matrix multiplication. On the command line, the dimensions need to be specified.
P1 generates n new threads, and each of those new threads reads rows and columns from both in1.txt and in2.txt.

Threads read separate file portions.

P1 reads rows and columns into P2 via IPC and program P2 generates threads to calculate product matrix cells.

In the code P1:

```
void initialize(){
    file1=fopen(in1,"r");
    file2=fopen(in2,"r");
    int matrix1[a][n], matrix2[n][b];
    mat1seek=(int*) malloc(a*n*sizeof(int));
    mat2seek=(int*) malloc(b*n*sizeof(int));
    for(int i=0;i<a;i++){
        for(int j=0;j<n;j++){
            mat1seek[i*n+j]=ftell(file1);
            fscanf(file1,"%d ",&matrix1[i][j]);
        }
    }
    for(int i=0;i<n;i++){
        for(int j=0;j<b;j++){
            fscanf(file2,"%d",&matrix2[i][j]);
        }
    }
    printf("\n");
    transpose=fopen("transpose.txt","a");
    transpose=fopen("transpose.txt","w+");

    int k=0;
    for(int j=0;j<b;j++){
        for(int i=0;i<n;i++){
            mat2seek[k++]=ftell(transpose);
            fprintf(transpose,"%d ",matrix2[i][j]);
        }
        fprintf(transpose,"\n");
    }
}
```

The initialise() function pre-traverses input matrices and stores the beginning of each character. We save input2.txt's transpose as transpose.txt for quicker access.

Now, a buffer of arbitrarily large size is created and used to deliver data to P2.c through shared memory. Then N threads are created, and they start reading data from files and storing it in a shared buffer with P2 (using Shared memory).
The first ‘a’ rows of shared memory hold the input1 matrix, whereas the following ‘b’ rows have the input2 matrix.

PROGRAM P2:

```
void* mul(void* arg){
    int *numTemp = (int*)arg;
    int num = *numTemp;
    int i, j, k;
    for(i=0; i<a; i++){
        for(j=0; j<b;j++){
            ans[i][j]=0;
            for(k=0; k<n; k++){
                ans[i][j]+=buf1Ptr[i][k]*buf1Ptr[a+j][k];
            }
        }
    }
}
```

This time in P2, we'll use the same id and key to create a shared memory that will allow us to read data sent from P1.
We need to be patient since P1 must first do some preprocessing on the input files before it can begin sending data to P2. There will be a short sleep of 150 ms (selected randomly).

```
int main(int argc, char *argv[])
{
    // printf("HEREEEEEEE-----> %d\n", getppid());
    a=atoi(argv[0]);n=atoi(argv[1]);b=atoi(argv[2]);
    in1=argv[3];in2=argv[4];out=argv[5];
    createBuf();
    pthread_t tid[N];
    usleep(150000);
```

This is followed by P2 making N separate calls to threads, where the multiplication takes place. Multiplication output is saved in out.txt. The scheduler is informed that P2 has completed execution when it receives a signal indicating that P2 has ended. To top it all off, it would indicate the termination of the parent process.

```
shmctl(shmBuf1id, IPC_RMID, NULL);
kill(getppid(), SIGUSR2);
```

After P2 has received the contents of P1, we release P2's shared memory using shmctl(), as illustrated above.

Scheduler:

```
int main(int argc, char *argv[]) {

    signal(SIGUSR1, signalHandler);
    signal(SIGUSR2, signalHandler);
```

Two signal handlers are included in Schedular to accept signals from P1 and P2. Scheduler duties will be performed by the function RR(). It would be forked twice, with the first child executing to call P1 and the second calling P2. Round robin scheduling will now be carried out by a thread that the parent has just spawned.

```
void* RRscheduling() {
    // int *p2 = (int*) z;
    // int p = *p2;
  long long int timeQuantum = 1000;
```

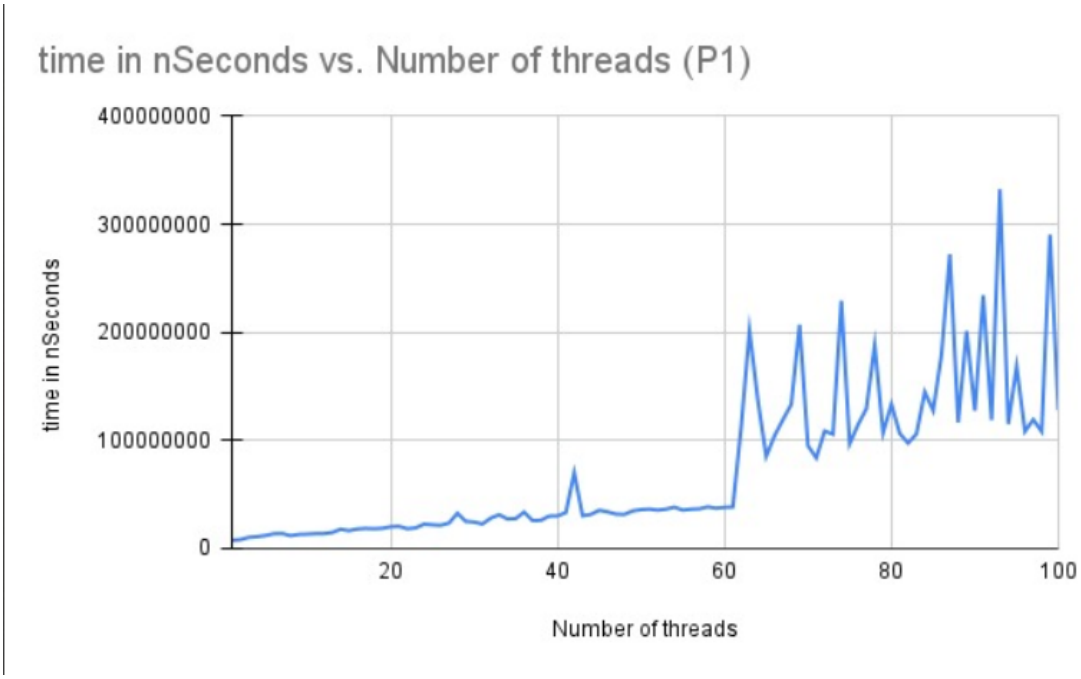
We may define the time quantum that we wish to employ for our processes here in round robin scheduling.

We also compute the typical duration of a context transition and output that value to the console.

```
demantor@DESKTOP-LGKUSMG:/mnt/c/All Files/4-1/OS/ass2$ ./t 50 20 50 in1.txt in2.txt out1.txt

Caught SIGUSR1 signal.
Caught SIGUSR2 signal.
waitTimeP1: 25.000000
waitTimeP2: 50.000000
tat1: 50.000000
tat2: 188.664993
Average switching time : 54270 nanoseconds
```

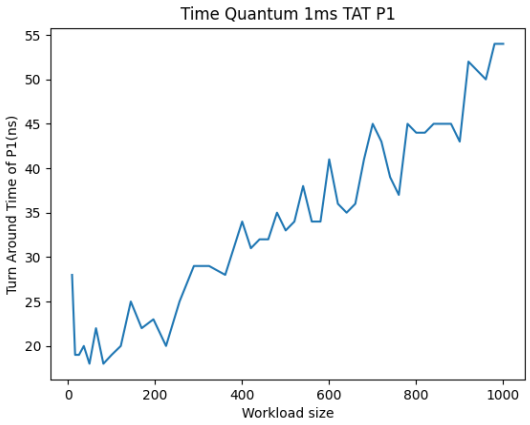
RESULTS AND DISCUSSIONS:

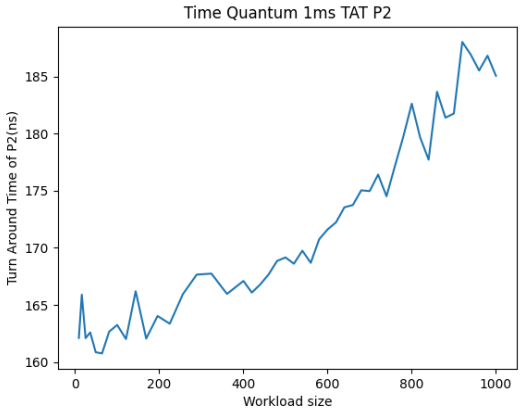


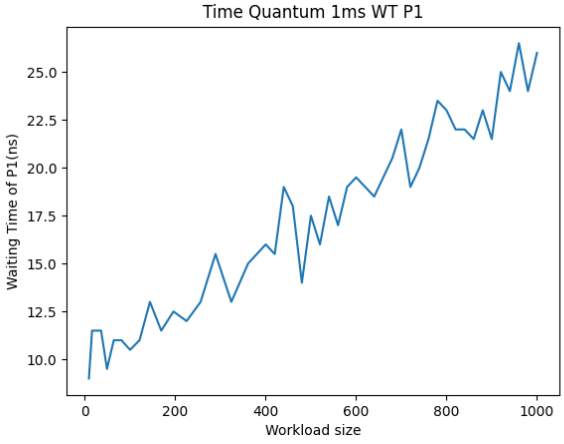
The above graph displays the time taken for P1 to read data from file in granularity of nano seconds by varying the number of threads from 1 to 100.
We find that on an average the least time was taken if the number of threads were 2. So, optimal no. of threads to read contents from files using threads in P1 is 2.

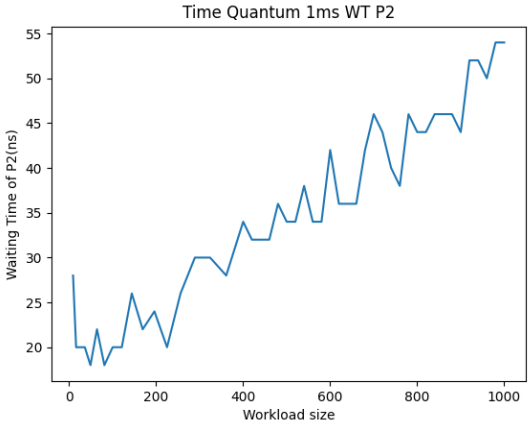
The above graph displays the time taken for P2 to multiply the matrices and store the result in output file. Here we see that on increasing the number of threads the time is increasing on an average. Since, various threads are attempting perform operations on same matrix simultaneously, some data that is required by the threads for multiplication might not be available which results in increase in time consumed by P2. Here too we found the optimal number of threads equal to 2.

Case 1: When Time Quantum is 1ms







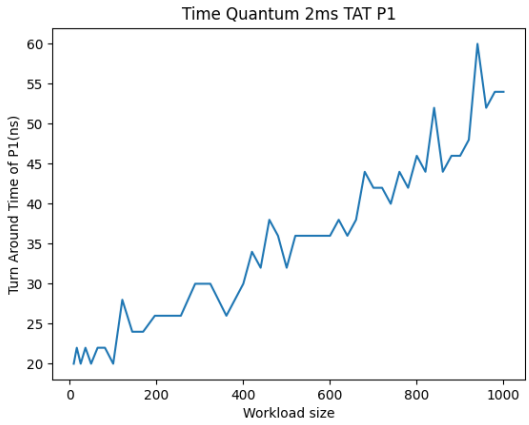


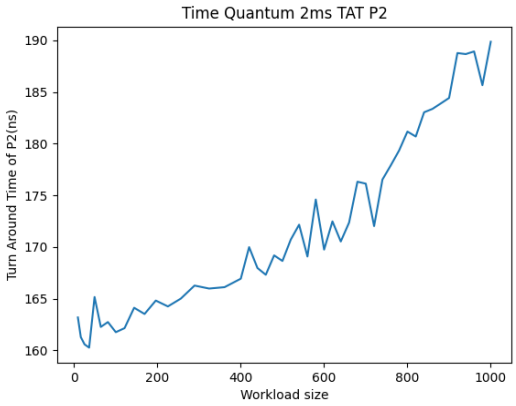
The four graphs plotted above are of:

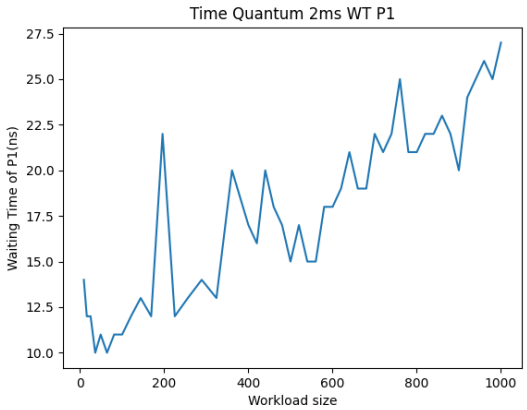
1. Turn Around Time for P1 vs WorkloadSize
2. Turn Around Time for P2 vs WorkloadSize
3. Waiting Time for P1 vs Workload Size
4. Waiting Time for P2 vs Workload Size.

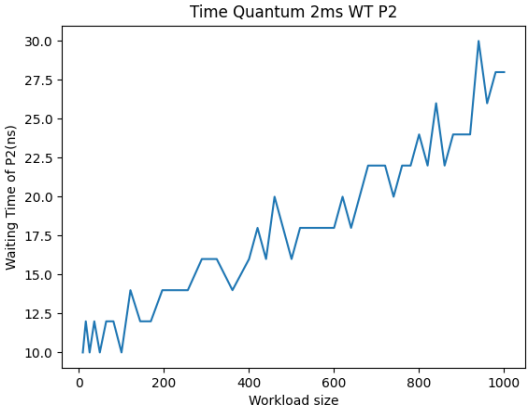
When looking at the time graphs, it is clear that there is a tendency toward an increase in the amount of time required as turnaround as well as the amount of time spent waiting as the amount of workload increases.

Case 1: When Time Quantum is 2ms









The four graphs plotted above are of:

5. Turn Around Time for P1 vs WorkloadSize
6. Turn Around Time for P2 vs WorkloadSize
7. Waiting Time for P1 vs Workload Size
8. Waiting Time for P2 vs Workload Size.

It is clearly visible that as the workload size increases, both Turn around time and Waiting time increases (for both P1 and P2).

Conclusion :

It is clear that the Turnaround time and Waiting time will both increase as the size of the workload increases.

The time needed for beginning, halting, and maybe maintaining a thread must be added to the time already needed for the computation.
We know that starting or stopping a thread consumes a lot of resources. It takes some time, and some memory is needed.
It's likely that this may lead to costs that outweigh any advantages of threads for process that we had expected.

Also, thread continue to engage in a unique kind of context swapping. Instead of improving performance capacity, adding more threads makes the kernel work even harder, which consumes more CPU time. So, threading has decreasing rewards, and doing too much degrades performance.

The duration of the process is very certainly due to the additional time needed to switch between the contexts of the two activities.

This demonstrates that the processes are not optimised by the round robin scheduling, and that it will take less time for the processes to finish if they are permitted to run individually rather than in round robin scheduling. This can be demonstrated by the fact that it will take less time for the processes to finish if they are allowed to run individually.
.