

While doing this project, my knowledge and learning about threads and processes as grown extensively. It has also grown my knowledge on the time evaluation of **creation/deletion** of threads, understanding how to **measure/compare** the performances of a thread-based parallel matrix multiplication program on multi-core workstations.

Copy-On-Write (COW) is a resource-management technique that optimizes the memory usage during process and thread creation. Unlike the traditional fork implementation where the *entire address space* of the parent process is *duplicated* for the child process, potentially wasting processor time and memory, **COW** employs a more efficient approach.

With **COW**, when a new process is created using fork, the operating system allows both the parent and the child processes to initially share the same address space. This *shared memory* is marked as read-only. If either process attempts to modify this shared memory, the operating system intercepts the operation (through a page fault) and creates a copy of the affected portion of the memory (page) for that process. This way, the modifications are isolated to the writing process, preserving the integrity and independence of each process's memory space.

The key idea behind **COW** is to delay the copying of the memory until it's necessary, which is typically when a write operation occurs. This strategy prevents the *needless duplication* of entire address spaces when a process is created, *conserving memory*, and *reducing system load*. It is particularly beneficial in scenarios where the child process might not modify the memory, such as when it calls exec to load a new program immediately after fork.

In the context of threads, which naturally share the address space of the process they belong to, **COW** doesn't apply in the same way as with fork. However, **COW** can still be relevant for threads in terms of shared resources, like memory-mapped files, where similar principles of deferring copies to the point of modification can be applied.

Overall, **COW** enhances system performance by reducing the overhead associated with process creation and memory management, making it an important technique in modern operating systems.

I have mentioned a detailed description of **COW** because, this project of ours, is based on this concept.

For this project we were to create:

- 3 executable files:
 - ***et.c***
 - ***ep.c***
 - ***para_mm.c***
- 1 non-executable file:
 - ***etime.c***
- 1 compiling file:
 - ***makefile***

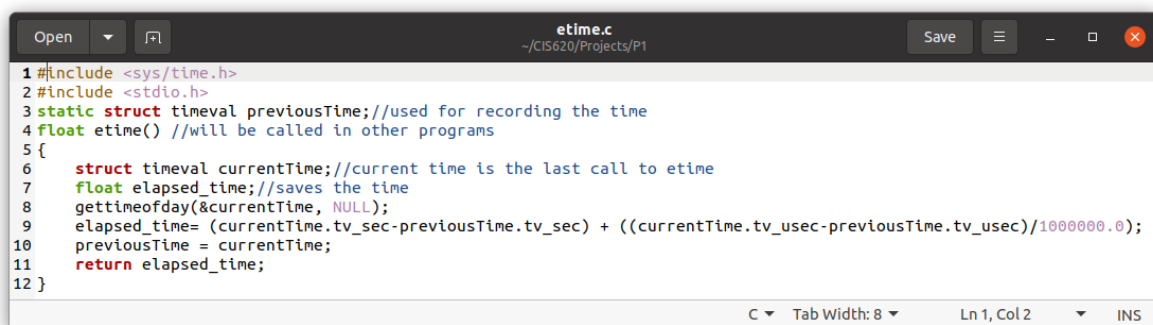
To begin the project, I started with the **etime.c** file. As per the instructions, the **etime.c** file should contain a **etime()** function which returns the elapsed time since last call to **etime()**. Also, the **etime()** function can be implemented by using **gettimeofday()** and a static local variable. As discussed in class, I started with a very simple implementation of **etime.c** in which I incorporated a simple for loop to understand the basic working of using **gettimeofday()** and the way timestamp works on system.

```
for(i = 0; i < 100; i++){
    a[i] = sqrt(a[i]);
}
```

Along with this, I learnt the how to calculate the elapsed time:

$$(stop_time.tv_sec - start_time.tv_sec) + ((stop_time.tv_usec - start_time.tv_usec)/1000000.0)$$

The divide by 1000000.0 is for conversion of microseconds to seconds. Also, take is taken with respect to **integer division**. All of this helped me understand how to write the code for the function **etime()**.



```
1 #include <sys/time.h>
2 #include <stdio.h>
3 static struct timeval previousTime; //used for recording the time
4 float etime() //will be called in other programs
5 {
6     struct timeval currentTime; //current time is the last call to etime
7     float elapsed_time; //saves the time
8     gettimeofday(&currentTime, NULL);
9     elapsed_time = (currentTime.tv_sec - previousTime.tv_sec) + ((currentTime.tv_usec - previousTime.tv_usec)/1000000.0);
10    previousTime = currentTime;
11    return elapsed_time;
12 }
```

As it was instructed to use a static local variable, I used,

static struct timeval previousTime;

Here, I have used a static variable to make sure that only the **etime()** function could manipulate this data. This means that, I am hiding this data from other files and also functions from not being able to manipulate this data at any given point in time.

Next, to talk about the **et.c** file, the instruction was to call **pthread_create()/pthread_join()** to evaluate the time for thread creation/deletion. As instructed, I was supposed to see how the dynamic memory modifications affect the performance of threads/process creation by using the **calloc()** function to **allocate/clean** the memory before starting the timer. The inputs were to be given in the command line which would be in the form of strings which represent integers, and that argument value was required to calculate the size of the buffer in the **calloc()** function because of which, there was a need to convert them to integers. In class, we were instructed to use the **atoi()** function for the same.

int size = atoi(argv[2]) * 1024;

The multiplication with 1024 is to set the size of the buffer to 1024k bytes.

Hence, the **calloc()** function shall be used as:

buf = (char *)calloc(size, 1);

Where, ***char * buf*** is a pointer.

Next, to make the string comparison to check the command line input, whether it was ***-a*** or ***-b***, which indicates the program to perform write operations on the dynamic memory before or after the child thread/process is created.

```
if(strcmp(argv[1], "z") == 0
```

The above line was used for string comparison where ***"z"*** stands for either ***-a*** or ***-b***.

This was followed by a for loop.

```
for(i = 0; i < size; i += 4096){  
    buf[i] = 'x';  
}
```

The loop was incremented 4096 times because the page size was 4096 (4K). also, if we would modify only the 1st character of every page, it would cause to duplicate the entire page. Therefore, only one update per page to check whether the system duplicates or not is enough. We need not modify every character of the page. If we sit to modify every character, we shall have to modify 1M characters which is too time consuming and is not required the job is anyways getting done as the system duplicates according to PAGES.

The code inside the for loop is to modify the buffer value.

Apart from the comparisons, we need to create and join the thread using ***pthread_create ()*** and ***pthread_join()***.

```
pthread_create (&t, NULL, dummy, argv[1]);
```

Here, dummy is a function taking an arbitrary number of arguments and returning a value that, when dereferenced, is of type void. We do the comparison of ***-a*** here. Because we want the program to create a thread at ***-a***.

Here, we use (***void * ptr***) as the passing parameter for the function ***dummy()*** and we use a pointer to point to [***argv1***].

```
pthread_join(t, NULL);
```

The above line is used to join the thread.

As we need to use the ***etime()*** function,

```
extern float etime();
```

We call this function once to start the timer and store the stopped timer into a variable and finally, print out the elapsed time.

While writing the code, proper header files need to be included to avoid unnecessary errors and warnings. Variables that are used inside the ***dummy()*** function as well as the ***main()*** function need to be declared globally.

```
char * buf;
```

```
int size;
```

The pointer **** buf*** and ***size*** are declared as global variables because they are being accessed not only by the ***main()*** function but also by the ***dummy()*** function.

```
et.c
~/CIS620/Projects/P1

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <pthread.h>
5 #include <string.h>
6
7 extern float etime();
8 char *buf;
9 int size; //global variable to compare the string
10
11 void *dummy(void *ptr){
12     char *p = (char *) ptr;
13     if (strcmp(p, "-a")==0){ //string comparison with '-a' value
14         for (int i=0; i<size; i+=4096){ //i+=4096 for incrementing it k times
15             buf[i]='x'; //modify the buffer
16         }
17     }
18 }
19
20 int main(int argc, char *argv[]){
21     pthread_t t;
22
23     if (argc != 3){
24         printf("Error!: Expected 2 arguments, but got %d\n", argc - 1); //to check the number of paramaters
25     }
26
27     size=atoi(argv[2])*1024; //conversion to string
28     buf=(char*)calloc(size, 1); //buffer allocation with calloc
29
30     etime(); //calling etime
31
32     if (strcmp(argv[1], "-b")==0){ //string comparison with '-b' value
33         for (int i=0; i<size; i+=4096){ //i+=4096 for incrementing it 4k times
34             buf[i]='x'; //modify the buffer
35         }
36     }
37
38     pthread_create(&t, NULL, dummy, argv[1]); //thread creation
39     pthread_join(t, NULL); //thread joining
40
41     float e = etime();
42     printf("elapsed time:%f\n", e);
43 }
44
```

Outputs of et:

To check the output:

- the values for argument **argv[2]** which determines the size of the dynamic memory (in K-Bytes) to be allocated are **0, 1024, 8192, 16384, 32768**.
- the values for argument **argv[1]** which are either **-a** or **-b**,
 - Option -a**: The memory is written to by the child thread after it is created. This means the thread's operation includes modifying the memory, which adds to the thread's workload. Therefore, the elapsed time reported includes both the overhead of thread creation and the time taken by the thread to perform the memory write operation. The time reported with **-a** will typically be longer because it includes the time taken by the thread to perform the memory write operation.
 - Option -b**: The memory is written to in the main thread before the child thread is created. This separates the memory operation's time from the thread's execution time, aiming to measure mostly the overhead of thread creation and deletion without the additional workload of memory modification. The elapsed time reported is more reflective of the time it takes to create and join a thread without the extra memory operation delay within the thread. The time reported with **-b** will likely be shorter as it includes only the overhead of thread creation and joining.

When a wrong input is given on the command line:

```
maradhes@brahms: ~/CIS620/Projects/P1
maradhes@brahms:~/CIS620/Projects/P1$ make
gcc -c et.c -o et.o -lpthread
gcc -c etime.c -o etime.o
gcc -o et et.o etime.o -lpthread
gcc -c ep.c -o ep.o
gcc -o ep ep.o etime.o
gcc -c para_mm.c -o para_mm.o -lpthread
gcc -o para_mm para_mm.o etime.o -lpthread
maradhes@brahms:~/CIS620/Projects/P1$ ./et -a
Error!: Expected 2 arguments, but got 1
Segmentation fault (core dumped)
maradhes@brahms:~/CIS620/Projects/P1$
```

When the correct input is given on the command line:

```
maradhes@brahms: ~/CIS620/Projects/P1
maradhes@brahms:~/CIS620/Projects/P1$ make
gcc -c et.c -o et.o -lpthread
gcc -c etime.c -o etime.o
gcc -o et et.o etime.o -lpthread
gcc -c ep.c -o ep.o
gcc -o ep ep.o etime.o
gcc -c para_mm.c -o para_mm.o -lpthread
gcc -o para_mm para_mm.o etime.o -lpthread
maradhes@brahms:~/CIS620/Projects/P1$ ./et -b 0
elapsed time:0.000234
maradhes@brahms:~/CIS620/Projects/P1$ ./et -a 0
elapsed time:0.000253
maradhes@brahms:~/CIS620/Projects/P1$ ./et -b 1024
elapsed time:0.000250
maradhes@brahms:~/CIS620/Projects/P1$ ./et -a 1024
elapsed time:0.000233
maradhes@brahms:~/CIS620/Projects/P1$ ./et -b 8192
elapsed time:0.000157
maradhes@brahms:~/CIS620/Projects/P1$ ./et -a 8192
elapsed time:0.000157
maradhes@brahms:~/CIS620/Projects/P1$ ./et -b 16384
elapsed time:0.000431
maradhes@brahms:~/CIS620/Projects/P1$ ./et -a 16384
elapsed time:0.000216
maradhes@brahms:~/CIS620/Projects/P1$ ./et -b 32768
elapsed time:0.000254
maradhes@brahms:~/CIS620/Projects/P1$ ./et -a 32768
elapsed time:0.000256
maradhes@brahms:~/CIS620/Projects/P1$
```

Next, for *ep.c* file, instructions were to invoke *fork()/waitpid()* to measure the process creation/deletion time. As instructed, here too, I was supposed to see how the dynamic memory modifications affect the performance of threads/process creation by using the *calloc()* function to **allocate/clean** the memory before starting the timer. The inputs were to be given in the command line which would be in the form of strings which represent integers, and that argument value was required to calculate the size of the buffer in the *calloc()* function because of which, there was a need to convert them to integers. In class, we were instructed to use the *atoi()* function for the same.

```
int size = atoi(argv[2]) * 1024;
```

The multiplication with 1024 is to set the size of the buffer to 1024k bytes.

Hence, the **calloc()** function shall be used as:

```
buf = (char *)calloc(size, 1);
```

Where, **char * buf** is a pointer.

Next, to make the string comparison to check the command line input, whether it was **-a** or **-b**, which indicates the program to perform write operations on the dynamic memory before or after the child thread/process is created.

```
if(strcmp(argv[1], "z") == 0
```

The above line was used for string comparison where "z" stands for either **-a** or **-b**.

This was followed by a for loop

```
for(i = 0; i < size; i += 4096){  
    buf[i] = 'x';  
}
```

The loop was incremented 4096 times because the page size was 4096 (4K). also, if we would modify only the 1st character of every page, it would cause to duplicate the entire page. Therefore, only one update per page to check whether the system duplicates or not is enough. We need not modify every character of the page. If we sit to modify every character, we shall have to modify 1M characters which is too time consuming and is not required the job is anyways getting done as the system duplicates according to PAGES.

The code inside the for loop is to modify the buffer value.

As I had to **fork()** a child process, the comparison for **-a** was done inside the child process.

The child process loop was:

```
if(cpid = fork() == 0){
```

Here, the comparison and the for loop is present.

It is closed with,

```
    exit(0);  
}
```

Without the **exit(0)** statement, the loop grows exponentially.

The parent process loop was:

```
else{  
    waitpid(cpid, NULL, 0);  
}
```

Here, **cpid**: parent waits for cpid (i.e. child process) to finish and then recycles all the resources.

NULL: it's the status.

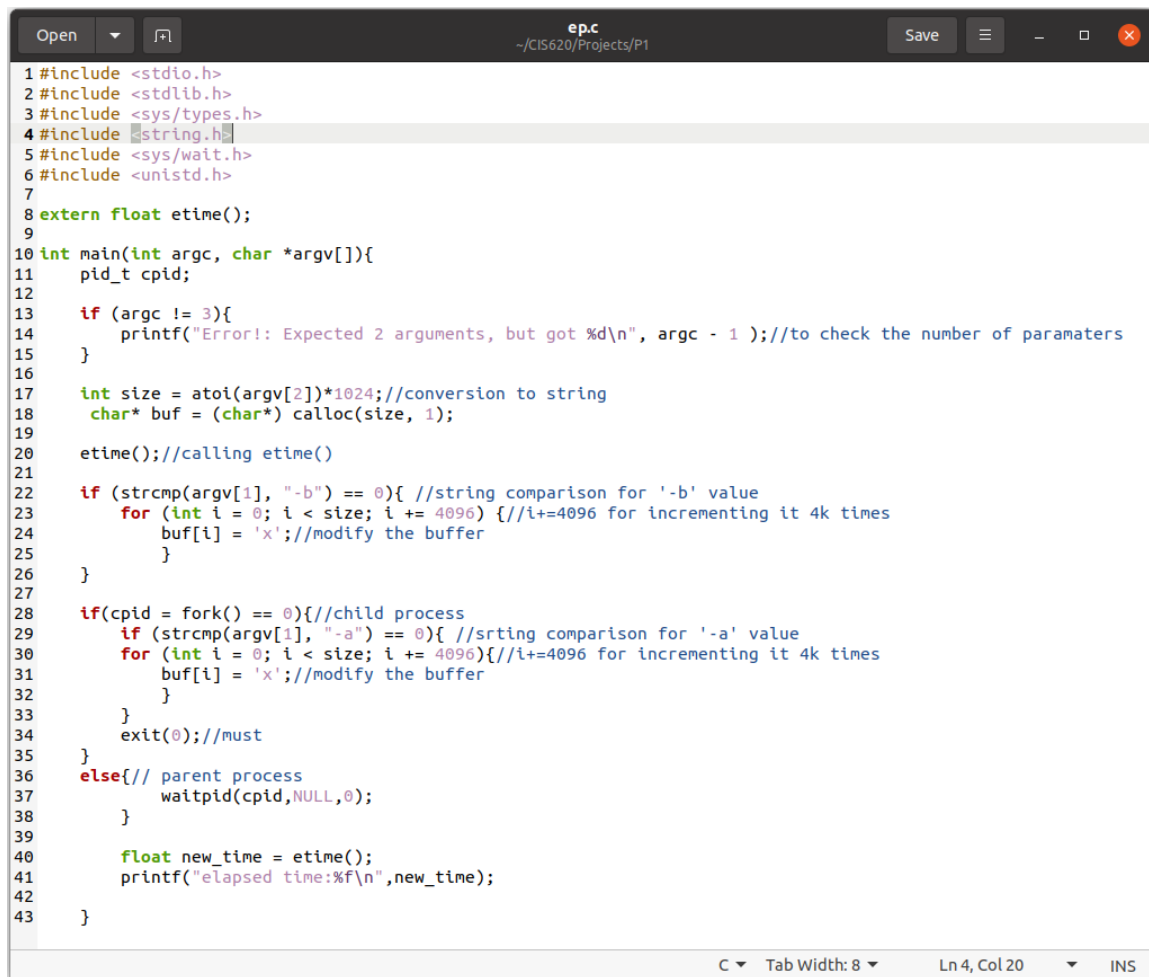
0: indicates, 'no option'.

As we need to use the **etime()** function,

```
extern float etime();
```

We call this function once to start the timer and store the stopped timer into a variable and finally, print out the elapsed time.

While writing the code, proper header files need to be included to avoid unnecessary errors and warnings.



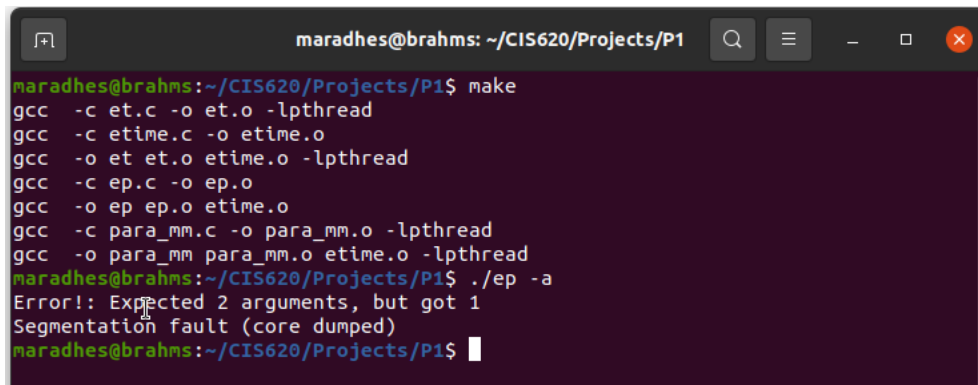
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <string.h>
5 #include <sys/wait.h>
6 #include <unistd.h>
7
8 extern float etime();
9
10 int main(int argc, char *argv[]){
11     pid_t cpid;
12
13     if (argc != 3){
14         printf("Error!: Expected 2 arguments, but got %d\n", argc - 1 );//to check the number of paramaters
15     }
16
17     int size = atoi(argv[2])*1024;//conversion to string
18     char* buf = (char*) calloc(size, 1);
19
20     etime();//calling etime()
21
22     if (strcmp(argv[1], "-b") == 0){ //string comparison for '-b' value
23         for (int i = 0; i < size; i += 4096) { //i+=4096 for incrementing it 4k times
24             buf[i] = 'x';//modify the buffer
25         }
26     }
27
28     if(cpid = fork() == 0){//child process
29         if (strcmp(argv[1], "-a") == 0){ //string comparison for '-a' value
30             for (int i = 0; i < size; i += 4096) { //i+=4096 for incrementing it 4k times
31                 buf[i] = 'x';//modify the buffer
32             }
33         }
34         exit(0);//must
35     }
36     else{// parent process
37         waitpid(cpid,NULL,0);
38     }
39
40     float new_time = etime();
41     printf("elapsed time:%f\n",new_time);
42
43 }
```

Outputs of ep:

To check the output:

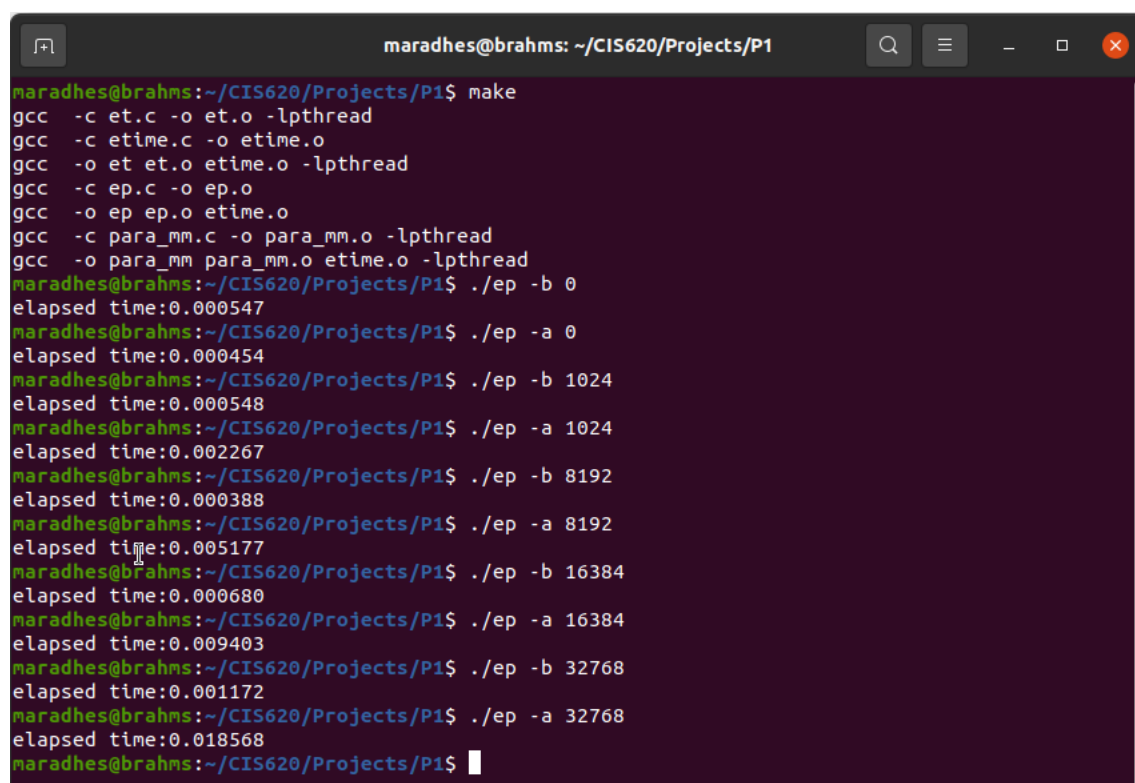
- the values for argument **argv[2]** which determines the size of the dynamic memory (in K-Bytes) to be allocated are **0, 1024, 8192, 16384, 32768**.
- the values for argument **argv[1]** which are either **-a** or **-b**.
 - Option -a:** Like *et.c*, with **-a**, the memory modification is done in the child process after it is created. This means the forked process's execution includes modifying the memory. The elapsed time will include the overhead of process creation and the time taken by the child process to perform the memory write operation. The time reported with **-a** will usually be longer since it encompasses the process creation time plus the time taken by the child process to modify the memory.
 - Option -b:** The memory is modified before the fork operation in the parent process. This approach measures the overhead of process creation and deletion more directly, as the memory operation time is not included in the child process's activity. The reported elapsed time focuses on the time it takes to fork and then wait for the child process to exit. The time reported with **-b** should be shorter, reflecting mostly the time taken to create and clean up the child process without the additional memory operation time.

When a wrong input is given on the command line:

A terminal window titled 'maradhes@brahms: ~/CIS620/Projects/P1' showing the execution of a 'make' command followed by './ep -a'. The output shows several 'gcc' compilation commands for files 'et.c', 'etime.c', 'ep.c', and 'para_mm.c'. After the compilation, the command './ep -a' is executed, resulting in an error message: 'Error!: Expected 2 arguments, but got 1' followed by 'Segmentation fault (core dumped)'.

```
maradhes@brahms:~/CIS620/Projects/P1$ make
gcc -c et.c -o et.o -lpthread
gcc -c etime.c -o etime.o
gcc -o et et.o etime.o -lpthread
gcc -c ep.c -o ep.o
gcc -o ep ep.o etime.o
gcc -c para_mm.c -o para_mm.o -lpthread
gcc -o para_mm para_mm.o etime.o -lpthread
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -a
Error!: Expected 2 arguments, but got 1
Segmentation fault (core dumped)
maradhes@brahms:~/CIS620/Projects/P1$
```

When the correct input is given on the command line:

A terminal window titled 'maradhes@brahms: ~/CIS620/Projects/P1' showing the execution of 'make' followed by a series of './ep' commands with different arguments. Each command is followed by an 'elapsed time' output. The commands and their corresponding elapsed times are: './ep -b 0' (0.000547), './ep -a 0' (0.000454), './ep -b 1024' (0.000548), './ep -a 1024' (0.002267), './ep -b 8192' (0.000388), './ep -a 8192' (0.005177), './ep -b 16384' (0.000680), './ep -a 16384' (0.009403), './ep -b 32768' (0.001172), and './ep -a 32768' (0.018568).

```
maradhes@brahms:~/CIS620/Projects/P1$ make
gcc -c et.c -o et.o -lpthread
gcc -c etime.c -o etime.o
gcc -o et et.o etime.o -lpthread
gcc -c ep.c -o ep.o
gcc -o ep ep.o etime.o
gcc -c para_mm.c -o para_mm.o -lpthread
gcc -o para_mm para_mm.o etime.o -lpthread
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -b 0
elapsed time:0.000547
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -a 0
elapsed time:0.000454
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -b 1024
elapsed time:0.000548
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -a 1024
elapsed time:0.002267
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -b 8192
elapsed time:0.000388
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -a 8192
elapsed time:0.005177
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -b 16384
elapsed time:0.000680
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -a 16384
elapsed time:0.009403
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -b 32768
elapsed time:0.001172
maradhes@brahms:~/CIS620/Projects/P1$ ./ep -a 32768
elapsed time:0.018568
maradhes@brahms:~/CIS620/Projects/P1$
```

Now, to point the difference between the elapsed time outputs between *et.c* and *ep.c*,

- **Using *-a*** typically results in longer elapsed times because the child thread or process must perform additional work (memory writing) before completing. This extra work adds to the total time measured from the parent's perspective.
- **Using *-b*** aims to isolate and measure just the overhead of creating and synchronizing (joining or waiting) the child thread or process. Since the memory writing is done beforehand (in the case of threads) or is not included in the child's execution time (in

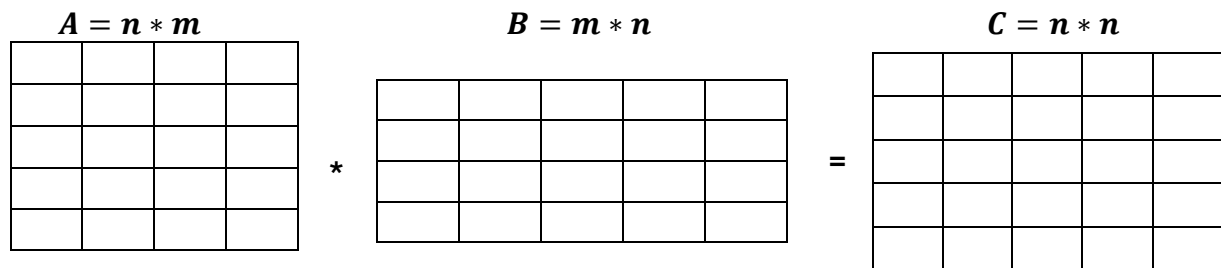
the case of processes), the elapsed time should primarily reflect the creation and deletion overhead.

To conclude:

The difference in timing between using $-a$ and $-b$ provides insight into how memory operations affect thread/process performance. Specifically, it helps in understanding the overhead introduced by such operations when they are part of the thread or process workload versus when they are executed outside of that context. This comparison can be useful for performance analysis and optimization, especially in systems where memory operations and thread/process management are critical components.

Following this, next is **para_mm.c** file. The instructions here were the need to measure the computation time for multiplying two **400X400** matrices. The program can split the task to a few threads and each thread just handles a portion of the matrix. The number of computational threads is given through the argument **argv[1]**. The additional instruction was to use semaphore synchronization to let the main thread know that all the **m** computational threads have done the computation and hence it can stop the timer.

As instructed in class for matrix multiplication,



```
for(i = 0; i < N; i++){
    for(j = 0; j < N; j++){
        c[i][j] = 0;
        for(k = 0; k < N; k++){
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

To consider the case of threads, we consider each row as a thread. Also, we only partition the 1st matrix i.e., matrix **A**. Therefore, the outer for loop needs to be partitioned.

To partition the thread, I had to change the loop to start and end at a different time.

start = $\frac{N}{m} * tid$
end = $\frac{N}{m} * (tid + 1)$

To use the semaphore synchronization,

Semaphore declaration : **sem_t sema;**

Semaphore initialization : **sem_init(&sema, 0, 0);**

Posting a semaphore : **sem_post(&sema);**

Waiting a semaphore : **sem_wait(&sema);**

The semaphore waiting is done in the **main()** function inside a for loop:

```
for(i = 0; i < m; i ++){  
    sem_wait(&sema);  
}
```

While setting the semaphore to wait, care needs to be taken to position it after the thread creation else, the process will not be KILLED. And the terminal will not produce any further output causing an infinite loop for the semaphore to wait.

Here, **m** is the command line argument input **argv[1]**. That is also the number of times the loop must be iterated to create the required number of threads as per the input.

The initialization of the semaphore,

```
sem_init(&sema, 0, 0);
```

This is done in the **main()** function.

The semaphore posting is done in the **worker()** function. Here, worker is a function taking an arbitrary number of arguments and returning a value that, when dereferenced, is of type void. Here, we use (**void * arg**) as the passing parameter for the function **worker()** and we use a pointer to point to **tid**. Inside this **worker()** function, I have calculated the matrix multiplication. The semaphore posting is done immediately after the matrix multiplication.

Inside the **main()** function, there are two for loops to initialize the two matrices, **A** and **B**.

```
for(j = 0; j < N; j ++){  
    for(k = 0; k < N; k ++){  
        a[j][k] = 1;  
        b[j][k] = 1;  
    }  
}
```

I also have a for loop for thread creation.

```
for(i = 0; i < m; i ++){  
    pthread_create (&t[i], NULL, worker, (void *)i);  
}
```

In the above thread creation, care needs to be taken to maintain difference between the address of **i** or else, “**Error**” shall be encountered.

As we need to use the **etime()** function,

```
extern float etime();
```

We call this function once to start the timer and store the stopped timer into a variable and finally, print out the elapsed time.

In the **main()** function,

While writing the code, proper header files need to be included to avoid unnecessary errors and warnings. Variables that are used inside the **worker()** function as well as the **main()** function need to be declared globally.

```
sem_t sema;  
int m;  
int N;  
int c[400][400], b[400][400], c[400][400];
```

The semaphore *sem_t sema*, integer *m*, arrays *c[400][400]*, *b[400][400]*, *a[400][400]*, and integer *N* are declared as global variables because they are being accessed not only by the *main()* function but also by the *worker()* function.

Also, for the count of number of threads which shall be given from the command line, it shall be in the form of string. We need to convert it to an integer.

m = atoi(argv[1]);



```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <pthread.h>
5 #include <string.h>
6 #include <semaphore.h>
7
8 extern float etime();
9 sem_t sema; // semaphore declaration
10 int m; // # of threads that has to be created
11 int N = 400; // fixed matrix size
12 int c[400][400], a[400][400], b[400][400]; // declaring all 3 matrices
13
14 void *worker(void *arg){
15     long tid = (long) arg;
16     int start = (N/m)*tid, end = (N/m)*(tid+1); // initializing the start and the end of matrix 'a'
17
18     for (int i = start; i < end; i++) { // 'i' to iterate through the rows
19         for (int j = 0; j < N; j++) { // 'j' to iterate through the column
20             c[i][j] = 0; // matrix to store the final matrix
21             for (int k = 0; k < N; k++) { // 'k' to iterate through the position of the element in matrix 'c'
22                 c[i][j] += a[i][k] * b[k][j]; // matrix multiplication
23             }
24         }
25     }
26     sem_post(&sema); // semaphore posting
27 }
28
29 int main(int argc, char *argv[]){
30     m = atoi(argv[1]); // conversion to string
31     pthread_t t[m];
32     long i;
33     sem_init(&sema, 0, 0); // initializing semaphore
34
35     if (argc != 2){
36         printf("Error! Expected 1 argument, but got %d\n", argc - 1); // to check the number of parameters
37     }
38
39     etime(); // calling etime
40
41     for (int j = 0; j < N; j++) { // 'j' to iterate through the column
42         for (int k = 0; k < N; k++) { // 'k' to iterate through the position of the element in matrix 'c'
43             a[j][k] = 1;
44             b[j][k] = 1; // initializing the matrices to '1'
45         }
46     }
47
48     for (i = 0; i < m; i++) {
49         pthread_create(&t[i], NULL, worker, (void*)i); // creating thread
50     }
51
52     for (i = 0; i < m; i++) {
53         sem_wait(&sema); // semaphore waiting
54     }
55
56     float new_time = etime();
57     printf("Elapsed time:%f\n", new_time);
58 }

```

Outputs of para_mm:

To check the output:

- the values for argument *argv[1]* which determines the number of threads should be allocated with the following values: **1, 2, 4, 8, 16**.

While checking the output of *para_mm.c* file, the time values should be decrementing to almost half the time that shall be required for the smaller input of the number of threads.

if elapsed time at thread count 1 = t;

then,

elapsed time at thread count 2 = $\frac{t}{2}$;

elapsed time at thread count 4 = $\frac{t}{4}$;

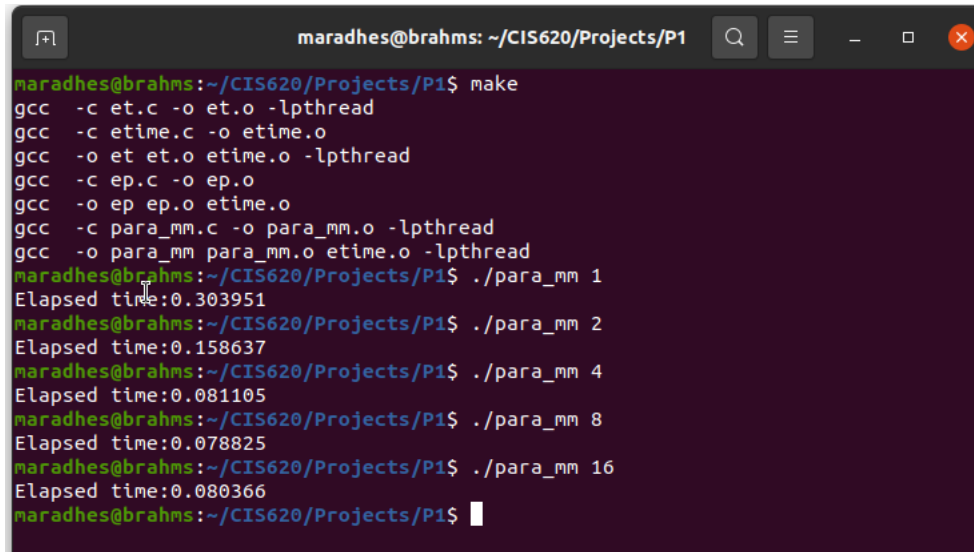
elapsed time at thread count 8 = $\frac{t}{8}$;

When a wrong input is given on the command line:



```
maradhes@brahms: ~/CIS620/Projects/P1
maradhes@brahms:~/CIS620/Projects/P1$ make
gcc -c et.c -o et.o -lpthread
gcc -c etime.c -o etime.o
gcc -o et et.o etime.o -lpthread
gcc -c ep.c -o ep.o
gcc -o ep ep.o etime.o
gcc -c para_mm.c -o para_mm.o -lpthread
gcc -o para_mm para_mm.o etime.o -lpthread
maradhes@brahms:~/CIS620/Projects/P1$ ./para_mm 12 12
Error!: Expected 1 argument, but got 2
Elapsed time:0.087804
maradhes@brahms:~/CIS620/Projects/P1$
```

When the correct input is given on the command line:



```
maradhes@brahms: ~/CIS620/Projects/P1
maradhes@brahms:~/CIS620/Projects/P1$ make
gcc -c et.c -o et.o -lpthread
gcc -c etime.c -o etime.o
gcc -o et et.o etime.o -lpthread
gcc -c ep.c -o ep.o
gcc -o ep ep.o etime.o
gcc -c para_mm.c -o para_mm.o -lpthread
gcc -o para_mm para_mm.o etime.o -lpthread
maradhes@brahms:~/CIS620/Projects/P1$ ./para_mm 1
Elapsed time:0.303951
maradhes@brahms:~/CIS620/Projects/P1$ ./para_mm 2
Elapsed time:0.158637
maradhes@brahms:~/CIS620/Projects/P1$ ./para_mm 4
Elapsed time:0.081105
maradhes@brahms:~/CIS620/Projects/P1$ ./para_mm 8
Elapsed time:0.078825
maradhes@brahms:~/CIS620/Projects/P1$ ./para_mm 16
Elapsed time:0.080366
maradhes@brahms:~/CIS620/Projects/P1$
```

The observed changes in execution time for matrix multiplication in *para_mm.c* as the number of threads increases can be attributed to the benefits and limitations of parallel processing and your CPU's specific architecture. Here's a concise overview and conclusion:

The overview:

- **Parallel Processing Efficiency:** Execution time decreases significantly when increasing the number of threads from 1 to 8, leveraging parallel computation across multiple CPU cores to perform tasks simultaneously. This efficiency is due to the workload being evenly distributed among the available threads, reducing overall

execution time when the number of threads aligns with the number of available CPU cores or logical processors (thanks to hyper-threading).

- **CPU Configuration:** Your CPU, with 4 cores and 2 threads per core, supports 8 hardware threads concurrently. This setup allows for optimal parallel processing with up to 8 threads, aligning with the observed performance improvements.
- **Diminishing Returns Beyond Optimal Thread Count:** When the number of threads exceeds 8 (e.g., `./para_mm 16`), the execution time slightly increases due to the limitations of hyper-threading efficiency, increased thread management overhead, resource contention, and the principle of diminishing returns as per Amdahl's Law.

Conclusion:

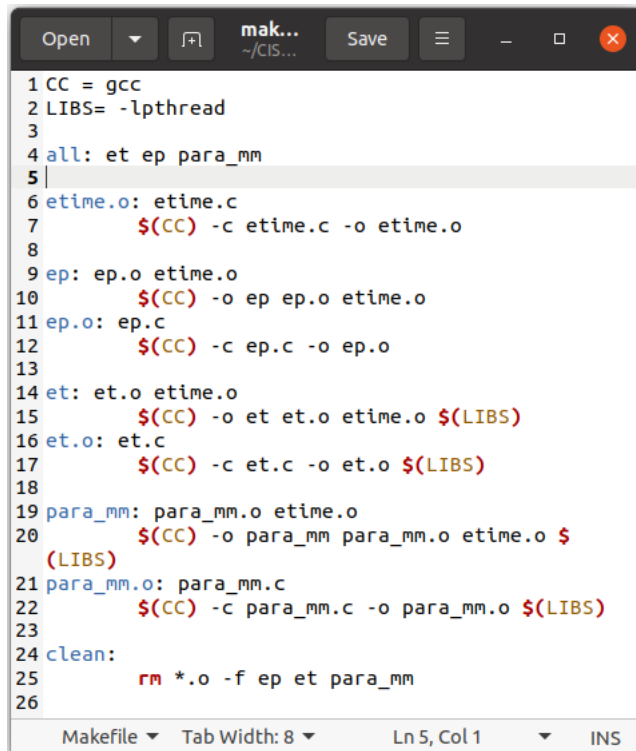
The performance gains from increasing the number of threads in `para_mm.c` illustrate the effectiveness of parallel processing up to the point where the number of threads matches the hardware's capacity to run them concurrently. The optimal performance is observed with 8 threads, corresponding to your CPU's ability to handle 8 hardware threads efficiently. Beyond this, the slight increase in execution time with 16 threads highlights the limits of hyper-threading and the increasing impact of overheads and resource contention, underscoring the importance of aligning the number of threads with the CPU's capabilities for best performance.

The CPU configuration:

```
maradhes@brahms:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
Address sizes:          39 bits physical, 48 bits virtual
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):              1
NUMA node(s):          1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                 158
Model name:             Intel(R) Xeon(R) CPU E3-1245 v6 @ 3.70GHz
Stepping:               9
CPU MHz:               3700.000
CPU max MHz:           4100.0000
CPU min MHz:           800.0000
BogoMIPS:              7399.70
Virtualization:         VT-x
L1d cache:             128 KiB
L1i cache:             128 KiB
L2 cache:              1 MiB
L3 cache:              8 MiB
NUMA node0 CPU(s):     0-7
Vulnerability Gather data sampling: Mitigation: Microcode
Vulnerability Itlb multihit:       KVM: Mitigation: VMX disabled
Vulnerability L1trf:              Mitigation: PTE Inversion; VMX conditional cache flushes, SMT vulnerable
Vulnerability Mds:                Mitigation: Clear CPU buffers; SMT vulnerable
Vulnerability Meltdown:           Mitigation: PTI
Vulnerability Mmio stale data:     Mitigation: Clear CPU buffers; SMT vulnerable
Vulnerability Retbleed:           Mitigation: IBRS
Vulnerability Spec rstack overflow: Not affected
Vulnerability Spec store bypass:   Mitigation: Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:         Mitigation: usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:         Mitigation: IBRS, IBPB conditional, STIBP conditional, RSB filling, PBSRB-eIBRS Not affected
Vulnerability Srbds:              Mitigation: Microcode
Vulnerability Tsx async abort:     Mitigation: TSX disabled
Flags:                          fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts apti mnx fpxr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art a
rch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 sse3 sdbg fma cx16 xtpr pdcm pcid sse4_1
sse4_2 xzapic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow v
nmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bti1 avx2 smep bti2 erms invpcid mpx rdseed adx snap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida
arat pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear flush_lid arch_capabilities
```

Finally, the **makefile**. As instructed in class, we create this file so that it makes it easy to compile all the files with just one command, **make**. In this file, we have the code to compile all the files, viz., **etime.c**, **et.c**, **ep.c**, and **para_mm.c**. the command for that is **gcc**. Also, as we are dealing with threads, **-lpthread** is used for thread compilation. This file makes sure to create all the **.o** files that are required, viz., **etime.o**, **et.o**, **ep.o**, and **para_mm.o**. Also, the command the clean all the **.o** files is **clean:rm *.o -f ep et para_mm** files With all the rules explained in class, all the dependencies that need to be taken care of, I made my **makefile**.

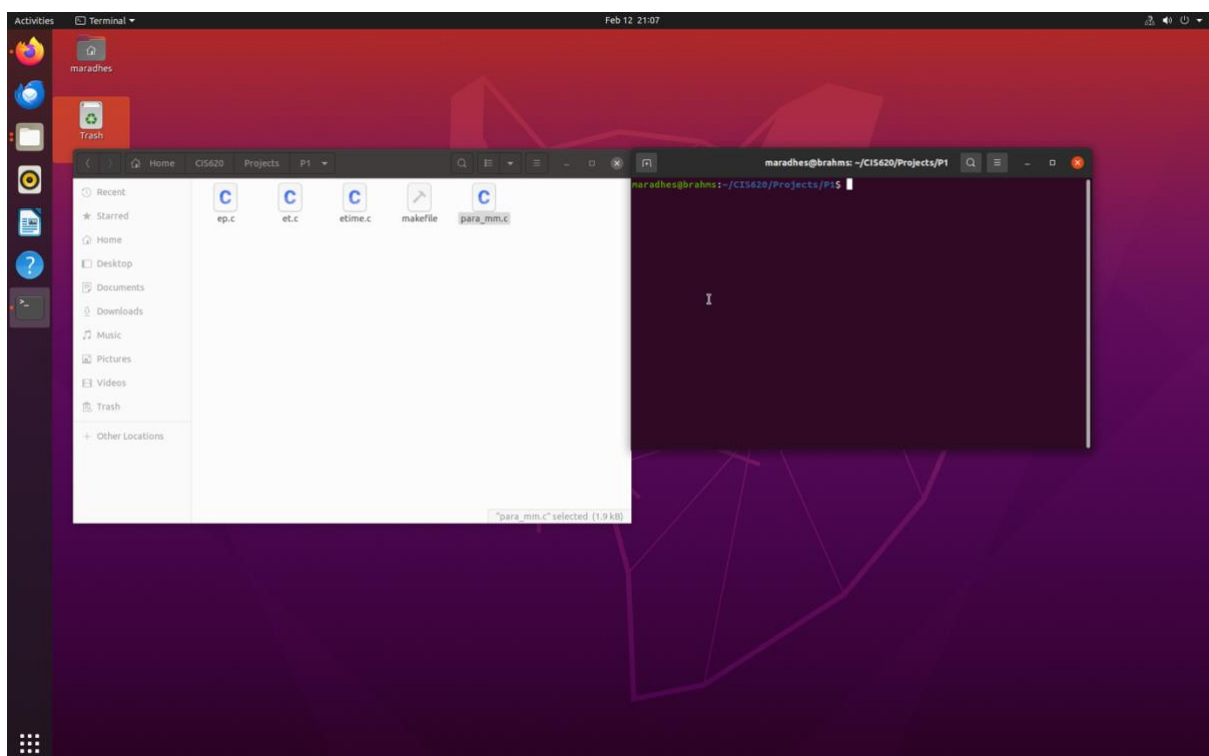
-f Stands for "force." This option tells **rm** to ignore non-existent files and arguments, and it will not prompt for confirmation when deleting files. This is particularly useful in clean targets to ensure that the make process doesn't halt or exit with an error if some of the files listed to be removed do not exist.



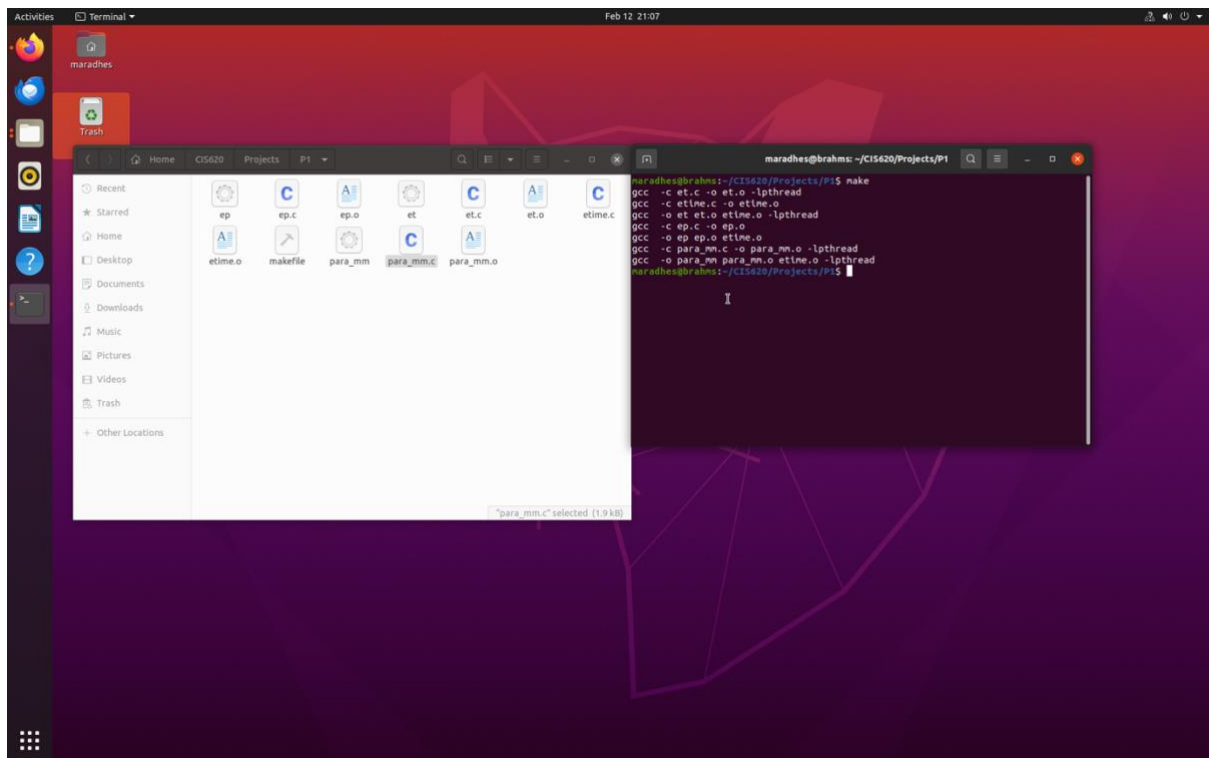
```
1 CC = gcc
2 LIBS= -lpthread
3
4 all: et ep para_mm
5
6 etime.o: etime.c
7     $(CC) -c etime.c -o etime.o
8
9 ep: ep.o etime.o
10     $(CC) -o ep ep.o etime.o
11 ep.o: ep.c
12     $(CC) -c ep.c -o ep.o
13
14 et: et.o etime.o
15     $(CC) -o et et.o etime.o $(LIBS)
16 et.o: et.c
17     $(CC) -c et.c -o et.o $(LIBS)
18
19 para_mm: para_mm.o etime.o
20     $(CC) -o para_mm para_mm.o etime.o $(LIBS)
21 para_mm.o: para_mm.c
22     $(CC) -c para_mm.c -o para_mm.o $(LIBS)
23
24 clean:
25     rm *.o -f ep et para_mm
26
```

Makefile ▾ Tab Width: 8 ▾ Ln 5, Col 1 ▾ INS

On the terminal, before I make the files,



On the terminal, after I make the files,



During the entire process of testing and debugging, in the *para_mm.c* file, I accidentally put

```
for(i = 0; i < m; i++){  
    sem_wait(&sema);  
}
```

Before the creation of the pthread. This caused my process to not get KILLED. Hence, on the terminal, the output screen continued to be displayed and not terminate. Despite the fact I used, `^C ^C ^C` multiple times, the process did not terminate. As this was discussed in the last class, I understand that the process was not getting killed. That is when I rechecked my code. I figured out that I did put,

```
for(i = 0; i < m; i++){  
    sem_wait(&sema);  
}
```

In the wrong place. Because I put it before the thread creation, the semaphore was waiting for the thread to get created, it entered an infinite loop which would never terminate by itself.

As soon as I realized, I put,

```
for(i = 0; i < m; i++){  
    sem_wait(&sema);  
}
```

After the creation of the thread,

```
for(i = 0; i < m; i++){  
    pthread_create(&t[i], NULL, worker, (void *)i);  
}
```

Apart from this, initially, I did not initialize the matrices *A* and *B*.

```
for(j = 0; j < N; j++){
```

```
    for(k = 0; k < N; k ++){  
        a[j][k] = 1;  
        b[j][k] = 1;  
    }  
}
```

Also, I did not set:

```
for(i = 0; i < m; i ++){  
    sem_wait(&sema);  
}
```

Because of which, the output of the time, which should have been decreasing to half was doubling up. Later, I realised, without setting up the *sem_wait*(&*sema*); the process will not wait for the threads to be created and hence the time was doubling up.

This made my code function correctly.

Finally, my entire project functions completely. (***WORKS COMPLETELY***)