



Faculty of Engineering and Technology
Electrical and Computer Engineering Department
Operating Systems-ENCS3390
Programming Task #1

Name :	Qusay Taradeh
ID No. :	1212508
Instructor :	Bashar Tahayna
Section No. :	4

Summary of Processes, Joinable Threads , Detached Threads and Parallelism

Processes

A process is an instance of a program that is being executed. Each process has an execution time and performs a specific part of the task. Therefore, we use processes to reduce the overall time required to complete the task. Every process has a parent process that holds its child process ID. We create processes, known as "child processes," using the fork function, where each child process requires one fork. Finally, the parent process must wait for its child process to finish its part of the task before the entire task is considered complete. However, if the parent process does not wait for its child process, the task will not be completed as expected.

Threads

A thread is a lightweight process that shares resources with other threads to perform a given task. It can operate concurrently or in parallel. There are two types of threads: joinable threads and detached threads.

Joinable Threads

A joinable thread is a thread that can be joined by another thread. When a thread joins another thread, it waits for the second thread to complete execution before proceeding. Joinable threads are particularly useful for tasks that require multiple parts to be completed successfully before the overall task can be considered finished.

Detached Threads

A detached thread is a thread that cannot be joined by another thread. When a detached thread terminates, its resources are automatically released back to the system. Detached threads are typically used when the parent thread does not need to wait for the child thread to complete its task.

Parallelism

The concept of parallelism is effectively applied in matrix multiplication, allowing us to divide the task into smaller parts that can be executed independently using processes or threads. This approach significantly reduces the overall execution time compared to a sequential execution. Multiprocessing and multithreading serve as practical implementations of parallelism, enabling efficient matrix multiplication.

Native Case:

There is in two pictures the code of native C code to initializing the matrices and the time variables that calculate the time for matrix multiplication of matrices:

```
OS_1212508 > C main.c > main(int, char *[])
42     k++;
43     }
44 }
45
46 printf("\nID * Year Matrix\n");
47 k = 0;
48 for (i = 0; i < N; i++) // loop to fill ID * Year matrix
49 {
50     for (j = 0; j < N; j++)
51     {
52         if (k > 9)
53             k = 0;
54
55         idXyearMatrix[i][j] = idXyear[k];
56         printf("%d ", idXyearMatrix[i][j]);
57         k++;
58     }
59 }
60
61 printf("\nResult Matrix\n");
62 start = clock(); // start measuring time
63 for (i = 0; i < N; i++) // loop for multiplication as a native without any process or thread
64 {
65     for (j = 0; j < N; j++)
66     {
67         resultMatrix[i][j] = 0;
68         for (k = 0; k < N; k++)
69         {
70             resultMatrix[i][j] += (idMatrix[i][k] * idXyearMatrix[k][j]);
71         }
72     }
73 }
74
75 end = clock(); // end measuring time
76 tm = (float)(end - start) / CLOCKS_PER_SEC;
77 printf("\n\nTime is: %f\n\n", tm);
78
79 //////////////////////////////////////////////////

OS_1212508 > C main.c > main(int, char *[])
14 #define N 100
15
16 void childProcess(int idMatrix[N][N], int idXyearMatrix[N][N], int resultMatrix[N][N]);
17
18 int main(int argc, char *argv[])
19 {
20     clock_t start, end; // time variables
21     float tm; // tm: time calculated
22     pid_t pid; // pid : process id
23     int pipeFD[4]; // pipeFD : pipe file discovery for IPC
24     int i, j, k = 0; // variables for loops
25     int id[7] = {1, 2, 1, 2, 5, 0, 8}; // my ID represented as digits in array
26     int idXyear[10] = {2, 4, 2, 8, 6, 5, 3, 5, 2, 4}; // idXyear = id * year = 1212508 * 2003 = 2428653524
27     int idMatrix[N][N]; // matrix of my ID number
28     int idXyearMatrix[N][N]; // matrix of my ID multiplied by my birth year
29     int resultMatrix[N][N]; // matrix wich is the result of multiplication between ID matrix and ID * year matrix
30
31     //Native C code for implementing and multiplication
32     printf("\nID Matrix\n");
33     for (i = 0; i < N; i++) // loop to fill ID matrix
34     {
35         for (j = 0; j < N; j++)
36         {
37             if (k > 6)
38                 k = 0;
39
40             idMatrix[i][j] = id[k];
41             printf("%d ", idMatrix[i][j]);
42             k++;
43         }
44     }
45
46     printf("\nID * Year Matrix\n");
47     k = 0;
48     for (i = 0; i < N; i++) // loop to fill ID * Year matrix
49     {
50         for (j = 0; j < N; j++)
51         {
52             if (k > 9)
53                 k = 0;
54
55             idXyearMatrix[i][j] = idXyear[k];
56             printf("%d ", idXyearMatrix[i][j]);
57             k++;
58         }
59     }
60
61     printf("\nResult Matrix\n");
62     start = clock(); // start measuring time
63     for (i = 0; i < N; i++) // loop for multiplication as a native without any process or thread
64     {
65         for (j = 0; j < N; j++)
66         {
67             resultMatrix[i][j] = 0;
68             for (k = 0; k < N; k++)
69             {
70                 resultMatrix[i][j] += (idMatrix[i][k] * idXyearMatrix[k][j]);
71             }
72             printf("%d ", resultMatrix[i][j]);
73         }
74     }
75     end = clock(); // end measuring time
76     tm = (float)(end - start) / CLOCKS_PER_SEC;
77     printf("\n\nTime is: %f\n\n", tm);
78
79     //////////////////////////////////////////////////
```

And below in Terminal the time that taken to do the task without process or thread which is equal 0.005664s which is 5.664ms and throughput is $176.55s^{-1}$

```
75     end = clock(); // end measuring time
76     tm = (float)(end - start) / CLOCKS_PER_SEC;
77     printf("\n\nTime is: %f\n\n", tm);
78
79     //
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
538 1076 538 1076 538 2152 1614 1345 807 1345 538 1076 538 1076 538 2152 1614 1345 807 1345 538 1076 538 1076 538 2152 1614 1345 807 1345 538
2152 1614 1345 807 1345 538 1076 538 1076 538 2152 1614 1345 807 1345 538 1076
Time is: 0.005664
[1] + Done                                "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-glnmbdn.oij" 1>"/tmp/Microsof
t3dji.izm"
qusay@qusay-VirtualBox:~/CProjects$
```

So we notice that time complexity is $O(n^3)$, so to solve this problem and actually in our task that multiply matrices that takes a long time specially when number of rows and cols is increased. Furthermore we go to part one of task that we will use processes to manage the execution time and make it less than native way coding the task.

Process Management:

From below snapshots we used 2 child processes to do the multiplication and we notice that the time reduced to 0.002361s which is 2.361ms instead of 5.664ms and this shows the importance of multiprocesses, and the throughput equals $423.55s^{-1}$

```
OS_1212508 > C main.c > main(int, char* [])
01 //////////////////////////////////////////////////
02 for(i = 0; i < 2; i++)
03     if (pipe(pipeFD[i]) == -1) // error handling when creating pipe failure and this return 2
04     {
05         printf("Can't create pipe!\n");
06         return 2;
07     }
08
09 gettimeofday(&start, NULL);
10 pid = fork(); // forking the process
11 if (pid == -1) // error handling when fork failure and this return 1
12 {
13     printf("An error occurred with fork!\n");
14     return 1;
15 }
16 if (pid == 0)
17 {
18     //Child process 1
19
20     close(pipeFD[0][0]); //close read of process 1
21     close(pipeFD[1][0]); //close read of process 2
22     close(pipeFD[1][1]); //close write of process 2
23
24     childProcess(idMatrix, idYearMatrix, resultMatrix);
25
26     if(write(pipeFD[0][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 4; } // writing and handling error
27
28     close(pipeFD[0][1]); //close writing in process 1
29     return 0;
30 }
31
32 pid_t pid2 = fork();
33 if (pid2 == -1) // error handling when fork failure and this return 1
34 {
35     printf("An error occurred with fork!\n");
36 }
```

```
OS_1212508 > C main.c > main(int, char* [])
92 pid_t pid2 = fork();
93 if (pid2 == -1) // error handling when fork failure and this return 1
94 {
95     printf("An error occurred with fork!\n");
96     return 1;
97 }
98
99 if (pid2 == 0)
100 {
101     //Child process 2
102     close(pipeFD[1][0]); //close read of process 2
103     close(pipeFD[0][0]); //close read of process 1
104     close(pipeFD[0][1]); //close write of process 1
105
106     childProcess(idMatrix, idYearMatrix, resultMatrix);
107
108     if(write(pipeFD[1][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // writing and handling error
109
110     close(pipeFD[1][1]); //close writing in process 2
111     return 0;
112 }
113 //Parent process
114 //start = clock();
115 close(pipeFD[0][1]); //close write of process 1
116 close(pipeFD[1][1]); //close write of process 2
117
118 gettimeofday(&end, NULL);
119
120 waitpid(pid, NULL, 0); // waiting for process 1 to finish operation
121 waitpid(pid2, NULL, 0); // waiting for process 2 to finish operation
122
123 if(read(pipeFD[0][0], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // reading from process 1
124
125 if(read(pipeFD[1][0], resultMatrix + N/2, sizeof(int)*N*N/2) == -1) { return 5; } // reading from process 2
126
```

```
OS_1212508 > C main.c > main(int, char*[])
110     close(pipeFD[1][1]); //close writing in process 2
111     return 0;
112 }
113 //Parent process
114 //start = clock();
115 close(pipeFD[0][1]); //close write of process 1
116 close(pipeFD[1][1]); //close write of process 2
117
118 gettimeofday(&end, NULL);
119
120 waitpid(pid, NULL, 0); // waiting for process 1 to finish operation
121 waitpid(pid2, NULL, 0); // waiting for process 2 to finish operation
122
123 if(read(pipeFD[0][0], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // reading from process 1
124
125 if(read(pipeFD[1][0], resultMatrix + N/2, sizeof(int)*N*N/2) == -1) { return 5; } // reading from process 2
126
127 close(pipeFD[0][0]); //close read of process 1
128 close(pipeFD[1][0]); //close read of process 2
129 //printMatrix(resultMatrix);
130
131
132 tm = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
133 printf("\n\nTime is: %fs\n\n", tm);
134
135 return 0;
136 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Time is: 0.002361s

[1] + done                               "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-lv2klwtz.2rq" 1>"/tmp/Microsoft-MIEngine-Out-hhb
wyoqlm.j4t"
qusay@qusay-VirtualBox:~/CProjects$
```

Now we want to increase number of child processes to 4 instead of 2 child processes and noting the difference in time and throughput.

```
OS_1212508 > C main.c > main(int, char*[])
61 ///////////////////////////////////////////////////end of initializing/////////////////////////////////
62 for(i = 0; i < 4; i++)
63 {
64     if (pipe(pipeFD[i]) == -1) // error handling when creating pipe failure and this return 2
65     {
66         printf("Can't create pipe!\n");
67         return 2;
68     }
69
70 gettimeofday(&start, NULL);
71 pid = fork(); // forking the process
72 if (pid == -1) // error handling when fork failure and this return 1
73 {
74     printf("An error occured with fork!\n");
75     return 1;
76 }
77 if (pid == 0)
78 {
79     //Child process 1
80     close(pipeFD[0][0]); //close read of process 1
81     close(pipeFD[1][0]); //close read of process 2
82     close(pipeFD[1][1]); //close write of process 2
83     close(pipeFD[2][0]); //close read of process 3
84     close(pipeFD[2][1]); //close write of process 3
85     close(pipeFD[3][0]); //close read of process 4
86     close(pipeFD[3][1]); //close write of process 4
87
88     childProcess(idMatrix, idYearMatrix, resultMatrix);
89
90     if(write(pipeFD[0][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 4; } // writing and handling error
91     close(pipeFD[0][1]); //close writing in process 1
92     return 0;
93 }
94
95 pid_t pid2 = fork();
```

```

OS_1212508 > C main.c > main(int, char * [])
95 pid_t pid2 = fork();
96 if (pid2 == -1) // error handling when fork failure and this return 1
97 {
98     printf("An error occured with fork!\n");
99     return 1;
100 }
101
102 if (pid2 == 0)
103 {
104     //Child process 2
105     close(pipeFD[1][0]); //close read of process 2
106     close(pipeFD[0][0]); //close read of process 1
107     close(pipeFD[0][1]); //close write of process 1
108     close(pipeFD[2][0]); //close read of process 3
109     close(pipeFD[2][1]); //close write of process 3
110     close(pipeFD[3][0]); //close read of process 4
111     close(pipeFD[3][1]); //close write of process 4
112
113     childProcess(idMatrix, idYearMatrix, resultMatrix);
114
115     if(write(pipeFD[1][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // writing and handling error
116
117     close(pipeFD[1][1]); //close writing in process 2
118     return 0;
119 }
120
121 pid_t pid3 = fork();
122 if (pid3 == -1) // error handling when fork failure and this return 1
123 {
124     printf("An error occured with fork!\n");
125     return 1;
126 }
127
128 if (pid3 == 0)
129 {

```

```

OS_1212508 > C main.c > main(int, char * [])
127
128 if (pid3 == 0)
129 {
130     //Child process 3
131     close(pipeFD[2][0]); //close read of process 3
132     close(pipeFD[0][0]); //close read of process 1
133     close(pipeFD[0][1]); //close write of process 1
134     close(pipeFD[1][0]); //close read of process 2
135     close(pipeFD[1][1]); //close write of process 2
136     close(pipeFD[3][0]); //close read of process 4
137     close(pipeFD[3][1]); //close write of process 4
138
139     childProcess(idMatrix, idYearMatrix, resultMatrix);
140
141     if(write(pipeFD[2][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // writing and handling error
142
143     close(pipeFD[2][1]); //close writing in process 3
144     return 0;
145 }
146
147 pid_t pid4 = fork();
148 if (pid4 == -1) // error handling when fork failure and this return 1
149 {
150     printf("An error occured with fork!\n");
151     return 1;
152 }
153
154 if (pid4 == 0)
155 {
156     //Child process 4
157     close(pipeFD[3][0]); //close read of process 4
158     close(pipeFD[0][0]); //close read of process 1
159     close(pipeFD[0][1]); //close write of process 1
160     close(pipeFD[1][0]); //close read of process 2
161     close(pipeFD[1][1]); //close write of process 2

```

```

OS_1212508 > C main.c > main(int, char *[])
154 if (pid4 == 0)
155 {
156     //Child process 4
157     close(pipeFD[3][0]); //close read of process 4
158     close(pipeFD[0][0]); //close read of process 1
159     close(pipeFD[0][1]); //close write of process 1
160     close(pipeFD[1][0]); //close read of process 2
161     close(pipeFD[1][1]); //close write of process 2
162     close(pipeFD[2][0]); //close read of process 3
163     close(pipeFD[2][1]); //close write of process 3
164
165     childProcess(idMatrix, idYearMatrix, resultMatrix);
166
167     if(write(pipeFD[3][1], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // writing and handling error
168
169     close(pipeFD[3][1]); //close writing in process 4
170     return 0;
171 }
172
173 //Parent process
174 //start = clock();
175
176 close(pipeFD[0][1]); //close write of process 1
177 close(pipeFD[1][1]); //close write of process 2
178 close(pipeFD[2][1]); //close write of process 3
179 close(pipeFD[3][1]); //close write of process 4
180
181 gettimeofday(&end, NULL);
182
183 waitpid(pid, NULL, 0); // waiting for process 1 to finish operation
184 waitpid(pid2, NULL, 0); // waiting for process 2 to finish operation
185 waitpid(pid3, NULL, 0); // waiting for process 3 to finish operation
186 waitpid(pid4, NULL, 0); // waiting for process 4 to finish operation
187

```

```

OS_1212508 > C main.c > main(int, char *[])
171 }
172
173 //Parent process
174 close(pipeFD[0][1]); //close write of process 1
175 close(pipeFD[1][1]); //close write of process 2
176 close(pipeFD[2][1]); //close write of process 3
177 close(pipeFD[3][1]); //close write of process 4
178
179 gettimeofday(&end, NULL);
180
181 waitpid(pid, NULL, 0); // waiting for process 1 to finish operation
182 waitpid(pid2, NULL, 0); // waiting for process 2 to finish operation
183 waitpid(pid3, NULL, 0); // waiting for process 3 to finish operation
184 waitpid(pid4, NULL, 0); // waiting for process 4 to finish operation
185
186 if(read(pipeFD[0][0], resultMatrix, sizeof(resultMatrix)) == -1) { return 5; } // reading from process 1
187
188 if(read(pipeFD[1][0], resultMatrix + N/4, sizeof(resultMatrix)/4) == -1) { return 5; } // reading from process 2
189
190 if(read(pipeFD[2][0], resultMatrix + 2*N/4, sizeof(resultMatrix)/4) == -1) { return 5; } // reading from process 3
191
192 if(read(pipeFD[3][0], resultMatrix + 3*N/4, sizeof(resultMatrix)/4) == -1) { return 5; } // reading from process 4
193
194 close(pipeFD[0][0]); //close read of process 1
195 close(pipeFD[1][0]); //close read of process 2
196 close(pipeFD[2][0]); //close read of process 3
197 close(pipeFD[3][0]); //close read of process 4
198 //printMatrix(resultMatrix);
199
200 tm = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
201 printf("\n\nTime is: %fs\n\n", tm);
202
203 return 0;
204
205

```

Time is: 0.004283s

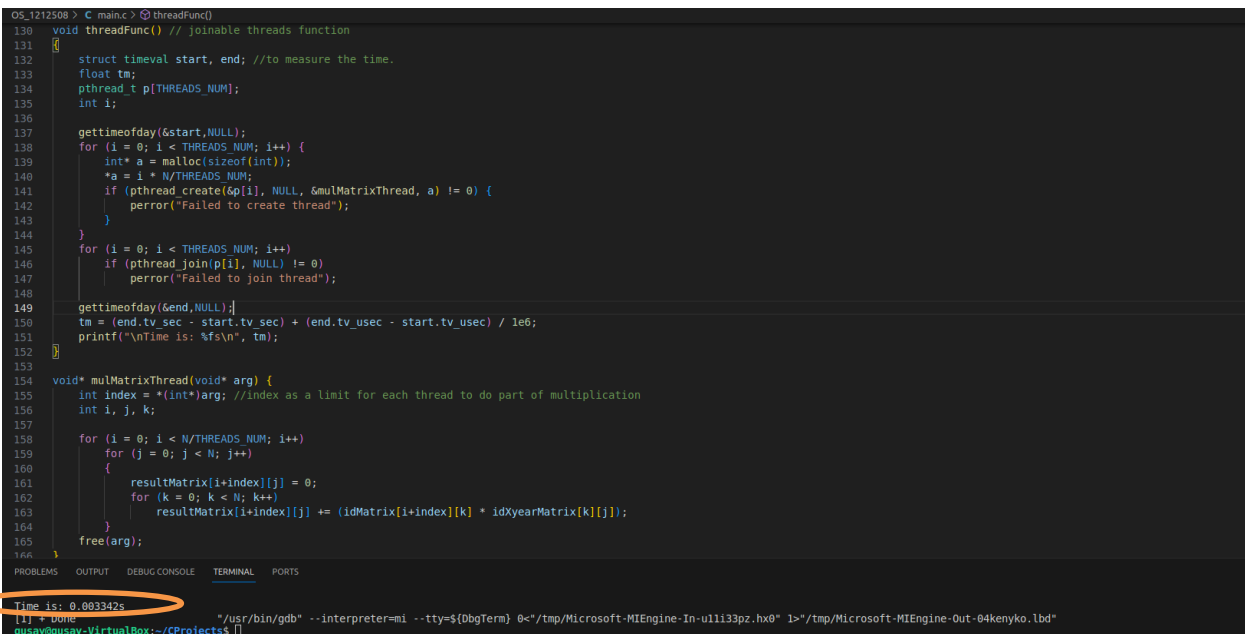
From above snapshots that we used 4 child processes we notice that the time increased to 0.004283s which is 4.283ms instead of decreasing and throughput equals $233.48s^{-1}$ which is less than throughput of 2 child processes.

As a result of trying different number of child processes and based of setting 5G Ram and 4 Cores for my linux VM so the best number of child processes is 2 child process, so incresing number of processes does not make a sense.

Joinable Threads:

There is below picture of using joinable threads in threadFunc() to do the task, also we implement a new function to do the task "multiplication" that special just for threads its called mulMatrixThread(), then we called thread function after we globalize our matrices.

The time shown in the picture below that for using 2 Threads which is 3.342ms and throughput is $299.22s^{-1}$



```
05.1212508 > C main.c > threadFunc()
130 void threadFunc() // joinable threads function
131 {
132     struct timeval start, end; //to measure the time.
133     float tm;
134     pthread_t p[THREADS_NUM];
135     int i;
136
137     gettimeofday(&start, NULL);
138     for (i = 0; i < THREADS_NUM; i++) {
139         int* a = malloc(sizeof(int));
140         *a = i * N/THREADS_NUM;
141         if (pthread_create(&p[i], NULL, &mulMatrixThread, a) != 0) {
142             perror("Failed to create thread");
143         }
144     }
145     for (i = 0; i < THREADS_NUM; i++)
146         if (pthread_join(p[i], NULL) != 0)
147             perror("Failed to join thread");
148
149     gettimeofday(&end, NULL);
150     tm = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
151     printf("\nTime is: %fs\n", tm);
152 }
153
154 void* mulMatrixThread(void* arg) {
155     int index = *(int*)arg; //index as a limit for each thread to do part of multiplication
156     int i, j, k;
157
158     for (i = 0; i < N/THREADS_NUM; i++)
159         for (j = 0; j < N; j++)
160         {
161             resultMatrix[i+index][j] = 0;
162             for (k = 0; k < N; k++)
163                 resultMatrix[i+index][j] += (idMatrix[i+index][k] * idYearMatrix[k][j]);
164         }
165     free(arg);
166 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Time is: 0.003342s
[1] + Done
qusay@qusay-VirtualBox:~/CProjects\$

And the time shown in the picture below that for using 4 Threads which is 5.372ms and throughput is $186.15s^{-1}$



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Time is: 0.005372s
[1] + Done
qusay@qusay-VirtualBox:~/CProjects\$

As a result of increasing threads the time was increased so that depend on our system and attributes of its components, so using 2 "joinable" threads is better for our case that need 3.342ms and make throughput of $299.22s^{-1}$.

* Note: I moved code of multiplying matrices that special for threads to threadFunc and creating threads, joining/detaching in the main function

Detached Threads:

In the pictures bellow we used 2 threads and replacing joining threads with detached and of actually we disable joining to notice the difference in the time.

```
33 pthread_t p[THREADS_NUM]; // array of threads to be
34 pthread_attr_t detached;
35 pthread_attr_init(&detached);
36 pthread_attr_setdetachstate(&detached, PTHREAD_CREATE_DETACHED);

49 ///////////////end of initializing//////////
50 gettimeofday(&start,NULL);
51 //childProcess();
52
53 int m;
54 for (m = 0; m < THREADS_NUM; m++) // creating threads loop
55 {
56     int* a = malloc(sizeof(int));
57     *a = m * N/THREADS_NUM;
58     if (pthread_create(&p[m], &detached, &threadFunc, a) != 0) {
59         perror("Failed to create thread");
60     }
61     pthread_detach(p[m]); //detaching threads
62 }
63
64 //for (m = 0; m < THREADS_NUM; m++) //joining threads loop
65     if (pthread_join(p[m], NULL) != 0)
66         perror("Failed to join thread");*/
67
68
69 pthread_attr_destroy(&detached);
70 gettimeofday(&end, NULL);
71 tm = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
72 printf("Time from main is: %fs\n", tm);
73 //printMatrix(resultMatrix);
74 pthread_exit(0);
75
76
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

Time from main is: 0.002307s
[1] + Done
qusay@qusay-VirtualBox:~/CProjects\$

As we see from above the time is 2.307ms and throughput is $433.46s^{-1}$

Now we gonna use 4 threads to show which is better

```
Time from main is: 0.003835s
[1] + Done
qusay@qusay-VirtualBox:~/CProjects$
```

We notice that using 4 threads takes 3.835ms and $260.76s^{-1}$ which are not better than 2 threads.

So in our case the best case is 2 Detached Threads.

* Note: we can't measure time in detached threads and the reason is main thread terminate its execution before detached and that make complex, but we meagure for all detached and printing the time on the screen as above.

Table of results:

Number/Type	Native approach	Child Processes	Joinable Threads	Detached Thrads
Time	5.664ms	2 Processes 2.361ms	2 Threads 3.342ms	2 Threads 2.307ms
Throughput	176.55s ⁻¹	423.55s ⁻¹	299.22s ⁻¹	433.46s ⁻¹
Time	-	4 Processes 4.283ms	4 Threads 5.372ms	4 Threads 3.835ms
Throughput	-	233.48s ⁻¹	186.15s ⁻¹	260.76s ⁻¹

There is in above table we showed the difference between processes and threads in time and throughput.

And we conclude that the best case in child processes is 2 and in both joinable or detached 2 thrads is the best in hand of the fast and throughput, and that is suitable for my linux VM as we mentioned above its properties.

* Note: all code captured in pictures above **have an update** to fits requiered and sent in .rar file that also include this report.