System Practicum

CS-307

(Semester - 6)

Assignment - 1
Report

Team Members:

Shubham Saurav (B19222)

Prashant Kumar (B19101)

Ravi Kumar (B19191)

Saloni Patidar (B19111)

Sagar Tarafdar (B19110)

Gaurav Sahitya (B19083)

Part - 1:

Explanation:

```
1  // Our Hacker Shell
2  // System Practicum - A1 - Part1
3  // Team Details:
4  // Prashant Kumar  -  B19101
5  // Ravi Kumar      -  B19191
6  // Saloni Patidar  -  B19111
7  // Gaurav Sahitya  -  B19083
8  // Sagar Taffardar -  B19110
9  // Shubham Saurav  -  B19222
10
11 // Including all necessay library
12 #include <iostream>
13 #include <map>
14 #include <fstream>
15 #include <vector>
16 #include <string>
17 #include <signal.h>
18 #include <dirent.h>
19 #include <sys/wait.h>
20 #include <sys/types.h>
21
22 // library separate for windows and linux
23 #ifdef __unix__ /* __unix__ is usually defined by compilers targeting Unix systems */
24
25 #define OS_Windows 0
26 #include <unistd.h>
27 #include <stdlib.h>
28 #include <stdio.h>
29
30 #elif defined(_WIN32) || defined(WIN32) /* _Win32 is usually defined by compilers targeting 32 or   64 bit Windows systems */
31
32 #define OS_Windows 1
33 #include <windows.h>
34 #include <stdio.h>
35 #include <tchar.h>
36
37 #endif
```

Line 1-9: This is a comment section containing the details of the assignment and our team members.

Line 11-20: We are including all the necessary libraries which are required to implement our own shell.

Line 23-28: Here, we are using the if condition and checking that if the code is being run on the Unix system, the libraries following the if statement are included.

Line 29-35: Here, we are using the else-if condition and checking that if the code is being run on the Windows system, the libraries following the else-if statement are included.

```
39  using namespace std;
40
41  // storing environment Variables in key value pair
42  map<string, string> environmentVariables;
43
44  // list of all commands and their description
45  vector<pair<string, string>> commands_Table = {
46      {"clr", "Clear the terminal screen"},
47      {"pause", "Pause operations of the shell until 'Enter' is pressed."},
48      {"help", "Display user manual"},
49      {"quit", "Quit the shell"},
50      {"history", "Display the list of previously executed commands, even on shell restart."},
51      {"pwd", "GIve the current working directory"},
52      {"cd", "change directory"},
53      {"dir", "list content of directory"},
54      {"echo", "display a line of text"},
55      {"environ", "List all the environment strings of the current shell and the bash shell"}};
56
57  // Class contain execution of all commands
58  class ALLCommands
59  {
60  public:
61      // 1. clr
62      // function for clear the terminal screen
63      void clear_screen()
64      {
65          printf("\e[H\e[2J\e[3J");
66      }
```

Line 42: We make a map "*environmentVariables*" in which the key value is string and the mapped value is also a string. This map will be used for storing the environment variables. (will be used later)

Line 45-55: We make a vector "*commands_Table*" which contains pairs of string and string and store all the commands which we are required to implement and also store the description of the commands.

Line 58: Here, we define a class "*AllCommands*" under which we will define the functions which are required to implement the given commands.

Line 63-66: We define a function "*clear_screen*" which is used to clear the command palette. We just make use of the regex "***\e[H\e[2J\e[3J***" using the print function. Here, " ***\e[H***" moves the cursor to the home position, "***\e[2J***" prints the spaces wherever there were already existing characters and "***\e[3J***" clears the scroll buffer.

```
68      // 2. pause
69      // function for Pausing operations of the shell until ‘Enter’ is pressed.
70      void pause_cmd()
71      {
72          cout << "Execution is paused! :)\nPress enter to continue!\n";
73          while (cin.get() != '\n');
74      }
75
76      // 3. help
77      // function for dispaying user manual and all commands info
78      void help()
79      {
80          for (auto x : commands_Table)
81              cout << x.first << " - " << x.second << "\n";
82      }
83
84      // 4. quit
85      // function that quit the shell
86      void quit_shell()
87      {
88          exit(1); // exit() which cause process termination. Include in <stdlib.h>
89      }
90
91      // 5. history
92      // Display the list of previously executed commands, even on shell restart
93      void print_cmd_history()
94      {
95          ifstream read_history("history.txt");
96          string line;
97          int line_no = 0;
98          if (read_history.is_open())
99          {
100             while (read_history)
101             {
102                 getline(read_history, line);
103                 cout << line_no << " " << line << endl;
104                 line_no++;
105             }
106             read_history.close();
107         }else perror("File open error");
108     }
```

<u>Line 68-74:</u> Here, we make a function "*pause_cmd*", which is used to pause all inputs until the "*Enter*" key is pressed. To implement this, we have made a while loop that does not end until "\n", i.e. a new line character is given as input which eventually is the "*Enter*" key.

<u>Line 78-82:</u> Here, we make a function "*help*" which is used to give all the commands and their description for the user. We make use of the vector "*command_Table*", which we had created earlier. We directly print the pairs which we had stored in the vector.

<u>Line 86-89:</u> Now, we make a function "*quit_shell*" which will help us in exiting our shell. Here, we make use of the "*exit()*" function which is included in the "*<stdlib.h>*" library.

__Line 93-108:__ We make a function "*print_cmd_history()*" which is used to print the history of the command which we have used in the past, including those commands which we were used in the last session that is before restarting the shell. To implement this function, we make a history.txt file which stores all the previously stored commands and simply prints them. If for any reason, we are unable to open the file then, we print an error message of unable to open file.

```
110     // 6. pwd
111     // give the present working directory
112     string getPWD()
113     {
114         char tmp[1000];
115         if (getcwd(tmp, sizeof(tmp)) == nullptr)
116             perror("get directory path error");
117         // give the absolute pathname of CWD. Include in <unistd.h>
118         return string(tmp);
119     }
120
121     // 7. cd <directory>
122     // Change the current default directory to <directory>. If the <directory> argument is
123     // not present, report the current directory. If the directory doesn’t exist an
124     // appropriate error should be reported.
125     void changeDirectory(vector<string> arg)
126     {
127         int arg_length = arg.size() - 1;
128         if (arg_length > 1)
129             cout << "shell: cd: too many arguments\n";
130         else if (arg_length == 1)
131         {
132             if (chdir(arg[1].c_str()) != 0){ // chdir include in <unistd.h>
133                 perror("Error");
134             }
135             else
136                 environmentVariables["PWD"] = this->getPWD();
137         }
138     }
139
140     // 8. dir <directory>
141     // list content of directory
142     void listDirContent(vector<string> args)
143     {
144         int arg_len = args.size() - 1;
145         if (arg_len == 0)
146             this->printDirContent(".");
147         else
148         {
149             for (int i = 1; i < arg_len + 1; i++)
150             {
151                 cout << args[i] << "\n";
152                 this->printDirContent(args[i]);
153                 cout << "\n";
154             }
155         }
156     }
```

Line 112-119: We make a function "*getPWD*" which returns string.The "*getcwd*" gets the current working directory of the program and stores it in an array of characters which is returned at last. If due to any reason, the function is not able to find the current working directory, then an error message is displayed.

Line 125-138: We make a function "*changeDirectory*" which takes the argument entered by the user, checks if the command entered is valid or not. It is valid then it changes the directory using the "*chdir()*" function. If the directory is not present then an appropriate error message is displayed. We also set the environment variable "PWD" to the current directory.

Line 142-156: We make a function "*listDirContent*" which takes the argument of the vector of string of the command line. If the user just enters "dir" which means total argument = 0. Hence we call the printDirContent for present working directory. Else if the user enters "n" arguments then print the n different arguments.

```
158     // function for listing content of particular path directory
159     void printDirContent(string path)
160     {
161         DIR *dir;
162         struct dirent *diread;
163 
164         if ((dir = opendir(path.c_str())) != nullptr)
165         {
166             while ((diread = readdir(dir)) != nullptr)
167             {
168                 cout << diread->d_name << "\n";
169                 // cout << diread->d_type<<"\n";
170             }
171             closedir(dir);
172         }
173         else
174         {
175             string error = "cannot access " + path + ": ";
176             perror(error.c_str());
177             // return EXIT_FAILURE;
178         }
179     }
180 
181     // 9. environ
182     // List all the environment strings of the current shell and the bash shell
183     void printEnvVariables()
184     {
185         for (auto x : environmentVariables)
186         {
187             cout << x.first << "=" << x.second << "\n";
188         }
189     }
190 
191     // 10. echo <comment>
192     // Display <comment> on the display followed by a new line. Multiple spaces/tabs
193     // should be reduced to a single space.
194     void echo(vector<string> args)
195     {
196         int i = 1, arg_len = args.size();
197         for (; i < arg_len; i++)
198             cout << args[i] << " ";
199         cout << "\n";
200     }
201 };
```

<u>**Line 159-179:**</u> "printDirContent" which takes the path as a string. This function is used to print the directory content of that path. For this purpose we have used DIR struct which takes the pointer to the directory stream using opendir() function. After this to read the directory content line by line we have used readdir(). The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp.

```
struct dirent {
        ino_t          d_ino;       /* Inode number */
        off_t          d_off;       /* Not an offset; see below */
        unsigned short d_reclen;    /* Length of this record */
        unsigned char  d_type;      /* Type of file; not supported by all filesystem types */
        char           d_name[256]; /* Null-terminated filename */
};
```

Line 183-189: We make a function “*printEnvVariables()*” which is used to print all the environment variables. We just make use of the map “*environmentVariables*” and print the keys and the values corresponding to each of the keys.

Line 194-201: We make a function “*echo()*” which takes the argument vector as a parameter and just prints whatever is written after echo. Now, as we know that the args is an array of the input arguments so we just print the arguments which are after the “echo” command.

```
203 // to check that the command exist or not
204 bool command_Exist(string commad)
205 {
206     for (auto x : commands_Table)
207         if (commad == x.first)
208             return true;
209     return false;
210 }
211 
212 // function for splitting commands
213 vector<string> splitString(string command)
214 {
215     vector<string> wordList;
216     string word = "";
217     for (auto x : command)
218     {
219         if (x == ' ')
220         {
221             if (word != "")
222                 wordList.push_back(word);
223             word = "";
224         }
225         else
226             word = word + x;
227     }
228     if (word != "")
229         wordList.push_back(word);
230     return wordList;
231 }
232 
233 ALLCommands func;
```

Line 204-210: We make a function "*command_Exist*" to check if the command entered by the user is valid or not. We pass the command entered as a parameter to the function and check the parameter with the first element of the pairs which we had stored in the vector "*commands_Table*". If the command is found then we return "*true*" and if not found then we return "*false*".

Line 213-231: We make a simple function "*splitString*" which is used to split the commands and other arguments which are entered through the command palette. It just splits the string from wherever it finds space.

```cpp
235 void run_command(vector<string> splitCommand)
236 {
237 
238     string cmd = splitCommand[0];
239     if (cmd == "clr")
240         func.clear_screen();
241     else if (cmd == "pause")
242         func.pause_cmd();
243     else if (cmd == "help")
244         func.help();
245     else if (cmd == "quit")
246         func.quit_shell();
247     else if (cmd == "history")
248         func.print_cmd_history();
249     else if (cmd == "environ")
250         func.printEnvVariables();
251     else if (cmd == "pwd")
252         cout << func.getPWD() << "\n";
253     else if (cmd == "cd")
254         func.changeDirectory(splitCommand);
255     else if (cmd == "dir")
256         func.listDirContent(splitCommand);
257     else if (cmd == "echo")
258         func.echo(splitCommand);
259 }
260 
261 
262 // func for storing all env in environmentVariables
263 void stroringAllEnvVariables(char **envp)
264 {
265     for (char **env = envp; *env != 0; env++)
266     {
267         string key = "", value = "";
268         char *thisEnv = *env;
269         string str = string(thisEnv);
270         int i = 0, n = str.length();
271         for (i = 0; i < n; i++)
272         {
273             if (str[i] == '=')
274                 break;
275         }
276         key = str.substr(0, i);
277         value = str.substr(i + 1, str.length() - i - 1);
278         environmentVariables[key] = value;
279     }
280 }
```

<u>Line 235-259:</u> We make a function "*run_command*" which is used to run all the commands as per the user gives the input. Here, the splitted arguments are given as input. The string at the "0th" index is always the command. Hence, we check the commands and call the function corresponding to that command.

Line 263-280: We make a function "*storingAllEnvVariables*" where we pass "envp" which is an array of pointers to all the environment variables. Using, this function we just store the environment variables as key and their path as values in the map "*environmentVariables*" which we created in the starting.

```
282 // setting SHELL env equal to our shell path
283 void setting_shell_Environ()
284 {
285     char buf[1000];
286     if (readlink("/proc/self/exe", buf, sizeof(buf)) < 0)
287         perror("readlink() error: not able to set shell env");
288     else
289         environmentVariables["SHELL"] = buf;
290 }
291
292
293 int main(int argc, char **argv, char **envp)
294 {
295     // storing all env
296     stroringAllEnvVariables(envp);
297     // set SHELL env
298     setting_shell_Environ();
299
300     // for executing batchfile
301     if (argc == 2)
302     {
303         if (freopen(argv[1], "r", stdin) == NULL)
304         {
305             perror("Error");
306             return 0;
307         }
308     }
309
310     // ignoring Ctrl+C intrupt signal
311     signal(SIGINT, SIG_IGN);
312     string command;
313     if (argc == 1)
314         cout << "\033[1;31mHacker@root:>>\033[0m ";
315
```

Line 283-290: **(Bonus Part)** We make a function "*setting_shell_Environ*". We make a buffer of char. Now, we call the readlink function which returns to us the path of the executable file which we store in the buffer. If due to any reason "*readlink*" is not able to return the path of the executable file then we return an error else we set the value of the key "*Shell*" in the map "*environmentVariables*" as the path returned by readlink.

<u>Line 293-314:</u> We make the "*main()*" function which takes the parameters as argument count, argument vector and the array of the pointer to the environment variables. We call the function "*storingAllEnvVariables()*" which stores all the environment variables in the map, then we call the function "*setting_shell_Environ()*" which sets the shell environment to the current path. Now, we check that if the argument count is 2, then we read the batch file which is entered, else we return an error.

Now, we set the signal "*SIGINT*" to "*SIG_IGN*" which means that we are ignoring the signal "*Ctrl+C*".

```
316     // looping shell
317     while (getline(cin, command))
318     {
319         if(command[0]=='#' || command=="") continue;
320 
321         ofstream historyFile;
322         // storing history of all command in history file
323         historyFile.open("history.txt", ios::app);
324         historyFile << command << "\n";
325         historyFile.close();
326 
327         vector<string> splitCommand = splitString(command);
328 
329         if (command_Exist(splitCommand[0]))
330             run_command(splitCommand);
331         else
332         {
333             int status;
334             pid_t pid = fork();
335             if (pid == -1)
336             {
337                 // pid == -1 means error occured
338                 printf("can't fork, error occured\n");
339                 exit(EXIT_FAILURE);
340             }
341             else if (pid == 0)
342             {
343                 /* Child Process */
344 
345                 /* Allow signal's default behaviour for process */
346                 signal(SIGINT, SIG_DFL);
347                 char *arg[] = {NULL};
348                 if (execv(splitCommand[0].c_str(), arg) == -1)
349                 {
350                     cout << splitCommand[0] << ": command not found\n";
351                 };
352                 exit(0);
353             }
354             else
355             {
356                 wait(&status);
357             }
358         }
359         if (argc == 1)
360             cout << "\033[1;31mHacker@root:>>\033[0m ";
361     }
362     return 0;
363 }
```

<u>Line 317-363:</u> We make a while loop to implement our shell. If there is no command entered then the shell will move to another line and wait for input.

Now, we store the entered command in the "*history.txt*" file so we can print it when required. We split the command and check if the command exists and then run the command. We create a child process so that we run any file then the shell does not exit on its own after the file executing is done. We also change the signal to the default "*Ctrl+C*" so that the user can exit the execution of the file in the middle if required but it will not exit the shell. After that we wait for the processes to finish.

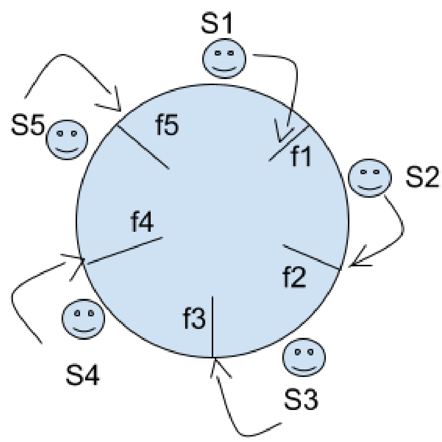
<u>Part - 2:</u> Dining Students

In this problem we are simulating the students' eat-think cycle using 3 approaches. We are using an array of mutex (initialized with 'mutex spoons[5]' in the code) for every spoon.

<u>Approach 1:</u>

In this approach *student 'A'* will first pick its right spoon then after picking it it will try to get the left spoon. Every student will try to do the same. Now since there are 5 spoons and 5 students, at a time at max only 2 students will be able to eat. Code for this approach is in the **main1.cpp** file. In this approach, we lock the mutex at index i if the students pick the spoon i, and similarly for index j if it picks jth spoons. Now if a student has finished eating, it will drop the right spoon first by releasing the lock, i.e. by unlocking the respective mutex and then it will drop the left spoon.

This approach may lead to deadlock, because there may be a situation where every student picks its right spoon, now all students have 1 spoon and no spoon is left on the table. But for eating 2 spoons are required hence everyone will wait for the next spoons and no one will drop the same.



The above Diagram shows the condition of deadlock.

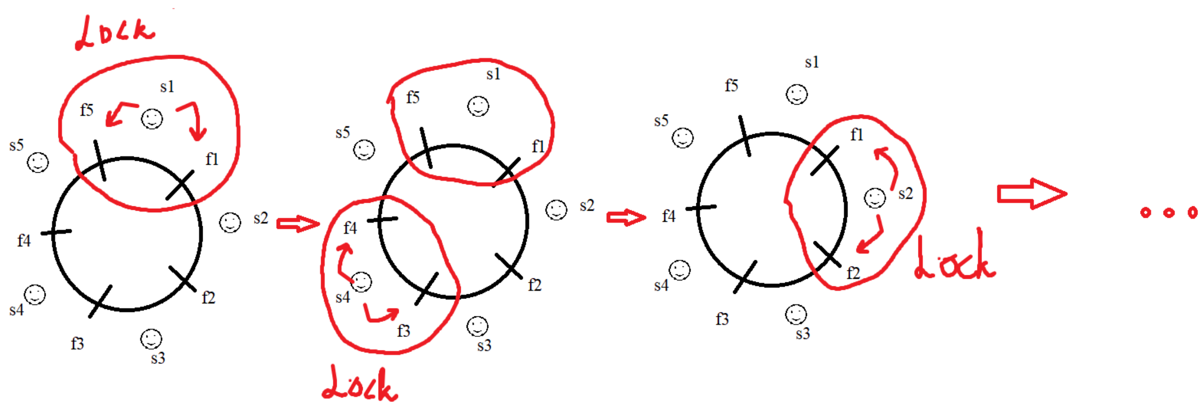
(Here f1 is spoon 1 and S1 is student 1).

This approach allows the fair allocation due to random think time which students take after eating. Starvation will also not occur in this case.

Student Number	Output 1 Eating	Output 2 Eating	Output 3 Eating	Output 4 Eating	Output 5 Eating	Average (Round off)
S1	37	37	37	37	37	37
S2	37	36	37	37	36	37
S3	36	37	36	36	37	36
S4	37	36	37	37	37	37
S4	36	37	36	36	36	36

Approach 2:

In this approach we are using another mutex ‘draw’. Codes for this part is in **main2.cpp** file Students will only be able to pick the spoon if both its left and right are available. To pick the spoon, one need to have the ‘draw’ mutex unlocked, and once the student get this mutex in unlocked position it will lock and take 2 spoon and, as soon as it gets both the spoon, it will release the ‘draw’ lock, so that others can also get this mutex to pick the spoon.

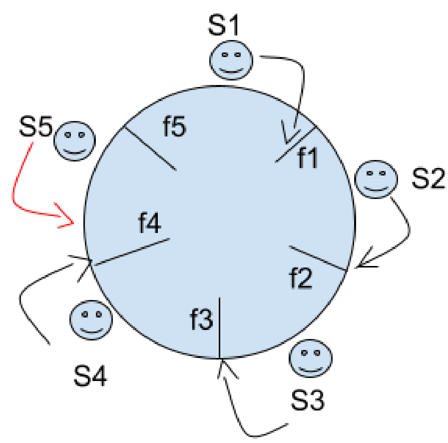


This approach prevents the deadlock condition as at every time, a student will have either 2 spoon or no spoon at all. Also this allows fair allocation to the student. This is also starvation free, because every pickup is random and it is almost impossible for a particular student to not get the spoons for a long time because of randomness and probability. In this approach since we are using extra lock, and only one who have this lock can take both spoon, total number of eating for each student reduces slightly.

Student Number	Output 1 Eating	Output 2 Eating	Output 3 Eating	Output 4 Eating	Output 5 Eating	Average
S1	31	31	31	31	31	31
S2	31	31	32	31	32	31
S3	31	31	31	31	31	31
S4	31	31	31	31	31	31
S5	31	31	30	31	31	31

Approach 3 :

Code for this approach is in the main3.cpp file. This approach is very similar to the first one, but the only change is that one student will pick the left spoon first if 4 other students are picking the right spoon first. This will prevent the deadlock because at any time at least one student will have the 2 spoons and hence deadlock won't be possible. In our code all students are picking the right spoon first except the 5th student who is trying to pick the left spoon.



Here Student 5 and student 4 will both try to get the spoon 4 but only one will be able to get that spoon, hence atleast one of them won't have any spoon. And if student 5 manages to get the spoon first then it will try to get spoon 5, meanwhile student 1 will also attempt to get the spoon 5. Hence initially atleast one of them will have 2 spoons. This does not allow fair allocation of spoons because student 5 and student 4 will have slight disadvantages over 1,2 and 3. This happens because student 5 and student 4 both are trying to get the spoon 5. It can be said that there is slight starvation for students 4 and 5.

Student Number	Output 1 Eating	Output 2 Eating	Output 3 Eating	Output 4 Eating	Output 5 Eating	Average
S1	38	38	39	39	39	39
S2	37	38	39	40	39	39
S3	38	38	38	39	39	38
S4	34	34	33	32	33	33
S5	34	34	32	33	32	33

Part - 3:

Code Explanation:

```
1   #include <iostream>
2   #include<bits/stdc++.h>
3   #include <thread>
4   #include <semaphore.h>
5   #include <mutex>
6   #include <condition_variable>
7   #include <vector>
8   #include <chrono>
9   #include <unistd.h>
10  #include <unordered_map>
11  using namespace std;
12
13  int n;
14  int** Matrix1;
15  int** Matrix2;
16  int** MatrixNormal;
17  int** MatrixThread;
18
19  void InitialiseMatrix(int **Matrix){
20      for (int i = 0; i < n; i++) {
21          Matrix[i] = new int[n];
22      }
23      for(int i=0; i< n; i++){
24          for(int j=0; j< n; j++){
25              int val = rand()%100;
26              Matrix[i][j] = val;
27          }
28      }
29  }
30  void PrintMatrix(int ** Matrix){
31      for(int i=0; i< n; i++){
32          for(int j=0; j< n; j++){
33              cout << Matrix[i][j] <<" ";
34          }
35          cout << endl;
36      }
37      cout << endl;
38  }
39
40  void thread_Multiplication(int row1, int row2 ){
41      for(int i = row1; i< row2; i++){
42          for(int j=0; j< n; j++){
43              for(int k = 0; k< n; k++){
44                  MatrixThread[i][j] += (Matrix1[i][k] * Matrix2[k][j]);
45              }
46          }
47      }
48  }
```

<u>Line 1-17:</u> First of all, we include all the important libraries which are required for the execution of the program. Now, we make 4 pointers namely, "*Matrix1*" and "*Matrix2*" which are used to store the matrices which are to be multiplied, "*MatrixNormal*" which is used to store the matrix which has been formed by the multiplication of the two matrices using the normal matrix multiplication method and "*MatrixThread*" which is used to store the matrix which has been formed by multiplication of the two matrices using threads.

<u>Line 19-29</u>: Here, we make an "*InitialiseMatrix*" function which is used to make columns for the matrix and also initialize the cells of the matrix with random values.

<u>Line 30-38:</u> Here, we make a "*PrintMatrix*" function which is used to print a matrix.

<u>Line 40-48:</u> Here, we make a "*thread_Multiplication*" function which is used to multiply a given number of rows to all the columns of another matrix, which can give that number of rows as the result of the matrix which we have to compute.

```
50  void MatMulNormal(){
51      for(int i=0; i< n; i++){
52          for(int j=0; j< n; j++){
53              for(int k = 0; k< n; k++){
54                  MatrixNormal[i][j]+= Matrix1[i][k] * Matrix2[k][j];
55              }
56          }
57      }
58  }
59
60  void MatMulThread(int n_thread){
61      int remainder  = n%n_thread;
62      int i=0; int j = n/n_thread;
63      thread threads[n_thread];
64      for(int k=0; k<n_thread; k++){
65          int extra = 0;
66          if(remainder){
67              extra = 1;
68              remainder--;
69          }
70          j = j + extra;
71          threads[k]=thread(thread_Multiplication,i,j);
72          i = j;
73          j += n/n_thread;
74      }
75      for(int k=0; k<n_thread; k++){
76          threads[k].join();
77      }
78  }
79
```

Line 50-58: Here, we make a "*MatMulNormal*" function which is used to multiply two matrices normally, using three for loops and form a resulting matrix which is "*MatrixNormal*".

Line 60-78: Here, we make a "*MatMulThread*" function. It finds the number of rows which can be computed by a single thread as per the number of threads specified. It first finds the base number of the rows to be computed by a single thread and if some rows are still left then it assigns 1 more row to a thread until no more rows are left. Then, it calls the "*thread_Multiplication*" function which multiplies the given number of rows which are assigned to each thread. At last, the threads are joined and the memory is cleaned.

```
81   int main(int argc, char *argv[]){
82       if(argc == 1){
83           cout <<"Please provide the size of the matrix"<<endl;
84           return 0;
85       }
86       if(argc > 2 ){
87           cout <<"Please provide only one positive integer"<<endl;
88           return 0;
89       }
90   
91       n = atoi(argv[1]);
92       //Initializing Matrix 1.
93       Matrix1 = new int*[n];
94       InitialiseMatrix(Matrix1);
95   
96       //Initializing Matrix 2.
97       Matrix2 = new int*[n];
98       InitialiseMatrix(Matrix2);
99   
100      //Initializing Matrix Normal, which is calculated using sequentially.
101      MatrixNormal = new int*[n];
102      for (int i = 0; i < n; i++) {
103          MatrixNormal[i] = new int[n];
104      }
105  
106      //Initializing Matrix Thread, which is calculated using thread.
107      MatrixThread = new int*[n];
108      for (int i = 0; i < n; i++) {
109              MatrixThread[i] = new int[n];
110      }
111      // Calling function to multiply matrix sequentially.
112      // time_t startSequential, endSequential;
113      // time_t startThreading, endThreading;
114      struct timespec startSequential, endSequential;
115      clock_gettime(CLOCK_MONOTONIC, &startSequential);
116      MatMulNormal();
117      clock_gettime(CLOCK_MONOTONIC, &endSequential);
118  
119      // Calling function to multiply using thread.
120      struct timespec startThreading, endThreading;
121      clock_gettime(CLOCK_MONOTONIC, &startThreading);
122      MatMulThread(4);
123      clock_gettime(CLOCK_MONOTONIC, &endThreading);
124  
125      double timeSequential = (endSequential.tv_sec  - startSequential.tv_sec)* 1e9;
126      timeSequential = (timeSequential + (endSequential.tv_nsec  - startSequential.tv_nsec))* 1e-9;
127      double timeThreading = (endThreading.tv_sec  - startThreading.tv_sec)* 1e9;
128      timeThreading = (timeThreading + (endThreading.tv_nsec - startThreading.tv_nsec))* 1e-9;
129      cout << "Time taken by sequential Multiplication : "
130          << fixed << timeSequential << setprecision(9)<<" seconds"<<endl;
131      cout << "Time taken by parallel Multiplication : "
132          << fixed << timeThreading << setprecision(9)<<" seconds"<<endl;
133  
134      // PrintMatrix(Matrix1);
135      // PrintMatrix(Matrix2);
136      // PrintMatrix(MatrixNormal);
137      // PrintMatrix(MatrixThread);
138  
139      return 0;
140  }
```

Line 81-140: Here, we make "*main()*" function which takes argument count (tells the number of arguments entered in the command line and argument input (Pointer which indicates to the array of the arguments entered). Then, we apply some check conditions on the size of the input size of the matrices. Now, we initialize the matrices "*Matrix1*" and "*Matrix2*" by calling the

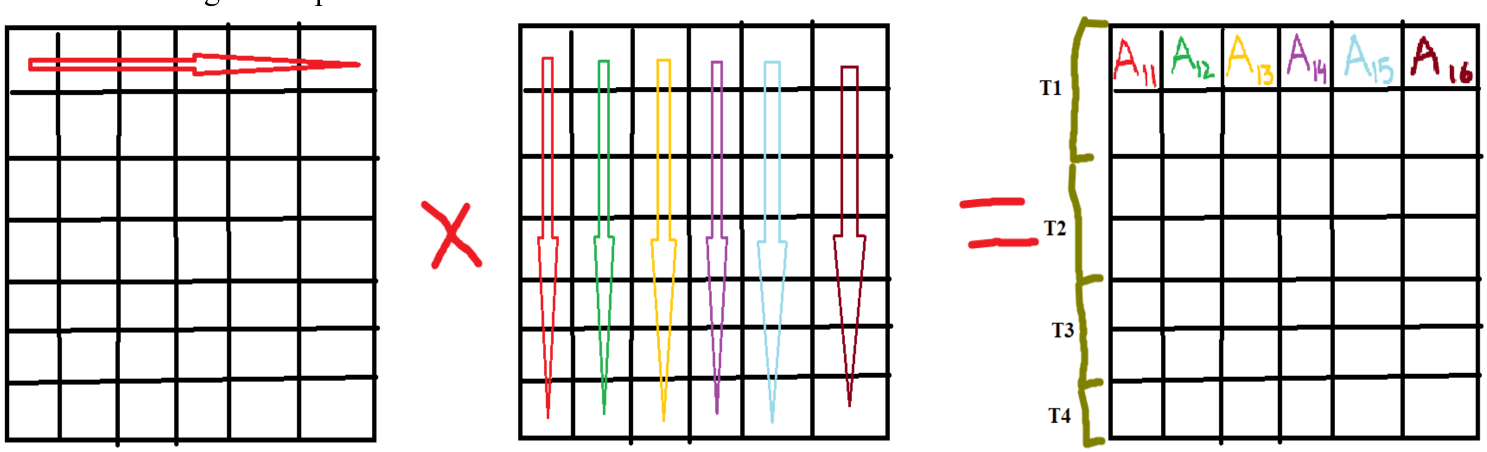
"*InitialiseMatrix*" function which we had made above. We also initialize the other two matrices in which are going to store the resulting matrices which are calculated using normal multiplication, i.e. "*MatrixNormal*" and the other matrix which is calculated using threads, i.e. "*MatrixThread*".

Now, we call the functions "*MatMulNormal*" and "*MatMulThread*" and measure the time which is required to complete the process of matrix multiplication in each case and the time is printed as an output.

Approach

For finding the resulting matrix, we just multiplied the two matrices filled with random numbers using 3 for loops and simply multiplied "i^{th}" row with "j^{th}" column to find the resultant element at "A_{ij}" position.

Now, for doing the matrix multiplication using threads. We first define the number of threads to be used in the program and assign each thread with a certain number of rows to be multiplied of Matrix-1 to be multiplied with all the columns of Matrix-2. Hence, due to parallelism the amount of time required to do the computation reduces if the number of cores are more than 1. Now, the computation time depends upon the number of threads and number of cores and the number of rows assigned to each thread. Though the computation can depend on the other processes running on the processor.



Here, we assume 4 threads and 7 rows where T1, T2, T3 and T4 are the threads. Hence, 3 of the threads get 2 rows to compute while the 4th thread get only one row to compute.

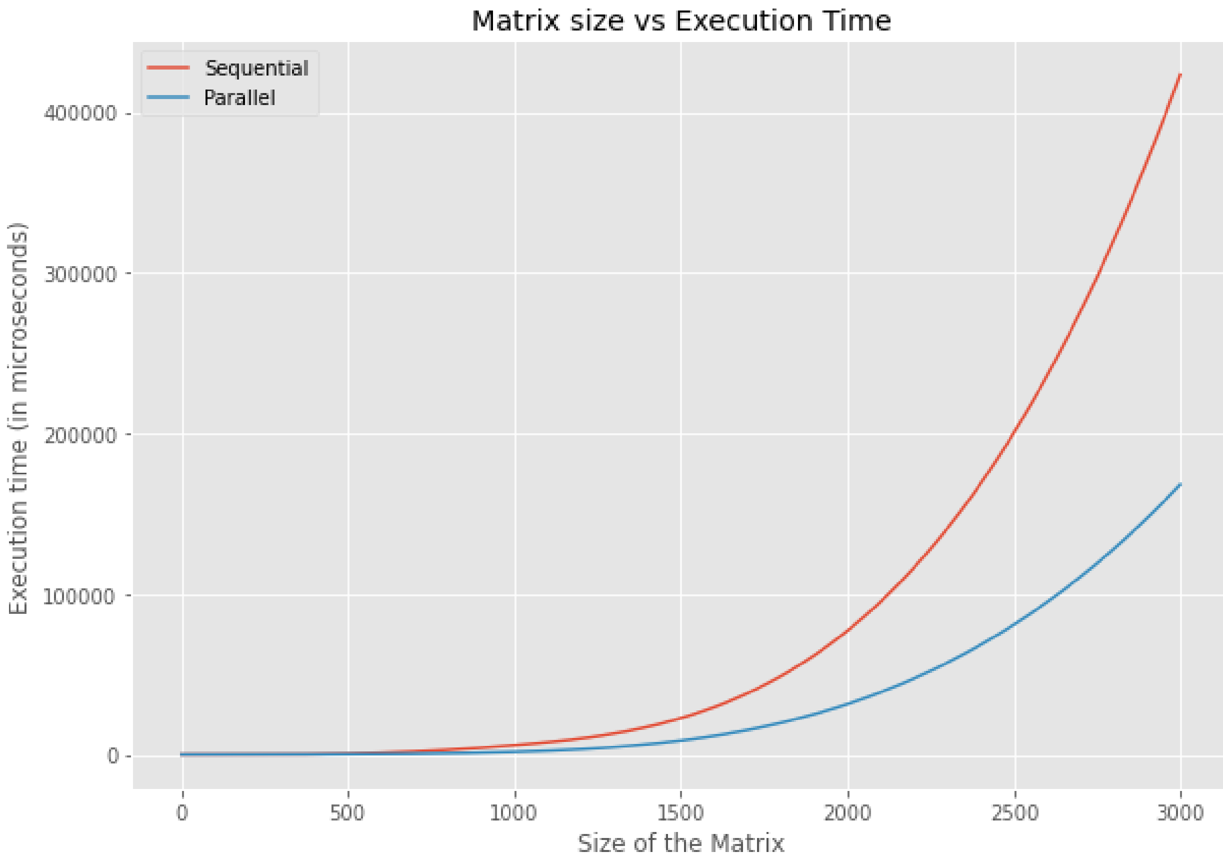
Output:

Size	Running Time (in ms)				
1	0	0	0	0	0
	0.3274	04.041842	0.851662	0.854227	07.657324
10	0.013	0.016	0.013	0.021	0.015
	0.52611	06.933684	02.277089	0.848617	05.781834
50	1.745	2.018	2.033	2.033	2.128
	1.972463	4.60946	1.735847	1.572385	1.456387
100	9.987	020.284	020.555	019.600	018.982
	2.725	4.609466	7.889853	6.164810	6.215809
500	499.25	520.6723	670.3401	453.234	512.139
	201.67	210.8734	280.342	203.2434	235.5443
1000	5826.02	6234.45	7234.324	5953.345	6765.902
	1646.03	1590.43	1902.024	1406.245	1690.324
2000	76977.07	75830.34	79345.23	81083.34	76802.30
	31383.90	31209.51	33490.59	32463.45	313405.04
3000	423195.5	447234.8	462069.2	452345.5	433245.8
	168089.7	174306.9	150345.3	140454.3	143453.8

Here Blue coloured numbers are showing the time taken by the sequential multiplication and brown coloured numbers are showing the time taken by the parallel multiplication.

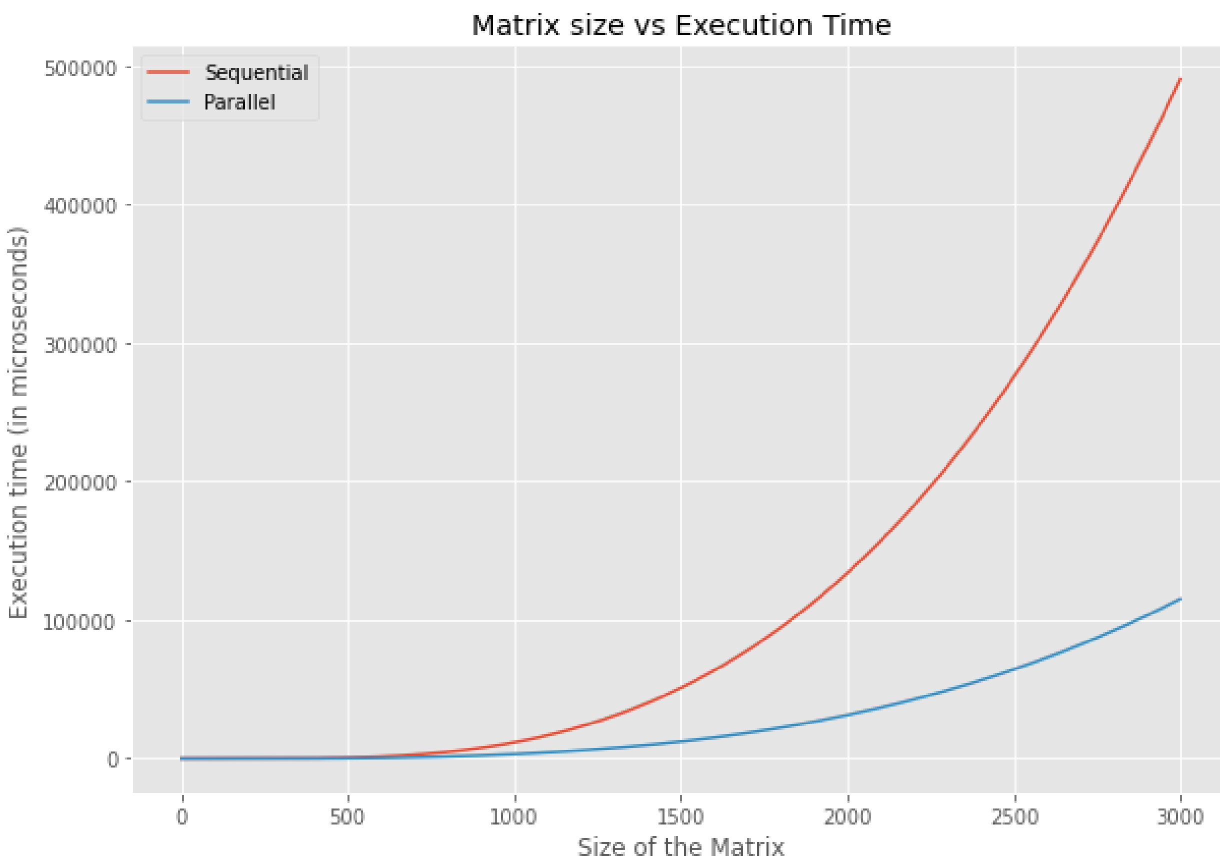
Graphs:

We run the program with a 4 core processor. And we used 4 threads for the calculaiton. Initially when the size of the input is small i.e., n is small then parallel multiplication using thread takes more time than that of the sequnetial multiplication. This is becuase at small input running time is low for the matrix calculation and the thread creation time is higher than the running time. Also at the higher output the parallel multiplication using thread one-fourth times the sequential multiplication.



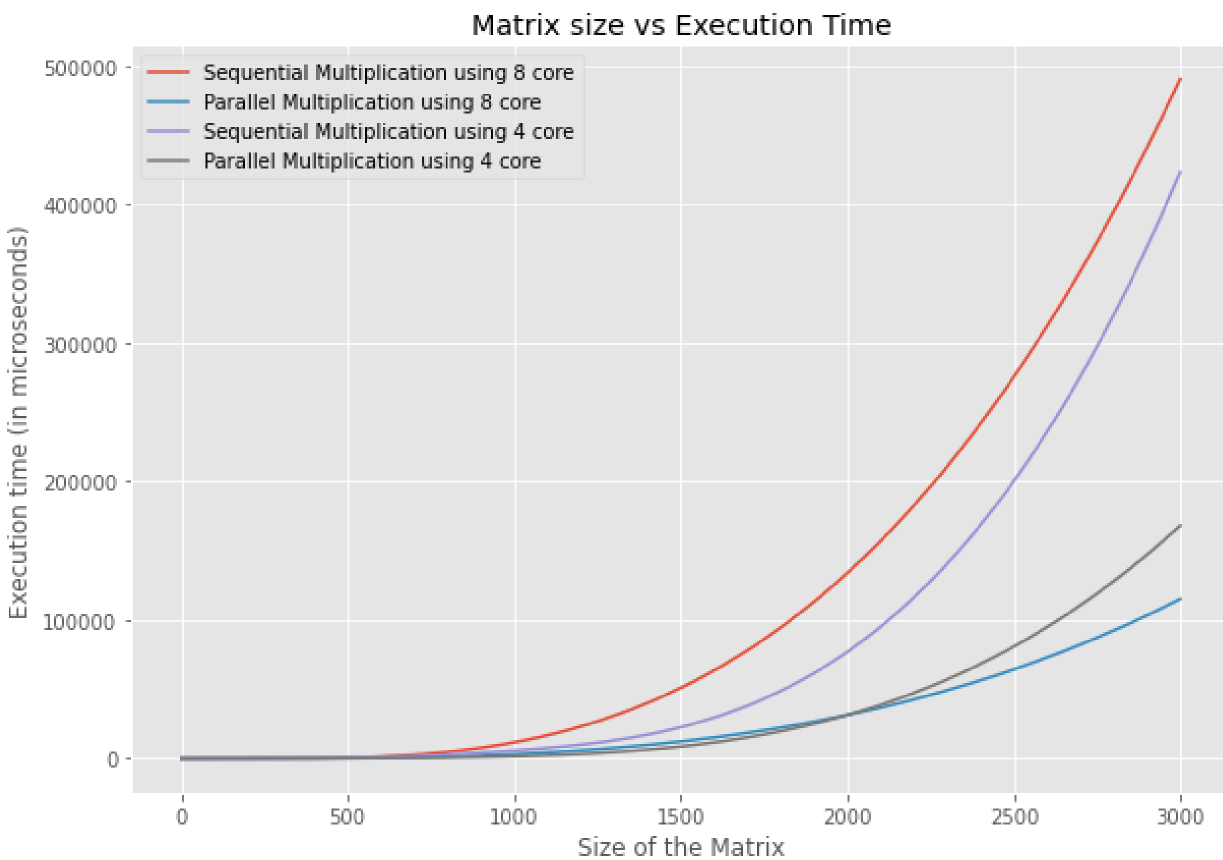
Graphs 1: Matrix multiplication using 4 core processors and 4 threads.

Then the program was run with the 8 core processors and 8 threads were used and for the low value of n, we get the same inference i.e., parallel multiplication takes more time than sequential multiplication. Also for the higher output the sequential multiplication takes more than 4 times than the parallel multiplication. Though in this, parallel multiplication is not about 1/8th times than the sequential multiplication. The possible reason for this is that many background processes runs and have higher priority than this, so, all the core was not used by the program.



Graphs 2: Matrix multiplication using 8 core processors and 8 threads.

Below graph shows the comparison using 4 threads and 8 threads. From the graph we can infer that calculation using 8 core processors takes less time than 4 core processors for the parallel computation.


Matrix size vs Execution Time
Sequential Multiplication using 8 core
Parallel Multiplication using 8 core
Sequential Multiplication using 4 core
Parallel Multiplication using 4 core
Execution time (in microseconds)
Size of the Matrix
0
100000
200000
300000
400000
500000
0
500
1000
1500
2000
2500
3000