Problem 1

First, I validate the command line arguments. If the validation is wrong, the usage of the command is printed or the cause of invalid arguments is explained. After that., the stopwords list is filled with the stopwords supplied by the file. This list will be later used to exclude words of the files being read to be included towards the top n words. Since I need to hold the frequency of words for each local file, the map construct is perfect, since it will use as unique keys the name of the word and as a value their frequency. For each file a thread is initialized which will read the file words, include their frequencies in the local maps, sort these maps and then insert the top words in the global map. To synchronize all threads a waiting mechanism is used, so when all of them are finished.

Both the Java and Posix code has the logic in mind, albeit with a different implementation. Except from the fact that a thread in Java needs to be created and then started, while in Posix they are created and started at the same time, there are some other differences. In Posix, a struct is used to include the filename and the pointer to the file for reading the content. In Java a LinkedHashMap is returned by the sort function, while in C++ this was replaced with a vector that included as a type a pair of string and int, since the map of C++ does not preserve insertion order. The result of this is that in the end the pairs are printed instead of map entries.

Problem 2

For this problem, I choose to use OpenMP for two reasons. First, I eliminate the Java boilerplate, since I am not utilizing any data structure. In addition, by using the parallel for directive of the OpenMP, I do not need to manually schedule what portion of the simulation loop is going to be executed by a thread. In this program, I simulate the MontyHall problem by comparing percentage of wins with switching and not switching the door. For each case, equal number of simulations are done. And the result is as expected around 66% for switching and 33% for not switching. In addition to the parallel for directive, the reduction clause is used for the wins variable.

Problem 3

For this problem, three matrices are initialized. Two will included random numbers, while the third is the product matrix which stores the result of the multiplication. Each thread will compute the product for a portion of the matrices' rows. These limits are computed in the constructor of the Task and are based on the rank of the thread, assuring that each thread will deal with a different portion. The limitation of this is that the matrix size must be a multiple of the number of threads, something which I assure during argument validation in the start of the program. The Java implementation is similar to the Posix one, with a small difference in the way the random numbers are generated. In Posix, I have utilized the std::uniform_real_distribtion class which produced random floating point values uniformly distributed based on the uniform probability distribution function. $P(x|a,b) = \frac{1}{b-a}$

Problem 4

For this problem I have chosen to implement a client/server solution for a chat room, where a server handles simultaneously many clients, as well as receives all text from clients and send it to all of them. I have created 4 classes. ServerThread extends Thread and is used to handle connection with each client thread, which on its own is being represented by the ClientThread class and is set up by the Client class. Server is the class which initialized a Server Socket based on the server port supplied by the command line. The client/server connection is made through Java Sockets/ServerSocket utilizing java.net package that provides classes for working with network applications. In this case, I am connecting through TCP/IP protocol. To connect to the server, a client needs to provide the ip of the host server (or localhost), the port number and the name to be used in the chat. The name and the localport from which the client is connecting is stored in a map, with the port number being the unique-valued key. So, each time a client is connected to the server, his info is stored on the hashmap. The purpose of the map is to identify the owner of the message being sent to each client.

```
PS C:\Users\jonim\[redacted]\homework\java> java Server 1444
Binding to port 1444, please wait  ...
Server started: ServerSocket[addr=0.0.0.0/0.0.0.0,localport=1444]
Type "exit" to close server
Waiting for a client ...
Client accepted! (ClientAddress: /127.0.0.1 ClientPort: 49560
Waiting for a client ...
Server Thread 49560 running.
Client accepted! (ClientAddress: /127.0.0.1 ClientPort: 49569
Waiting for a client ...
Server Thread 49569 running.
Removing client thread 49569 at 1
Removing client thread 49560 at 0
Client accepted! (ClientAddress: /127.0.0.1 ClientPort: 49571
Waiting for a client ...
Server Thread 49571 running.
Client accepted! (ClientAddress: /127.0.0.1 ClientPort: 49572
Waiting for a client ...
Server Thread 49572 running.
```

```
$ java Client localhost 1444 Jon
Establishing connection for Jon. Please wait ...
Connected to server! (ServerAddress: localhost/127.0.0.1 ServerPort: 1444)
Type "_exit" to disconnect from server
Hi!
Jon: Hi!
James: Ciao!
What color is your Bugatti?
Jon: What color is your Bugatti?
```

```
$ java Client localhost 1444 James
Establishing connection for James. Please wait ...
Connected to server! (ServerAddress: localhost/127.0.0.1 ServerPort: 1444)
Type "_exit" to disconnect from server
Jon: Hi!
Ciao!
James: Ciao!
Jon: What color is your Bugatti?
```