**Total Points 100**

**Due Wednesday 4/28/21 at 11:59 pm**

**TRUE/FALSE QUESTIONS [1 pt each]**

1. UDP protocol deals with retransmitting packets in case they are lost in route F
2. TCP protocol is more heavy-weight because it maintains state information about the connection T
3. Routing protocols are dynamic: they reconfigure under network topology change or outage automatically T
4. Domain Naming System (DNS) can be used for load balancing and faster content delivery T
5. The ping command is used to test whether you can make TCP connections with a remote host T
6. The accept() function needs to be called the same number of times as the number of client-side connect() calls to accept all of them T
7. HTTP protocol uses TCP underneath T
8. Ports [0,1023] are reserved for well-known services (e.g., HTTP, SMTP, DNS, TELNET) T
9. The master socket in a TCP server is used only to accept connections, not to run conversations with clients F
10. A socket is a pair of IP-address and port number combination in both client and server side T
11. For making a socket on the client side, the port number is chosen by the OS randomly from the available pool T
12. A socket is like other file descriptors and is added to the Descriptor Table T
13. UDP is connectionless while TCP is connection oriented T
14. The ping command is used to test network reachability T
15. The bind() function is used to set up a backlog of outstanding connections F

**SHORT ANSWERS**

**16. [25 pts]** Examine the Tannenbaum’s solution to the Dining-Philosophers problem posted in the following: <https://legacy.cs.indiana.edu/classes/p415-sjoh/hw/project/dining-philosophers/index.htm>.

1. [25 pts] Rewrite the given solution in C++ using the semaphore class from our class lecture, make sure the program compiles and runs. Test the program and verify that it works correctly and show that using screenshots.
2. #include <iostream>
3. #include <thread>
4. #include <stdlib.h>
5. #include <vector>
6. #include <unistd.h>
7. #include "Semaphore.h"
8. using namespace std;
9. Semaphore mtx (1); // will use as mutex
10. Semaphore s1 (0);
11. Semaphore s2 (0);
12. Semaphore s3 (0);
13. Semaphore s4 (0);
14. Semaphore s5 (0);
15. int status[5] = {0}; // 0 for thinking, 1 for hungry, 2 for eating
16. // let eating takes 3 seconds to complete
17. void tryToEat(int i){ /\* Let phil[i] eat, if waiting \*/
18. if ( status[i] == 1 && status[i-1] != 2 && status[i+1] != 2){
19. status[i] = 2;
20. if(i == 1){
21. s1.V();
22. }else if(i == 2){
23. s2.V();
24. }else if(i == 3){
25. s3.V();
26. }else if(i == 4){
27. s4.V();
28. }else if(i == 5){
29. s5.V();
30. }
31. cout << "number " << i << " is allowed to eat." << endl;
32. }
33. }
34. void take\_chopsticks(int i){
35. mtx.P();/\* critical section \*/
36. status[i] = 1;
37. cout << "number " << i << " is hungry." << endl;
38. tryToEat(i);
39. mtx.V(); /\* end critical section \*/
40. /\* Eat if enabled \*/
41. if(i == 1){
42. s1.P();
43. }else if(i == 2){
44. s2.P();
45. }else if(i == 3){
46. s3.P();
47. }else if(i == 4){
48. s4.P();
49. }else if(i == 5){
50. s5.P();
51. }
52. cout << "number " << i << " is eating." << endl;
53. sleep(3);
54. }
55. void drop\_chopsticks(int i){
56. mtx.P(); /\* critical section \*/
57. status[i] = 0;
58. cout << "number " << i << " back to thinking after eating." << endl;
59. tryToEat(i-1); /\* Let phil. on left eat if possible \*/
60. tryToEat(i+1); /\* Let phil. on rght eat if possible \*/
61. mtx.V(); /\* up critical section \*/
62. sleep(3);
63. }
64. // each producer gets an id, which is pno
65. void philo\_function (int i){
66. // int count = 0; // each producer threads has its own count
67. while (true){
68. // THINKING;
69. take\_chopsticks(i);
70. // EATING;
71. drop\_chopsticks(i);
72. }
73. }
74. int main (){
75. vector<thread> philos;
76. for (int i=0; i< 5; i++){
77. philos.push\_back (thread (philo\_function, i + 1));
78. }
79. for (int i=0; i<philos.size (); i++){
80. philos [i].join();
81. }
82. }

Output:

Text

Description automatically generated

[15 pts] Then, describe how this solution works and why it does not run into deadlocks.

As you can see from the above output, each of the philosopher has a chance to eat. Additionally, a philosopher will be “allowed to eat” soon after the philosopher beside him/her has “back to thinking after eating”.

The reason why my solution does not run into deadlocks is that a mutex lock is used whenever: 1. A philosopher’s status is updating 2. Checking if a philosopher is allowed to eat. The update of status involves thinking to hungry, hungry to eating, and eating to thinking. Additionally, each philosopher has their own semaphore to record if they are able to eat, thus each of them will only eat if their semaphore was incremented when they were allowed to eat.

Furthermore, I added a 3 seconds delay for eating so that the behavior is more realistic.

**17. [30 pts]:** You are given the responsibility of managing your home network that has the private IP address range of 172.16.0.0/12.

1. How many distinct internet-connected devices can your home support simultaneously?

The private IP address range of 172.16.0.0/12 indicates that ranging from 172.16.0.0 to 172.31.255.255 IP addresses will be allowed. In other words, 16 of class B address spaces will be allowed.

1. Is 173.16.0.1 a valid IP address in this range? How about 172.17.0.5? Explain your answer.

As mentioned above, only the addresses ranging from 172.16.0.0 to 172.31.255.255 are available. Thus 173.16.0.1 is not valid while 172.17.0.5 is valid.

1. You have disabled DHCP in your network. Now, can a friend, who has never been to your home before, expect to automatically connect to your wireless network given that he knows the network id and password key?

Having the DHCP disabled in your network indicates that new devices would require a static IP address, which means you will need to remember which device corresponds to which IP address, and distinguish it among all other connected devices. Therefore, as long as you obtain an IP address of the network and the password key, although it is less comfortable, we can expect the device to be connected.

**18. [30 pts]:** Write a program that looks the IP addresses of a given host name (e.g., [www.google.com](http://www.google.com)) and filters those addresses based on a string matching. For example, the following were addresses returned by when I looked up [www.google.com](http://www.google.com). Obviously, these addresses may change when you look them up yourself.

|  |
| --- |
| **prompt:~$** dig +short www.google.com  142.250.138.103  142.250.138.106  142.250.138.105  142.250.138.147  142.250.138.99  142.250.138.104 |

Now, your program should take 2 arguments: the host name and the search string, like the following:

|  |
| --- |
| **prompt:~$**./a.out www.google.com 250  142.250.138.99  142.250.138.147  142.250.138.106  142.250.138.103  142.250.138.105  142.250.138.104 |

Notice that it is keeping all the entries because they all have the string “250” in them. If you rather use the search string “10”, it should give you the following output because only these IP addresses have the string “10” in them.

|  |
| --- |
| **prompt:~$**./a.out www.google.com 10  142.250.138.106  142.250.138.103  142.250.138.105  142.250.138.104 |

For comparison, you can use the following query to find out the expected output:

**prompt:~$** dig +short <hostname> | grep <search string>

**Restrictions: You cannot use “grep” or execvp() to take help from any of the Linux commands.**

Write this program, name if dnsfilter.cpp and include it in your submission