CS-350 - Fundamentals of Computing Systems
Homework Assignment #8

Due on November 21, 2020 at 11:59 pm

Prof. Renato Mancuso

Renato Mancuso

Problem 1

The life of you and your roomates revolves around dinner. Hence the following code — Listing 1 — that uses semaphores has been designed to implement the logic of a generic roomie in your apartment, including yourself. In the code, each roomie always arranges a 2-course dinner. For each course, the roomie rolls a three-faced dice every night to decide what to eat for that course. Hence, for each dinner a roomie will roll the dice twice. The food in second course is always something different from the first.
Regardless of what each roomie wants to eat, they will need one plate and one glass for water for the whole dinner. In each course, they will also need a fork if eating pasta; a pair of chopsticks to eat sushi; and a spoon if eating soup. Because of the restricted cabinet space in the apartment, there are in total 4 plates, 3 glasses, 2 pairs of chopsticks, 2 forks, and 2 spoons.

Listing 1: A roomie in semaphore language.

```
1  /* Global SHARED variables */
2  semaphore plates := 4;
3  semaphore glasses := 3;
4  semaphore chop_pairs := 2;
5  semaphore forks := 2;
6  semaphore spoons := 2;
7
8  Process Roomie:
9
10    /* Local variables */
11    enum choice_1st_course;
12    enum choice_2nd_course;
13
14    Repeat:
15
16       /* Have a long day at school */
17
18       /* Dinner time! */
19
20       wait(glasses)
21       wait(plates)
22
23       choice_1st_course = roll_dice();
24       switch (choice_1st_course) {
25       case PASTA:
26           wait(forks);
27           break;
28       case SUSHI:
29           wait(chop_pair);
30           break;
31       case SOUP:
32           wait(spoon);
33           break;
34       deaful:
35           print("Oh boy I rolled the wrong thing.\n");
36           exit();
37       }
38
39       do {
40          choice_2nd_course = roll_dice();
41       while (choice_2nd_course == choice_1st_course);
42
43       switch (choice_2nd_course) {
44       case PASTA:
45           wait(forks);
46           break;
47       case SUSHI:
48           wait(chop_pair);
49           break;
50       case SOUP:
51           wait(spoon);
52           break;
53       deaful:
54           print("Oh boy I rolled the wrong thing.\n");
55           exit();
56       }
57
```

Problem 1 continued on next page...

```
58        eat_2_course_dinner(); /* YAY! */
59
60        if (choice_1st_course == PASTA || choice_2nd course == PASTA) {
61           /* Wash used fork */
62           signal(forks);
63        }
64
65        if (choice_1st_course == SUSHI || choice_2nd course == SUSHI) {
66           /* Wash used chopsticks */
67           signal(chop_pairs);
68        }
69
70        if (choice_1st_course == SOUP || choice_2nd course == SOUP) {
71           /* Wash used spoon */
72           signal(spoons);
73        }
74
75        /* Wash used glass and plate */
76
77        signal(plates)
78        signal(glasses)
79
80        /* Watch some Neshflix */
81        /* Sleep for 7ish hours */
82
83     Forever
```

a) Assume that there are 2 roomies in the house. Is it possible that the roomies end up (literally) starving? Motivate your answer. If you answer yes, provide a scenario leading to the deadlock.

Ans-1a With 2 roomies, no deadlock situation arises as there is atleast 2 of each utensils and the food from the second course is always different from the first.

b) Assume that there are 4 roomies in the house. Is it possible that some of the roomies end up (literally) starving? Motivate your answer. If you answer yes, provide a scenario leading to the deadlock.

Ans-1b No deadlock. With 3 glasses, 3 roomies can proceed beyond waiting for glasses and roll the die for the 1st course and atleast one of them will be able to acquire utensils for both courses to proceed eating in the worst case. Therefore atleast 1 roomie will be able to finish eating and release the resources for the others to progress.

c) Good news! One of your roomies kindly donated 3 more glasses to the house. From now on (including the parts below) assume that there are a total of 6 glasses. Does this change your answer for Part (a) and Part (b) above?

Ans-1c The answer for part a remains the same. For part b however, since 4 roomies could concurrently proceed to rolling the die after obtaining a glass and a plate each, a deadlock situation can occur after the first die roll as follows where each would be waiting for a utensil that the other roomie is holding:

Roomie	1	2	3	4
Utensil picked	f,-	f,-	s-	s-

Table 1: Ans-1c: Potential deadlock situation with 4 roomies

Problem 1 continued on next page...

With the situation predicted in Table 1, the first 2 roomies are waiting for the spoons and 3rd and 4th roomie are waiting for forks. They all starve!

d) Once again assume 4 roomies. What is the maximum number of plates that will be dirty at the same time? Motivate your answer.

Ans-1d A plate is considered dirty if it needs washing which implies that the roomie proceeded to eating their 2 course dinner after acquiring all the resources they need. Since there are 2 of each utensils, we consider the best case scenario to derive the maximum number of dirty plates at the same time. One such possibility is indicated in Table 2. With 2 of each utensils maximum number of dirty plates = 3.

Roomie	1	2	3
Utensil-1	f	c	s
Utensil-2	c	s	f

Table 2: Ans-1d: Maximum number of roomies that can proceed to eating with all resources acquired

e) Propose a fix for the code above that is free from deadlocks, regardless of the number of roomies. Here, the fix should be done by modifying the code, not by applying a deadlock avoidance scheme.

Ans-1e This is an open ended question so any fix that works would get full credit.
Since both courses are mutually exclusive therefore the roomies dont have to eat both courses together. Split the courses as follows (2):

Listing 2: A roomie who never starves in semaphore language.

```
1         /* Global SHARED variables */
2         semaphore plates := 4;
3         semaphore glasses := 6;  /*updated*/
4         semaphore chop_pairs := 2;
5         semaphore forks := 2;
6         semaphore spoons := 2;
7
8         Process Roomie:
9
10        /* Local variables */
11        enum choice_1st_course;
12        enum choice_2nd_course;
13
14        Repeat:
15
16        /* Have a long day at school */
17
18        /* Dinner time! */
19
20        wait(glasses)
21        wait(plates)
22
23        choice_1st_course = roll_dice();
24        switch (choice_1st_course) {
25        case PASTA:
26        wait(forks);
27        break;
28        case SUSHI:
29        wait(chop_pair);
30        break;
31        case SOUP:
32        wait(spoon);
33        break;
34        deafult:
35        print("Oh boy I rolled the wrong thing.\n");
```

```
36          exit();
37          }
38
39          eat_Ist_course();  /*YAY*/
40
41          if (choice_1st_course == PASTA) {
42                  /* Wash used fork */
43                  signal(forks);
44          }
45
46          if (choice_1st_course == SUSHI) {
47                  /* Wash used chopsticks */
48                  signal(chop_pairs);
49          }
50
51          if (choice_1st_course == SOUP) {
52                  /* Wash used spoon */
53                  signal(spoons);
54          }
55
56          do {
57          choice_2nd_course = roll_dice();
58          while (choice_2nd_course == choice_1st_course);
59
60          switch (choice_2nd_course) {
61          case PASTA:
62          wait(forks);
63          break;
64          case SUSHI:
65          wait(chop_pair);
66          break;
67          case SOUP:
68          wait(spoon);
69          break;
70          deafult:
71          print("Oh boy I rolled the wrong thing.\n");
72          exit();
73          }
74
75          eat_2nd_course(); /* YAY! */
76
77          if (choice_2nd course == PASTA) {
78                  /* Wash used fork */
79                  signal(forks);
80          }
81
82          if (choice_2nd course == SUSHI) {
83                  /* Wash used chopsticks */
84                  signal(chop_pairs);
85          }
86
87          if (choice_2nd course == SOUP) {
88                  /* Wash used spoon */
89                  signal(spoons);
90          }
91
92          /* Wash used glass and plate once done with both courses*/
93
94          signal(plates)
95          signal(glasses)
96
97          /* Watch some Neshflix */
98          /* Sleep for 7ish hours */
99
100         Forever
```

f) Consider the number of roomies to be 4. For the provided system in Listing 1, build the table of static (immutable) system parameters, as if you were to use the Banker's Algorithm for deadlock avoidance. You should provide a value of $R(k)$ for the generic resource $k \in \{plates, glasses, \ldots\}$, and $C_i(k)$ for a

		Parameter	Resources				
			plates	glasses	forks	spoons	pair-chops
		$R(k)$	4	6	2	2	2
Roomies	ro_1	$C_1(k)$	1	1	1	1	1
	ro_2	$C_2(k)$	1	1	1	1	1
	ro_3	$C_3(k)$	1	1	1	1	1
	ro_4	$C_4(k)$	1	1	1	1	1

Table 3: Static parameters Roomies.

generic roomie-process $i \in \{1, \ldots, 4\}$.

Ans-1f Refer to Table 3

Problem 2

Two towns are located on the two sides of a river. The town west of the river is called Westland (W), while the town east of the river goes under the name of Eastville (E). Everyday, a continuous flow of people want to cross the river to go from W to E, and vice-versa. However, there is only a ferry boat that can transport people from one side to the other. The ferry is at any one time either on the W side, in transit, or on the E side. Moreover, when waiting on the E or W side, the ferry will never leave unless every single seat on the boat has been taken.
A generic person living in this twin-city context behaves as follows. Let's say that the person is initially on the W side. She/he approaches the W river bank wanting to go E. If the boat is not there, the person sleeps. If there is a seat available on the boat, the person gets on the boat and waits until all the seats have been taken. When all the seats have been taken, the ferry can leave and reach the other side. All the current passengers disembark before people waiting on the E side can embark and travel to W. A person that has reached W from E will perform some unspecified business in W before wanting to go back to E. Analogously, the same happens for people who are initially in W and want to travel to E. The cycle for each person, as well as for the ferry boat repeats.

a) Call N the number of seats available on the ferry boat. Write pseudo-code for a generic person in the twin-city context and for the ferry boat. The code should only use semaphores to perform synchronization. Collectively, the processes should follow the behavior described above. Be particularly careful to make sure that people waiting on one side do not start embarking before all the passengers already on the ferry have disembarked.

Ans-2a This is an open ended question. Any answer that works will get you full credit. Once possible solution is given in listing 3. This solution assumes the ferry is initially at the East bank.

Listing 3: A generic person and the ferry boat.

```
1
2  semaphore ferry_west = 0; //initialize to 0 so wait takes it negative
3  semaphore ferry_east = 0; //initialize to 0 so wait takes it negative
4  semaphore embark_allowed = N;
5  semaphore cross_river = 0;
6  semaphore done_transit = 0;
7
8  semaphore mutex = 1; //binary semaphore
9
10 int seatCount = 0;
11 int bankFerry = 0; //Ferry initialized to be on the East=0 side.
12
13 /*
14 * The following person is assumed to be initially on the East bank
15 * People on the West bank should have local bankPerson variable set to 1.
16 */
17 Process Person:
18         int bankPerson = 0; //1=West, 0= East
19         Repeat:
20                 get_to_bank();
21
22                 if(bankPerson)
23                         wait(ferry_west); //inherent sleep while waiting for ferry to arrive
24                 else
25                         wait(ferry_east);
26
27                 wait(embark_allowed);
28
29                 embark_on_ferry()
30
31                 wait(mutex)
32                 seatCount++
33                 if(seatCount == N){
```

```
34                      signal(cross_river) //to the ferry boat
35              }
36              signal(mutex)
37
38              wait(done_transit);
39
40              disembark_from_ferry()
41              bankPerson = ~bankPerson;  //toggle the local variable bankPerson
42
43              wait(mutex)
44              seatCount--;
45              if(seatCount == 0){
46                      for (i = 0; i < N; ++i)
47                              signal(embark_allowed)
48              }
49              signal(mutex)
50
51              do_business()
52      Forever
53
54  ================================================================================
55  Process FerryBoat:
56      Repeat:
57              if(bankFerry)
58                      for (i = 0; i < N; ++i)
59                              signal(ferry_west);
60              else
61                      for (i = 0; i < N; ++i)
62                              signal(ferry_east);
63
64              wait(cross_river);
65
66              cross_the_river();
67
68              reach_bank();
69
70              for (i = 0; i < N; ++i)
71                      signal(done_transit);
72
73              bankFerry = ~bankFerry;
74      Forever
```

		Parameter	Resources		
			R_1	R_2	R_3
		$R(k)$	10	5	7
Processes	P_1	$C_1(k)$	7	5	3
	P_2	$C_2(k)$	3	2	2
	P_3	$C_3(k)$	9	0	2
	P_4	$C_4(k)$	2	2	2
	P_5	$C_5(k)$	4	3	3

Table 4: Static parameters for the considered system.

		Parameter	Resources			Parameter	Resources		
			R_1	R_2	R_3		R_1	R_2	R_3
		$V(k)$	3	3	2				
Processes	P_1	$A_1(k)$	0	1	0	$N_1(k)$	7	4	3
	P_2	$A_2(k)$	2	0	0	$N_2(k)$	1	2	2
	P_3	$A_3(k)$	3	0	2	$N_3(k)$	6	0	0
	P_4	$A_4(k)$	2	1	1	$N_4(k)$	0	1	1
	P_5	$A_5(k)$	0	0	2	$N_5(k)$	4	3	1

Table 5: System state for considered system.

Problem 3

Assume a system of 5 processes $P_1, \ldots, P_5$ that share 3 resources $R_1, \ldots, R_3$. The static system parameters are provided in Table 4.

a) Complete Table 5 given the value of allocated resources $A_i(k)$ already reported in the table.

Ans-3a See answer in Table 5.

b) Is the state reported in Table 5 safe? If so, derive the sequence in which processes can complete.

Ans-3b Yes the state is safe. The order of completion from first to last is: P_2, P_4, P_5, P_1, P_3

c) Assuming the state in Table 5. If P_5 requests $Q_5 = (3, 3, 0)$, what is the temporary state of the system? Fill up Table 6 accordingly. Can this allocation request be granted?

Ans-3c No, the request cannot be granted as the request leads to an unsafe state. See answer in Table 6.

		Parameter	Resources			Parameter	Resources		
			R_1	R_2	R_3		R_1	R_2	R_3
		$V(k)$	0	0	2				
Processes	P_1	$A_1(k)$	0	1	0	$N_1(k)$	7	4	3
	P_2	$A_2(k)$	2	0	0	$N_2(k)$	1	2	2
	P_3	$A_3(k)$	3	0	2	$N_3(k)$	6	0	0
	P_4	$A_4(k)$	2	1	1	$N_4(k)$	0	1	1
	P_5	$A_5(k)$	3	3	2	$N_5(k)$	1	0	1

Table 6: Temporary system state to check request $Q_5 = (3, 3, 0)$.

Problem 4

Code: In this problem you will write a function to navigate complex multi-level structures that are based on the ability to crack MD5 hashes. The numbers resulting from the sequence of encountered simple hashes will compose the key to decode the secret location of the treasure.

a) Extend your `Pirate.java` class to navigate a multi-level treasure map, build up the list of hints and then use the full list of hints to discover the instructions to locate the treasure!

As usual the starting point is a list of hashes. Some of these hashes can be cracked right away (simple hints), while others can be cracked with compound hints. The main difference compared to HW7 is that before we only had two processing stages: (1) initial stage for simple hints; (2) second stage for compound hints. Now, we might have multiple subsequent processing stages that build on the result of the previous stages. Let us review this concept.

Starting from the first stage, we first attempt to crack the input hashes as if they are simple hints and use the same timeout mechanism to detect so-far uncrackable hashes (just like in HW7). This leads to a first list of integers $L_0 = \{A, B, C, D, \ldots\}$. Next some of the hashes that could be not previously cracked might be crackable with a compound hint of the form "$\alpha; k; \beta$" generated in the same way as we previously did—and with the same constraints between α, k, and β.

Completing the processing described above leads to the definition of a new list of integers $L_1 = \{k_1, k_2, k_3, \ldots\}$ where each k_i comes from a different compound hint. Note that none of the integers in L_0 can show up in L_1. Using the new list of integers some more (or potentially all) of the so-far uncrackable hashes can be cracked with a new compound hint built in the same way as before. That is, using a compound hint of the form "$k_x; h; k_y$" where $k_x < h < k_y$ and $k_x, k_y \in L_1$. If after this step there are still uncrackable hashes, one more round of processing is needed by defining a new list of integers L_2 etc. In this problem, all the input hashes are crackable! Thus, a valid solution should not leave any hash uncracked.

Consider now the various list of integers $L_0, L_1, L_2, \ldots$ constructed in the multiple processing stages. To find the instructions for the treasure, we need to take the union of all the integers in the list, and then sort the integers in ascending order. Finally, each number is the offset of a single character inside a cipher-text file passed in input.

Your job is to print, in order, the corresponding characters in the second file at the correct offsets. The resulting string will provide the final directions to find the treasure. Let us make a full example. Let's say that the input list of hashes is:

```
c81e728d9d4c2f636f067f89cc14862c
c9f0f895fb98ab9159f51fd0297e236d
d3d9446802a44259755d38e6d163e820
c51ce410c124a10e0db5e4b97fc2af39
cc397dd8486d3b0aec12fd25c76b19e2
3e66ce357af2da396c17631e706f1941
dd6cf119df0327fd2fc7500985284f59
```

After the first stage we have the intermediate result below, so $L_0 = \{2, 8, 10, 13\}$:

```
2
8
10
13
```

```
cc397dd8486d3b0aec12fd25c76b19e2
3e66ce357af2da396c17631e706f1941
dd6cf119df0327fd2fc7500985284f59
```

Next with the integers in L_0 we are able to crack two more hashes, so the partial result is the one below and $L_1 = \{5, 11\}$:

```
2;5;10
8;11;13
dd6cf119df0327fd2fc7500985284f59
```

Finally, the remaining hash can be cracked with the compound hash below:

```
5;7;11
```

Putting all the integers from the lists together, we have the following list of numbers: $\{2, 5, 7, 8, 10, 11, 13\}$. Now if the content of the input cipher-text is:

```
ljDshoDne{ :S)43 298r76Qt0
```

The solution is the string: "`Done :)`", comprised by the characters in the position indicated by the integers in the final list.

Extend the Java class `Pirate.java` that defines a method called `findTresure(...)` that is able to find the location of the treasure as described above. The **only** output of the code should be the final instructions to find the treasure. Apart from implementing the `findTreasure(...)` method, the class should also include a `public static void main(String [] args)` function. The `main(...)` function should accept 4 parameters from the calling environment. (1) The path to the input hash list is passed as the first parameter. (2) The second parameter passed to your code is the number of available CPUs. (3) The third parameter encodes the length of the timeout for (currently) uncrackable hashes expressed in milliseconds. (4) The fourth parameter is the path to the cipher-text to decode to find the final instructions for the treasure.

Submission Instructions: in order to submit this homework, please follow the instructions below for exercises and code.
The solutions for Problem 1-3 should be provided in PDF format, placed inside a single PDF file named `hw8.pdf` and submitted via Gradescope. Follow the instructions on the class syllabus if you have not received an invitation to join the Gradescope page for this class. You can perform a partial submission before the deadline, and a second late submission before the late submission deadline.

The solution for Problem 4 should be provided in the form of Java source code. To submit your code, place all the `.java` files inside a compressed folder named `hw8.zip`. Make sure they compile and run correctly according to the provided instructions. The first round of grading will be done by running your code. Use CodeBuddy to submit the entire `hw8.zip` archive at `https://cs-people.bu.edu/rmancuso/courses/cs350-fa20/scores.php?hw=hw8`. You can submit your homework multiple times until the deadline. Only your most recently updated version will be graded. You will be given instructions on Piazza on how to interpret the feedback on the correctness of your code before the deadline.