Answer 1

a) Nope, two roomies would never result in a starvation situation, since we have 3 cups and 4 plates, there will always be an extra cup, and two extra plates present so no one roommate will ever lack any of these. Furthermore, since there are two pairs of each of the utensils required to eat the courses, and no one roommate can eat the same course for both meals, that means that no matter what combination of courses the two roommates pick, they will never encounter a situation where they will forever be waiting for the utensils, in fact with this combination of utensils, they will never have to wait at all.

b) Yes! There is a scenario where one of the roommates starves to death. If all 4 roommates attempt to eat at once, they won't have enough glasses to serve all 4 roommates. As such, it is possible that one roommate constantly stays awaiting for a glass, and even when another roommate clean a glass, if the semaphore is not implemented in a fair way, it's possible that, due to the loop structure, the same roommate gets the glass back, instead of allowing the other waiting roommate to get it.

Furthermore, due to the lack of utensils, it's possible that, if all roommates pick the same combination of meals in the same order, we can see quite a wait time to get the utensils, but this will not cause starvation, as there is only a 0.000772 probability of this happening in any 1 round, which means that the the probability of this happening enough times in a row for every roommate is null, meaning there won't be any starvation, rather just some momentarily hungry and possibly cranky roommates waiting while another roommate finishes their dinner.

c) For Part A the answer remains the same, except that now there is a bigger margin of leftover glasses. As for Part B this does change our answer, with 6 glasses no roommates would ever need to wait to get glasses, which means that, as said before, there is still a small chance of roommates having to wait for a bit, but they won't be starving any time soon.

d) Well, with 4 roommates, using 6 glasses and 4 plates, we will see a maximum of 4 dirty plates at any given time, since there is the possibility of all 4 roommates eating at the same time. However this will only happen if the roommates roll the dice and get a combination of meal that allows them to eat all at once.

e) The current issue that might cause deadlocks is the fact that roommates only wash their utensils after eating both plates, which can cause a significant wait time, despite not causing starvation. Below is my code solution to reduce these deadlocks, and allow for easier access to utensils. The change I made was to the order of operations, making it so each roommate washes their meal-specific utensils after they have eaten each meal. This also means that the eating of the meal is done in two separate places instead of just one.

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       deafult:
35            print("Oh boy I rolled the wrong thing.\n");
36            exit();
37       }
38
39       eat_course_one_dinner(); /* 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_course_two_dinner(); /* 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 */
93
94          signal(plates)
95          signal(glasses)
96
97          /* Watch some Neshflix */
98          /* Sleep for 7ish hours */
99
100     Forever
```

f)

		Parameter	Resources				
			Plates	Glasses	Forks	Chopsticks	Spoons
		R(k)	4	6	2	2	2
Processes	P(1)	C1(k)	1	1	0.3	0.3	0.3
	P(2)	C2(k)	1	1	0.3	0.3	0.3
	P(3)	C3(k)	1	1	0.3	0.3	0.3
	P(4)	C4(k)	1	1	0.3	0.3	0.3

Answer 2

Listing 2: Twin Cities Problem.

```
/* Global SHARED variables */
semaphore eastWestSeats = N;
semaphore westEastSeats = N;


public Person {
    String name;
    String originSide;
    Ferry ferry;
    boolean running = True;

    public void stopExecution() {
        this.running = False;
    }

    public void crossRiver() {
        while (this.running) {
            this.wakeUpAndGetReady();
            this.goToDocs();

            if (originSide == "East") {
                eastWestSeats.acquire();

                while (ferry.isMoving) {
                    this.wait();
                }

                eastWestSeats.release();

                this.doBusiness();
                this.gotToDocs();

                westEastSeats.acquire();

                while (ferry.isMoving) {
                    this.wait();
                }
```

```
38
39                  westEastSeats.release();
40              } else {
41                  westEastSeats.acquire();
42
43                  while (ferry.isMoving) {
44                      this.wait();
45                  }
46
47                  westEastSeats.release();
48
49                  this.doBusiness();
50                  this.gotToDocs();
51
52                  eastWestSeats.acquire();
53
54                  while (ferry.isMoving) {
55                      this.wait();
56                  }
57
58                  eastWestSeats.release();
59              }
60
61              this.goHomeAndSleep();
62          }
63      }
64  }
65
66
67  public Ferry() {
68      String ID;
69      boolean running = True;
70      boolean moving;
71      String currentSide;
72      int travelTime
73
74      public boolean stopExecution() {
75          this.running = False;
76      }
77
78      public boolean isMoving() {
79          return moving;
```

```
80      }
81
82      public void crossModel() {
83          while (this.running) {
84              if (currentSide == "East") {
85
86                  /* Make sure there are no lingering permits in the opposite
87                     semaphore (ie: kick any stragglers out of the ferry) */
88                  while (westEastSeats.hasPermits()) {
89                      westEastSeats.release();
90                  }
91
92                  /* Load the ferry until full */
93                  while (eastWestSeats.hasPermits()) {
94                      this.wait();
95                  }
96
97                  /* Travel */
98                  long arrivalTime = System.currentTimeInMillis() + travelTime;
99
100                 this.moving = True;
101                 while (System.currentTimeInMillis() < arrivalTime) {
102                     this.travel();
103                 }
104                 this.moving = False;
105
106                 /* Remove passengers from the boat (ps: this shouldn't be needed,
107                    but seems like good practice for any stuck passengers) */
108                 while (eastWestSeats.getAvailablePermits() != N) {
109                     eastWestSeats.release();
110                 }
111
112                 /* Once all passengers have left, change current side
113                    & load the proper semaphore */
114                 this.currentSide = "West";
115                 while (westEastSeats.getAvailablePermits() != N) {
116                     westEastSeats.release();
117                 }
118
119             } else {
120
121                 while (eastWestSeats.hasPermits()) {
```

```
122                         eastWestSeats.release();
123                     }
124
125                     while (westEastSeats.hasPermits()) {
126                         this.wait();
127                     }
128
129                     long arrivalTime = System.currentTimeInMillis() + travelTime;
130
131                     this.moving = True;
132                     while (System.currentTimeInMillis() < arrivalTime) {
133                         this.travel();
134                     }
135                     this.moving = False;
136
137                     while (westEastSeats.getAvailablePermits() != N) {
138                         westEastSeats.release();
139                     }
140
141                     this.currentSide = "East";
142                     while (eastWestSeats.getAvailablePermits() != N) {
143                         eastWestSeats.release();
144                     }
145
146                 }
147             }
148         }
149 }
```

Answer 3

a)

		Parameter	Resources			Parameter	Resources		
			R1	R2	R3		R1	R2	R3
		V(k)	3	3	2				
Processes	P1	A1(k)	0	1	0	N1(k)	7	4	3
	P2	A2(k)	2	0	0	N2(k)	1	2	2
	P3	A3(k)	3	0	2	N3(k)	6	0	0
	P4	A4(k)	2	1	1	N4(k)	0	1	1
	P5	A5(k)	0	0	2	N5(k)	4	3	1

b) Yes, the state reported in Table 2 is safe, as we can complete the processes using the following sequence of allocation: P_4, P_2, P_3, P_1, P_5.

a)

		Parameter	Resources			Parameter	Resources		
			R1	R2	R3		R1	R2	R3
		V(k)	0	0	2				
Processes	P1	A1(k)	0	1	0	N1(k)	7	4	3
	P2	A2(k)	2	0	0	N2(k)	1	2	2
	P3	A3(k)	3	0	2	N3(k)	6	0	0
	P4	A4(k)	2	1	1	N4(k)	0	1	1
	P5	A5(k)	3	3	2	N5(k)	1	0	1

The system could not grant this allocation request, as, if it does, it will only have 2 units of R_3 left to allocate (and none of R_2 and R_1), which means it reaches a deadlock and cannot complete any of the processes.