1a. Process Q must go first for n to meet the condition for Process Q to start looping.

Process P	Process Q	n
p1: while n < 1	**q1: while n >= 0**	1
p1: while n < 1	**q2: n = n - 1**	1
p1: while n < 1	**q1: while n >= 0**	0
p1: while n < 1	**q2: n = n - 1**	0
p1: while n < 1	q1: while n >= 0	-1
p2: n = n + 1	q1: while n >= 0	-1
p1: while n < 1	q1: while n >= 0	0
p2: n = n + 1	q1: while n >= 0	0
p1: while n < 1	**q1: while n >= 0**	1
p1: while n < 1	**q2: n = n - 1**	1
p1: while n < 1	q1: while n >= 0	0
p2: n = n + 1	q1: while n >= 0	0
p1: while n < 1	q1: while n >= 0	1
exited	**q1: while n >= 0**	1
exited	**q2: n = n - 1**	1
exited	**q1: while n >= 0**	0
exited	**q2: n = n - 1**	0
exited	**q1: while n >= 0**	-1
exited	exited	-1

1b. The processes alternate so that n remains within 1 to 0, which is what is necessary to create an infinite loop for the two processes.

Process P	Process Q	n
p1: while n < 1	**q1: while n >= 0**	1
p1: while n < 1	**q2: n = n - 1**	1
p1: while n < 1	q1: while n >= 0	0
p2: n = n + 1	q1: while n >= 0	0
p1: while n < 1	**q1: while n >= 0**	1
p1: while n < 1	**q2: n = n - 1**	1
p1: while n < 1	q1: while n >= 0	0
p2: n = n + 1	q1: while n >= 0	0
...	...	...

2a. The program terminates easily, but there is a noticeable pattern of where n is always going to be either 0 or 1, nothing more, nothing less.

Process P	Process Q	n	flag	n == 0
p1: while flag == false	q1: while flag == false	0	FALSE	TRUE
p2: n = 1 - n	q1: while flag == false	0	FALSE	TRUE
p1: while flag == false	q1: while flag == false	1	FALSE	FALSE
p2: n = 1 - n	q1: while flag == false	1	FALSE	FALSE
p1: while flag == false	q1: while flag == false	0	FALSE	TRUE
p1: while flag == false	**q1: while flag == false**	0	FALSE	TRUE
p1: while flag == false	**q2: if n == 0**	0	FALSE	TRUE
p1: while flag == false	**q3: flag = true**	0	FALSE	TRUE
p1: while flag == false	**q1: while flag == false**	0	TRUE	TRUE
p1: while flag == false	exited	0	TRUE	TRUE
exited	exited	0	TRUE	TRUE

2b. When the program terminates, n will either be 1 or 0.

2c. There is always the possibility of Process P just being unfair and never giving the chance for Process Q to execute, making it so that flag never gets the chance to be changed into true, and remains false indefinitely. There is also the scenario where right after Process P finishes executing one loop, Process Q executes its entire loop, and so forth.

2d. The program does not terminate for all **fair** scenarios. There is the possibility of an infinite loop if p2 were to be interleaved between q1 and q2. Q2 checks if n is equal to 0, so when n is 0, p2 could be called which changes the value of n, making it a context switch and flag will never be set to true.

3a. The terminating scenario is straightforward.

Process P	Process Q	n	flag
p1: while flag == false	q1: while n == 0 {}	0	FALSE
p2: n = 1 - n	q1: while n == 0 {}	0	FALSE
p1: while flag == false	**q1: while n == 0 {}**	1	FALSE
p1: while flag == false	**q2: flag = true**	1	FALSE
p1: while flag == false	exited	1	TRUE
exited	exited	1	TRUE

3b. This program has one fair scenario where it does not terminate. The scenario may be a bit hard to explain through text, so here is a scenario table. After the last line before the …, it repeats back from the very first line. That is the one fair scenario where the program never terminates.

Process P	Process Q	n	flag
p1: while flag == false	q1: while n == 0 {}	0	FALSE
p2: n = 1 - n	q1: while n == 0 {}	0	FALSE
p1: while flag == false	**q1: while n == 0 {}**	1	FALSE
p1: while flag == false	q1: while n == 0 {}	1	FALSE
p2: n = 1 - n	**q1: while n == 0 {}**	1	FALSE
p2: n = 1 - n	q1: while n == 0 {}	1	FALSE
…	**…**	0	FALSE

4a.

Condition 1: if (a[i] < b[j] or a[i] < c[k])

Condition 2: if (b[j] < a[i] or b[j] < c[k])

Condition 3: if (c[k] < a[i] or c[k] < b[j])

4b.

```
global int i, j, k;

Main:
    array a, b, c;
    bool done = false;




Thread looper (array &a, array &b, array &c, bool &done):

    while (true) {
        if (a[i] == b[j] && a[i] == c[k]) {
            std::cout << "i: " << i << " j: " << j << " k: " << k << std::endl;
            done = true;
            break;
        }
    }
```

```
Thread a (array &a, array &b, array &c, bool &done):

while (!done) {
    if (a[i] < b[j] || a[i] < c[k]) {
        i++;
    }
}

Thread b (array &a, array &b, array &c, bool &done):

while (!done) {
    if (b[j] < a[i] || b[j] < c[k]) {
        j++;
    }
}

Thread c (array &a, array &b, array &c, bool &done):

while (!done) {
    if (c[k] < a[i] || c[k] < b[j]) {
        k++;
    }
}
```

This is pseudocode so I don't think I need to be too formal with the code and have it corrected, if you can see what is happening.

So, there are a total of 4 threads, 3 of which are responsible for each array. What happens in each thread is that it does its corresponding conditional expression that it normally would in a single threaded program but does so in a while loop. Eventually, it'll keep looping infinitely until either a value in another thread changes, or the looper thread sets the done variable to true.

What the looper thread does is that it keeps looping and seeing if the values are equal. If they are, it prints out the values and sets the done variable to true, and then breaks the loop. In turn, this also breaks the loop of the other threads as well since the done variable is now true.

In my mind, I think this code should run faster than the traditional single threaded code since each thread is doing their stuff concurrently and nothing is getting really delayed.