Assignment 3: multi-threading in assembly

Assignment Description

Your code will be written **entirely in assembly language**. No C code is allowed, and no usage of C standard library functions other than those specifically noted below. The only C standard library code allowed is the initialization code at "_start" which calls main( ), usage of printf and fprintf functions to print floating point numbers and integers (and whatever else you need to print), and sscanf in order to convert the command line arguments. Use of malloc( ), calloc( ) and free( ) is also allowed, to handle your dynamic memory allocation.

You are to write a simple user-mode co-routine ("multi-thread") simulation, using the co-routine mechanism code described and provided in class. The goal is further proficiency in assembly language, especially as related to floating point, and understanding stack manipulations, as needed to create a co-routine (equivalently thread) management scheme.

Program Goal

Drones Battle Royale

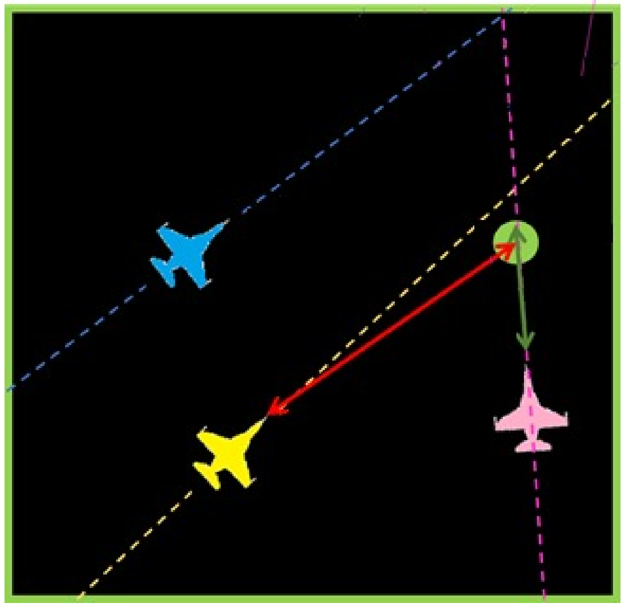
You will implement a multi-drones single-target game.

We have a 100x100 game board, on which a group of N drones hunt the same target from different points of view and from different distances. Each drone tries to detect where is the target on the game board, in order to destroy it. Drones may destroy the target only if the target is no more than some given distance from the drone. When the current target is destroyed, some new target appears on the game board in some randomly chosen place. Every R rounds one of the drones is "eliminated" (in a manner which is described below), the last drone left in the game is the winner of the game.

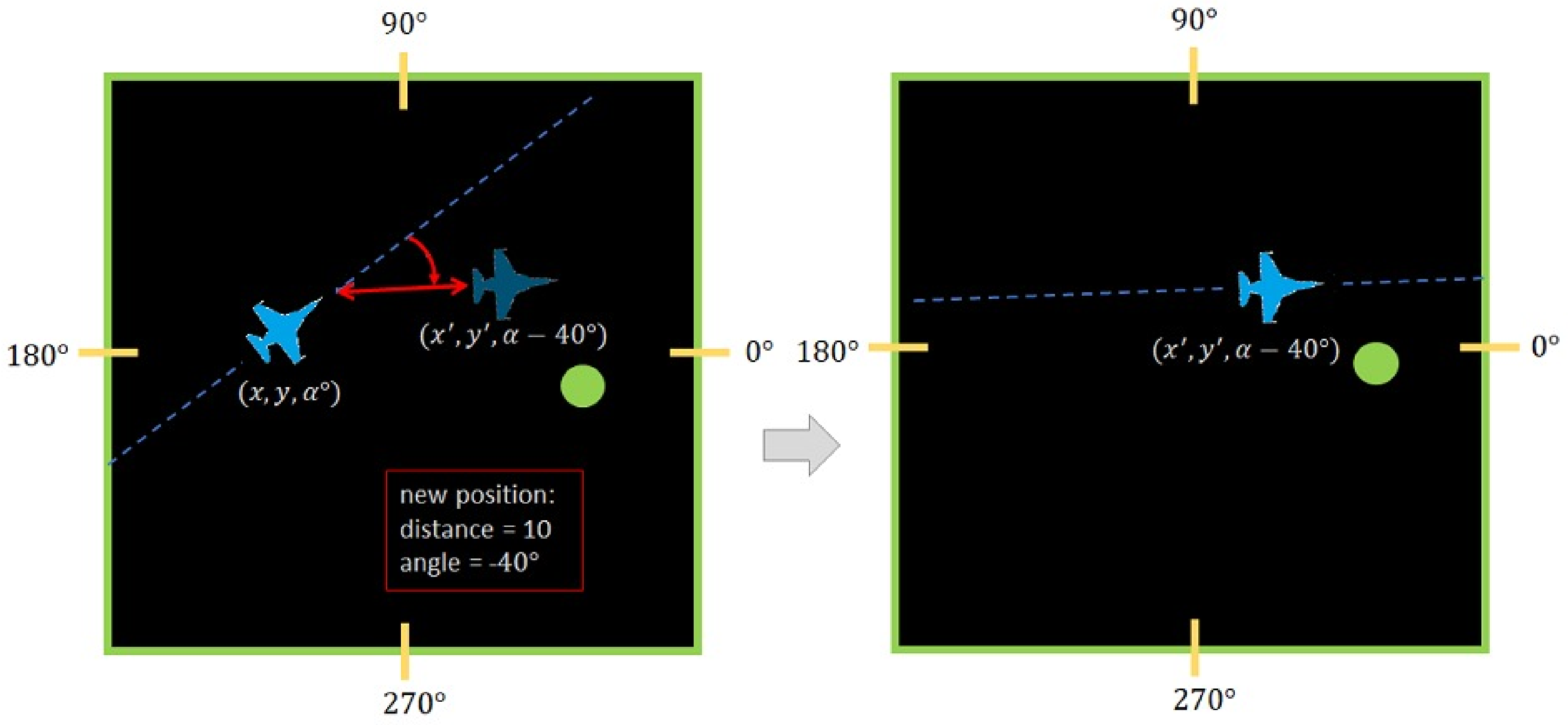
Each drone has a two-dimensional position on the game board: (coordinate x, coordinate y), and direction (angle from x-axis)*, two drones may occupy the same position. Moreover each drone has a current speed which is increased or decreased in each round. All the above state variables are represented by floating point numbers. Drones move according to their speed using their curret heading, from their current place. They then randomly change their speed and heading, as described below, before the next move. After each movement, a drone calls the **mayDestroy(…)** function with its new position on the board. The **mayDestroy(…)** function returns TRUE if the caller drone may destroy the target, otherwise returns FALSE. If the current target is destroyed, a new target is created at a random position on the game board. Note that drones do not know the coordinates of the target on the board game. On the example below, the blue and the yellow drones are very far from the target and cannot destroy it, and the pink drone is close enough to it. The pink drone can destroy the target.

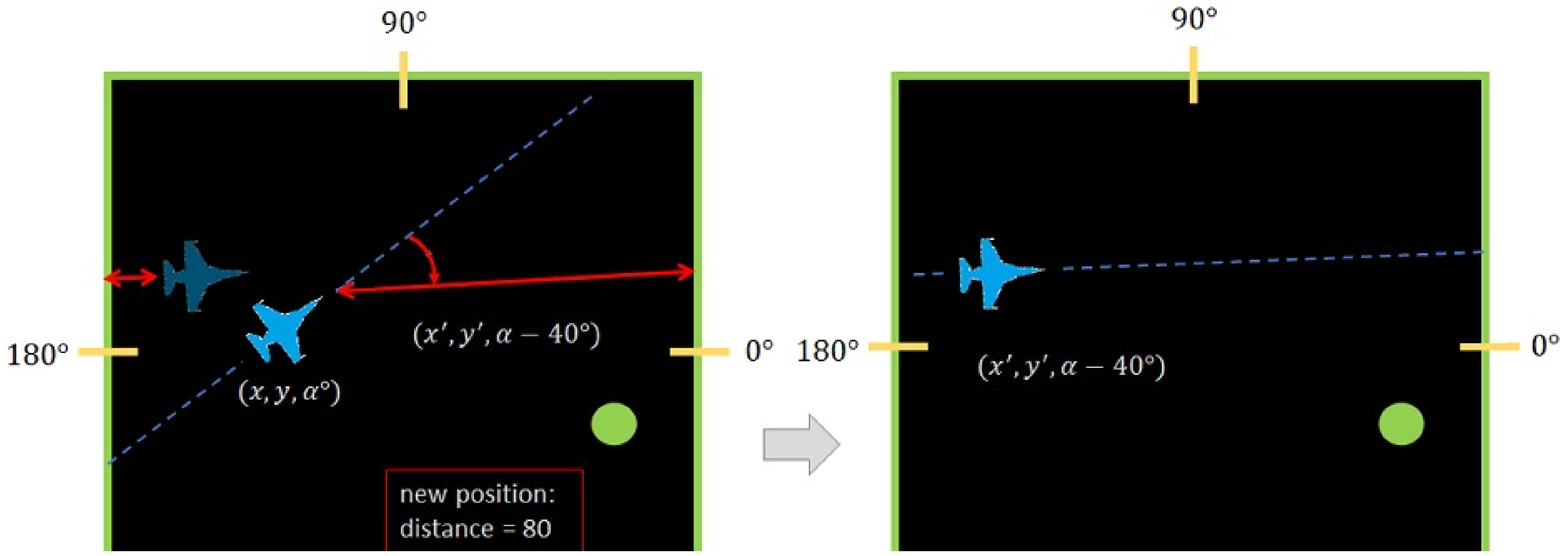
*these coordinates are represented using floating point numbers. two drones are allowed to have the same coordinates

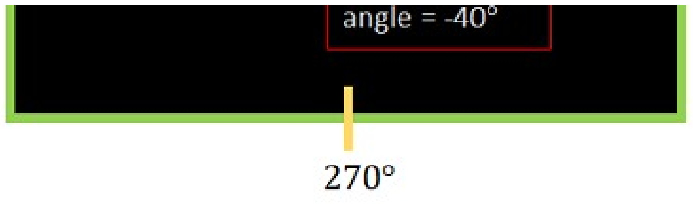
Please note: the target is just a point, not a circle as it appears in the figures. The painting of a circle is just for the convenience of illustration. Also note that drones are illustrated as airplanes but they do not have all the appropriate functionality of airplanes.

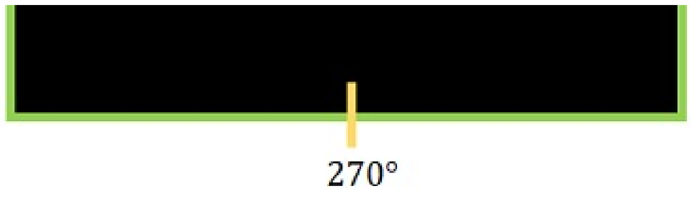


When a drone moves from its current position to a new position, it may happen that the distance move makes the drone cross the game field border. We treat drone motion as if on a torus. That is, when the next location is greater than the board width or height, or is negative in either axis (this requires checking 4 conditions), subtract or add the torus cycle (100 in this example) if needed. On the first figure below, we see a simple movement of the drone, and on the second figure we may see the movement that would move the drone out of the right border, and instead it is **wrapped around** to the left border of the game board. Speed may also change beyond the bounds (0 to 100), in this case we **clamp** (cut off) the value to the repective bound, such as: having current speed 96.4 and increase by 6.5 makes the new speed 100.


90°
180°
0°
270°
$(x, y, \alpha°)$
$(x', y', \alpha - 40°)$
new position:
distance = 10
angle = -40°
90°
180°
0°
270°
$(x', y', \alpha - 40°)$

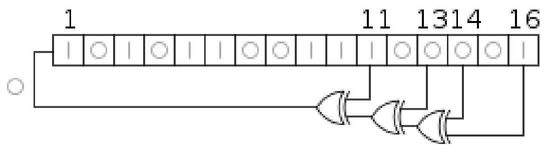

90°
180°
0°
$(x, y, \alpha°)$
$(x', y', \alpha - 40°)$
new position:
distance = 80
90°
180°
0°
$(x', y', \alpha - 40°)$





How to generate a pseudo-random number

You should use Linear-feedback Shift Register method to get a random number in Assembly. A l**inear-feedback shift register (LFSR)** is a shift register whose input bit is a linear function of its previous state. LFSR is a shift register whose input bit is driven by the XOR of some bits of the overall shift register value. The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. We would use Fibonacci LFSR version. The figure below describes a 16-bit Fibonacci LFSR. The feedback tap numbers shown correspond to a primitive polynomial in the table, so the register cycles through the maximum number of 65535 states excluding the all-zeroes state. The state shown, 0xACE1 will be followed by 0x5670.



The bit positions that affect the next state are called the taps. In the diagram the taps are [16,14,13,11]. The rightmost bit of the LFSR is called the output bit. The taps are XOR'd sequentially with the output bit and then fed back into the leftmost bit. The sequence of bits in the rightmost position is called the output stream. The bits in the LFSR state that influence the input are called taps.

Note that the XOR function is as follows:

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

scaling process of random number:

- suppose you generated integer random number x in unsigned range [0,MAXINT]
- suppose you need to scale this number to your [0,100] board size to calculate some coordinate

- then, just execute the following calculation: x_scaled = float point value of (x / MAXINT) * 100

Where we use Threads in the game

The drones and target are managed by threads, **one thread each**. We call thread a cooperating routine, or co-routine for short. In addition, we have a co-routine for the scheduler that manages when to run the other threads. A specialized co-routine called the printer prints the current state of the game board. A drone’s co-routine number is its id, which should be visible to it as an argument at activation (either in a register, or on its own stack, your choice). We use drone’s id for scheduling and for printing. The program begins by creating the initial state configuration of drones and the initial state of the target. The program initializes appropriate drones and target, and control is then passed to a scheduler co-routine which decides the appropriate scheduling for the co-routines. The scheduling algorithm for drones is ROUND ROBIN, meaning that co-routines are scheduled in a loop: 1,2,3,…,N,1,2,3,…N,… and so on. The printer co-routine should print the current state of the game board each K steps, where step means an execution step of one drone.

The function of a drone co-routine is as follows:

```
(*) Generate random heading change angle  Δα       ; generate a random number in range [-60,60] degrees, with 16 bit resolution
(*) Generate random speed change Δa         ; generate random number in range [-10,10], with 16 bit resolution
(*) Compute a new drone position as follows:
    (*) first move speed units at the direction defined by the current angle, wrapping around the torus if needed.
        For example, if speed=60 then move 60 units in the current direction.
    (*) then change the current angle to be α + Δα, keeping the angle between [0, 360] by wraparound if needed
    (*) then change the current speed to be speed + Δa, keeping the speed between [0, 100] by cutoff if needed
(*) Do forever
    (*) if mayDestroy(…) (check if a drone may destroy the target)
        (*) destroy the target
        (*) resume target co-routine
    (*) Generate random angle Δα       ; generate a random number in range [-60,60] degrees, with 16 bit resolution
    (*) Generate random speed change Δa    ; generate random number in range [-10,10], with 16 bit resolution
    (*) Compute a new drone position as follows:
        (*) first, move speed units at the direction defined by the current angle, wrapping around the torus if needed.
        (*) then change the new current angle to be α + Δα, keeping the angle between [0, 360] by wraparound if needed
        (*) then change the new current speed to be speed + Δa, keeping the speed between [0, 100] by cutoff if needed
    (*) resume scheduler co-routine by calling resume(scheduler)
(*) end do
```

The function of a target co-routine is as follows:

```
(*) call createTarget() function to create a new target with random coordinates on the game board
(*) switch to the co-routine of the "current" drone by calling resume(drone id) function
```

The function createTarget() is as follows:

```
(*) Generate a random x coordinate
(*) Generate a random y coordinate
```

The loop in the function of a scheduler co-routine is as follows:

Note, the code below was modified to fix a minor inconsistency. This is not crucial, as the results will not be auto-tested. This is a possible implementation of round-robin scheduling of "active" drones, and the printing requirement.

```
(*) start from i=0
(*)if drone (i%N)+1 is active
    (*) switch to the i’th drone co-routine
(*) if i%K == 0 //time to print the game board
    (*) switch to the printer co-routine
(*) if (i/N)%R == 0 && i%N ==0 //R rounds have passed
    (*) find M - the lowest number of targets destroyed, between all of the active drones
    (*) "turn off" one of the drones that destroyed only M targets.
(*) i++
(*) if only one active drone is left
    (*)print The Winner is drone: <id of the drone>
    (*) stop the game (return to main() function or exit)
```

The function of a printer co-routine is as follows:

```
(*) print the game board according to the format described below
(*) switch back to a scheduler co-routine
```

Command line arguments

Note that the game border size is pre-defined to be 100 x 100.

Your program should get the following command-line arguments (written in ASCII as usual, every argument is 4 bytes in size):

- N<*int*> – number of drones
- R<*int*> - number of full scheduler cycles between each elimination
- K<*int*> – how many drone steps between game board printings
- d<*float*> – maximum distance that allows to destroy a target
- seed<*int*> - seed for initialization of LFSR shift register

```
> ass3 <N> <R> <K> <d> <seed>
```

For example: `> ass3 5 8 10 30 15019`

*you can assume legal input

Initial configuration of the game

Initial configuration is calculated (pseudo)-randomly with 16 bit resolution (for each required number) before the game begins, by the main() function. What this means is: generate a 16-bit pseudo-random integer by using the LFSR to generate 16 **new** pseudo-random bits (for each required number, shift the register 16 time, computing a new radnom bit per shift), and scale this pseudo-random number to fit the desired range. All coordinates should be inside the board (may be also on the board borders). Initial angle values are floating point in the range [0,360], but note that you need to convert angles into radians if you want to use the SIN, COS, or SINCOS instructions (recommended). All other values are also float point numbers.

You should allocate a dynamic array of drones, which would contain drones' current positions, speeds, headings, and scores. Every co-routine will be able to access this array for reading and writing. After reading the arguments and the file, you need to allocate space and initialize the co-routine structures. Two additional co-routines should be initialized: scheduler and printer.

configuration order

1. initialize the target
2. initialize the drones

* random coordinate generation order is : x coordinate, y coordinate (and for the drones the speed, and the angle is last)

How to print the game board

Each K drone steps the game board should be printed by printer co-routine. All floating point numbers to be printed using %f or %lf with 2 digits after the decimal point, all angles to be printed in degrees (for readability). Each printed line should end with a newline character. The printing is in the following format:

```
x,y                                   ; this is the current target coordinates
1,x_1,y_1,α_1,speed_1,numOfDestroyedTargets_1    ; the first field is the drone id
2,x_2,y_2,α_2,speed_2,numOfDestroyedTargets_2    ; the fifth field is the number of targets destroyed by the drone
…
N,x_N,y_N,α_N,speed_N,numOfDestroyedTargets_N
```

Important implementation requirements

- A scheduler co-routine MUST be exclusively written in a separate file, the actual control transfer (context switch) should be done with the resume mechanism. Hence, label resume would be declared as extern to the scheduler, and register ebx would be used to transfer control.
- Different implementations (not ROUND ROBIN) of the scheduler will be optionally examined by the checkers
- Make all possible variables global in order not to pass them as arguments to your functions

Submission Instructions

You are to submit a single zip file containing ass3.s (file which contains main() function, common functions, and global variables), scheduler.s, printer.s, drone.s, and target.s files (no extra files are allowed). Your executable must be named ass3. Make sure you follow the coding and submission instructions correctly. **Submissions which deviate from these instructions will not be graded!**

Good Luck!