Master

1. Initialize semaphore A to 1
2. Initialize semaphore B to 1
3. Initialize Semaphore B to 0

A

1. while
2. P(X)
3. Print A
4. V(Y)
5. Endwhile

B

1. While
2. P(X)
3. Print B
4. V(Y)
5. Endwhile

C

1. While
2. P(Y)
3. Print C
4. V(X)
5. Endwhile

In this implementation, semaphore x is set to 1, and semaphore y is set to 0, so whichever process calls semaphore y, won’t be able to go first no matter what, in this case it is C. A and B share the same same semaphore that is originally set to 1, so only 1 of the two processes can go at once. When either of the ones print their letter, they release process C so that it can run, which in turn releases the next process after (either A or B) to go. This process doesn’t exhibit deadlock because the resource involved is shareable. This doesn’t exhibit starvation because none of the processes are favored over one another, each process has to take its turn in order to be run again.

Master

1. Initialize Semaphore A0
2. Initialize Semaphore A1
3. Initialize Semaphore A2
4. Initialize Semaphore A3
5. Initialize Semaphore A4
6. Initialize Semaphore A5
7. Initialize shared bool A0\_A0A1=false
8. Initialize shared bool B0\_B0B1=false
9. Initialize shared bool B2\_B1B2A2=false

RobotA

1. loop repeatedly
2. compute briefly
3. P(A0)
4. A0\_A0A1=true
5. occupy A0
6. A0\_A0A1=false
7. V(A0)
8. return to base
9. compute a little while
10. P(B0)
11. B0\_B0B1=true
12. occupy B0
13. B0\_B0B1=false
14. V(B0)
15. return to base

RobotB

1. loop repeatedly
2. calculate for a moment
3. P(B0)
4. B0\_B0B1=true
5. occupy B0
6. B0\_B0B1=false
7. V(B0)
8. return to base
9. calculate a little more
10. if B0\_B0B1==false and B2\_B1B2A2==false:
11. P(B1)
12. occupy B1
13. V(B1)
14. endif
15. return to base

RobotC

1. loop repeatedly
2. await a shipment
3. if A0\_A0A1==false:
4. P(A1)
5. occupy A1
6. V(A1)
7. endif
8. return to base
9. await a 2nd shipment
10. if B0\_B0B1==false and B2\_B1B2A2==false:
11. P(B1)
12. occupy B1
13. V(B1)
14. endif
15. return to base
16. endloop

RobotD

1. loop repeatedly
2. if A0\_A0A1==false:
3. P(A0)
4. A0\_A0A1=true
5. occupy A0
6. A0\_A0A1=false
7. V(A0)
8. endif
9. return to base
10. intermission to retool
11. if B2\_B1B2A2==false:
12. P(B2)
13. B2\_B1B2A2=true
14. occupy B2
15. B2\_B1B2A2=false
16. V(B2)
17. endif
18. return to base
19. download new instructions
20. endloop

RobotE

1. loop repeatedly
2. check inventory
3. if A0\_A0A1==false:
4. P(A1)
5. occupy A1
6. V(A1)
7. endif
8. return to base
9. do a quick calculation
10. if B2\_B1B2A2==false:
11. P(A2)
12. occupy A2
13. V(A2)
14. endif
15. return to base
16. endloop

explanation: In order to keep each process from stepping on the same square as the other processes, I created a mutex semaphore initialized to one for each square. So a process occupies that square, they lock the square by locking the semaphore from other processes. I used shared Booleans flags to follow each special rule. A0\_A0A1 is set to true if a process is occupying A0, and lets other processes know that it can’t occupy A0 or A1 when it is occupied. B0\_B0B1 is set to true if a process is occupying B0, and lets other processes know that they can’t occupy B0 or B1 if it is occupied. B2\_B1B2A2 is set to true if a process is set to true if B2 is occupied, and lets other process know the can’t occupy B1, B2, or A2.

In this implementation, deadlocks won’t occur, but starvation may occur. This can happen if one of the squares stay on a square that hinders other robots from occupying other squares. For example if a robot is stuck on A0, then A1 can never be occupied.