Examples of Concurrent Programming Exam Questions Using Harmony

by Robbert van Renesse, Cornell University

1. This is not a race. Or is it?

Which of the following programs suffers from a *data race*? Write “Y” (Yes) or “N” (No) in the box below each program.

```
x = 1
y = 0
z = 0

def f():
    x = 2

def g():
    y = x

spawn f()
spawn g()
```

a) ☐

```
x = 1
y = 0
z = 0

def f():
    y = x

def g():
    z = x

spawn f()
spawn g()
```

b) ☐

```
import synch

lock = synch.Lock()
x = 1
y = 0
z = 0

def f():
    synch.acquire(?lock)
    x = 2
    synch.release(?lock)

def g():
    synch.acquire(?lock)
    y = x
    synch.release(?lock)

spawn f()
spawn g()
```

c) ☐

```
import synch

lock = synch.Lock()
x = 1
y = 0
z = 0

def f():
    synch.acquire(?lock)
    x = 2
    synch.release(?lock)

def g():
    y = x

spawn f()
spawn g()
```

d) ☐

2. I'd rather be skating...

Lynah has a policy that does not allow skaters on the ice when their Zamboni is resurfacing the ice. They have asked our help to enforce this. Below is a Harmony program that simulates their Zamboni and the skaters.

```
1  from synch import Lock, acquire, release
2
3  sequential zamboni_on_ice, nskaters_on_ice
4
5  const NSKATERS = 10
6
7  zamboni_on_ice = False
8  nskaters_on_ice = 0
9  lock1  = Lock()
10 lock2  = Lock()
11
12 def zamboni():
13     while choose { False, True }:
14         acquire(?lock1)
15         zamboni_on_ice = True
16         # Zamboni goes onto the ice,
17         # resurfaces, and leaves the ice
18         zamboni_on_ice = False
19         release(?lock1)
20
21 def skater():
22     acquire(?lock2)
23     if nskaters_on_ice == 0:
24         acquire(?lock1)
25     nskaters_on_ice += 1
26     release(?lock2)
27     # Skater goes onto the ice,
28     # skates, and leaves the ice
29     acquire(?lock2)
30     nskaters_on_ice −= 1
31     if nskaters_on_ice == 0:
32         release(?lock1)
33     release(?lock2)
34
35 spawn zamboni()
36 for i in { 1 .. NSKATERS }:
37     spawn skater()
```

You have been asked to analyze the program without running it. In particular, on the following page there is a list of proposed invariants (i.e., logical statements that are always true *after initialization* in any execution). You have to say whether the statement is always true, always false, or neither (again, *after initialization*).

Reminder: False implies anything. Therefore, the statement "if 1 == 2 then 3 == 4" is an invariant because it is always true.

Zamboni is on the ice only here, in lines 16 and 17

This skater is on the ice only here, in lines 27 and 28

Place one checkmark **in each row** in *one* of the three columns *("neither" means sometimes True and sometimes False)*		always True	always False	neither
a)	if the Zamboni is on the ice (i.e., lines 16 and 17), then *zamboni_on_ice* == True			
b)	if *zamboni_on_ice* == True, then the Zamboni is on the ice			
c)	if the Zamboni is on the ice, then *nskaters_on_ice* == 0			
d)	*zamboni_on_ice* implies (*nskaters_on_ice* == 0)			
e)	(*nskaters_on_ice* == 0) implies *zamboni_on_ice*			
f)	if there is a skater on the ice (i.e., lines 27 and 28), then *nskaters_on_ice* > 0			
g)	if there are N skaters on the ice, then *nskaters_on_ice* $\leq N$			
h)	if there are N skaters on the ice, then *nskaters_on_ice* $\geq N$			
i)	if *nskaters_on_ice* $== N$, then there are **at least** N skaters on the ice			
j)	if *nskaters_on_ice* $== N$, then there are **at most** N skaters on the ice			
k)	if *lock2* is acquired by the Zamboni thread, then professor RVR is unicycling on the moon wearing a top hat			
l)	if *lock2* is held, then *zamboni_on_ice* == False			
m)	if there is a skater on the ice, then *lock1* is held			
n)	if there is a skater on the ice, then *lock2* is held			
o)	if the Zamboni is on the ice, then *lock1* is held			
p)	if the Zamboni is on the ice, then *lock2* is held			
q)	there is at most one skater on the ice			

3. I Need Some Privacy

There is a lot of demand for the Gates Zoom Booths, but only limited availability. A study of the situation is ordered, and a programmer who has their degree from the renowned MITT (Massachusetts Institute of Thread Technology) models the booths in Harmony as shown

```
1   from synch import Lock, acquire, release
2
3   sequential n_occupied # load&store are atomic
4
5   const N_BOOTHS = 2
6
7   booths = [Booth(),] * N_BOOTHS # list of booths
8   n_occupied = 0 # number of occupied booths
9
10  invariant 0 <= n_occupied <= N_BOOTHS
11  finally n_occupied == 0
12
13  def Booth():
14      result = { .lock: Lock() }
15
16  def booth_enter(b):
17      acquire(?booths[b].lock)
18      n_occupied = n_occupied + 1
19
20  def booth_exit(b):
21      n_occupied = n_occupied − 1
22      release(?booths[b].lock)
```

```
26  const N_STUDENTS = 3
27
28  def student():
29      let b = choose { 0 .. N_BOOTHS − 1 }:
30          booth_enter(b)
31          # zoom with bff
32          booth_exit(b)
33
34  for i in { 1 .. N_STUDENTS }:
35      spawn student()
```

Lines 26-35 show N_STUDENTS student() threads being spawned, each of which chooses a booth to enter. The code on the right shows the booth_enter() and booth_exit() functions, each of which take a booth number (0 through N_BOOTHS – 1) as argument. The code maintains an *n_occupied* variable that is stored in sequentially consistent memory, meaning that load and store operations are atomic, and operations are not delayed. The statement in Line 7 creates an array with N_BOOTHS elements. Each booth has a lock that protects access to the booth so that only one student can enter at a time.

Unfortunately, it was found that the code sometimes fails. In particular, the invariant in Line 10 is sometimes violated and *n_occupied* is not always 0 when all students threads have terminated. This depends on the values of N_BOOTHS and N_STUDENTS.

a) Fill in the table to the right with YES and NO.

N_BOOTHS	N_STUDENTS	FAILS?
1	3	
2	1	
2	2	
2	3	
3	2	

A graduate of Cornell CS 4410/5410 is hired to write a better model. Their program is shown below. Indeed, it seems to work for a variety of choices for positive N_BOOTHS and N_STUDENTS.

```
1   from synch import Lock, acquire, release
2
3   const N_BOOTHS = 2
4
5   booths = [Booth(),] * N_BOOTHS # list of booths
6
7   finally all(booths[b].free for b in { 0 .. N_BOOTHS − 1 })
8
9   def Booth():
10      result = { .lock: Lock(), .free: True }
11
12  def booth_enter(b):
13      acquire(?booths[b].lock)
14      # ...
15      booths[b].free = False
16      # ...
17
18  def booth_exit(b):
19      # ...
20      booths[b].free = True
21      # ...
22      release(?booths[b].lock)
```

```
26  const N_STUDENTS = 3
27
28  def student():
29      let b = choose { 0 .. N_BOOTHS − 1 }:
30          booth_enter(b)
31          # zoom with bff
32          booth_exit(b)
33
34  for i in { 1 .. N_STUDENTS }:
35      spawn student()
```

The program maintains two variables per booth: a *lock* and a boolean that indicates whether the booth is *free*. The program finds that all booths are free after all student threads are finished. In order to understand the program better, you are tasked with determining if certain properties always hold (i.e., invariant), never hold, or neither (i.e., sometimes true, sometimes false).

(b) Answer the following questions for the case N_BOOTHS = 2 and N_STUDENTS = 3. Put one checkmark in each row. (Carefully notice the indices to the *booths* variable below.)

Property	**Always**	**Never**	**Sometimes**
In Line 14, *booths*[*b*].*free* == True			
In Line 16, *booths*[**0**].*free* == False			
In Line 21, *booths*[*b*].*lock* is acquired (held)			

4. Race to the Finish

(Rookie Road, Dec 12 2023)

In a “Relay Race”, runners on a team take turns running legs around a field. One arbitrary runner of each team starts, carrying a baton. When the runner completes the leg, they pass the baton to another (arbitrary) runner in the team. This continues until all runners on the team ran a leg. The teams are ranked by the order in which the last runners cross the finish line.

The code to the right models a relay race with N_TEAMS teams and N_LEGS runners per team. There are the following variables:

- *batons*: a lock for each team modeling a baton
- *legs:* a counter per team keeping track of how many runners have been running
- *finish:* a lock modeling the finish line
- *rank:* a list of teams in the order in which they cross the finish line
- *total*: keeps track of the total number of runners that have been running.

```
1   from synch import *
2
3   sequential legs, rank, total
4
5   const N_TEAMS = 6
6   const N_LEGS = 5
7
8   batons = [ Lock(), ] * N_TEAMS # lock per team
9   legs = [ 0, ] * N_TEAMS # counter per team
10  finish = Lock() # lock for finish line
11  rank = [] # ranking of teams
12  total = 0 # total number of legs run
13
14  def runner(team):
15      acquire(?batons[team])
16      total = total + 1
17      legs[team] = legs[team] + 1
18      var last = legs[team] >= N_LEGS
19      release(?batons[team])
20      if last:
21          acquire(?finish)
22          rank += [team,]
23          release(?finish)
24
25  def main():
26      for team in { 0 .. N_TEAMS – 1 }
27      for _ in { 1 .. N_LEGS }:
28          spawn runner(team)
29      await len(rank) == N_TEAMS
30      print(total)
31
32  spawn main()
```

Line 3 states that both LOAD operations and STORE operations on variables *legs*, *rank*, and *total* are atomic. (This does not imply that other operations such as increment are atomic.)

The **main** method is a thread that starts all the runners for all the teams. The teams are numbered 0 through N_TEAMS – 1. The method waits for all teams to finish in line 29, and then prints the total number of runners that have run the relay race.
The **runner** method is a thread that takes the team identifier as argument. For each team, N_LEGS of this method are spawned, for a total of N_TEAMS * N_LEGS threads. The thread first waits to acquire its team’s baton. Then it increments both *total* and its team’s *legs* counter. The method stores whether this was the last leg for the team in local boolean variable *last*. After releasing the baton, the runner tries to cross the finish line if it is the last of their team. In Line 22, the method adds the team number to the end of variable *rank*.

We wish we could tell you that this code is correct and always prints a number that equals N_TEAMS * N_LEGS. Unfortunately, this turns out not to be the case. The program turns out to be able to print various numbers. Let’s see if we can find out why.

Answer the questions on the following page.

a) First, we want to see if some properties are *invariant* (always true) or not. The table below has a list of properties. Put a checkmark (✓) in each row depending on whether the property always holds, never holds, or neither (i.e., it holds sometimes but not always). Each row should have exactly one checkmark.

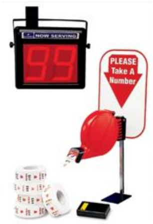
Q	Property	Always	Never	Sometimes
1	Between Lines 16 and 17, *legs*[*team*] < N_LEGS			
2	Between Lines 17 and 18, *legs*[*team*] < N_LEGS			
3	Between Lines 15 and 19, multiple (more than one) runners of the same team hold the baton (lock) of the team			
4	Between Lines 15 and 19, multiple runners of different teams hold a baton (lock)			
5	List *rank* contains duplicates			
6	Between Lines 29 and 30, the length of list *rank* is N_TEAMS			

b) The minimum and maximum value that the program prints appears to depend on N_TEAMS and N_LEGS. Fill in the following table with the minimum and maximum values that can be printed (the first row, in which there is only 1 team with 1 runner, is filled out as an example):

Q	N_TEAMS	N_LEGS	MINIMUM	MAXIMUM
1	1	1	1	1
2	6	1		
3	1	5		
4	2	3		
5	3	2		

c) Is it necessary to declare the variable *legs* as sequentially consistent to prevent a data race? Answer YES or NO in the box to the right.

5. European Bakery Lock



Locks are used to implement critical sections in which only the thread holding the lock can execute (mutual exclusion). Sometimes it is useful to have some "fairness" in that threads should enter the critical section in their arrival order. European bakeries have figured out how to do this. When a customer enters the bakery, they must take a numbered ticket at the door, and then they must wait until their number is on the big display.

The code to the right demonstrates various variations of this idea. A lock is implemented as a record that contains two (arbitrary precision) numbers: a *counter* (displayed to all in the bakery), and a *dispenser* (the ticket number given to the next customer). Initially, both are 0 (Line 13). A customer acquires the lock by calling one of the 3 acquire methods and releases the lock by calling one of the 3 release methods. The acquire method fetches a ticket by loading *dispenser* and incrementing it. The method then loops until the ticket number is equal to *counter*. The release method increments *counter*.

```
1   def fetch_and_increment(p) returns result:
2       atomically:
3           result = !p
4           !p += 1
5
6   def atomic_load(p) returns result:
7       atomically result = !p
8
9   def atomic_store(p, v):
10      atomically !p = v
11
12  def Lock():
13      result = { .counter: 0, .dispenser: 0 }
14
15  def acquire1(lk):
16      let my_ticket = fetch_and_increment(?lk→dispenser):
17          while atomic_load(?lk→counter) != my_ticket:
18              pass
19
20  def acquire2(lk):
21      let my_ticket = atomic_load(?lk→dispenser):
22          atomic_store(?lk→dispenser, my_ticket + 1)
23          while atomic_load(?lk→counter) != my_ticket:
24              pass
25
26  def acquire3(lk):
27      let my_ticket = fetch_and_increment(?lk→dispenser):
28          while lk→counter != my_ticket:
29              pass
30
31  def release1(lk):
32      lk→counter += 1
33
34  def release2(lk):
35      let v = lk→counter:
36          atomic_store(?lk→counter, v + 1)
37
38  def release3(lk):
39      let v = atomic_load(?lk→counter):
40          atomic_store(?lk→counter, v + 1)
```

This must be done while avoiding *data races* as data races lead to undefined behavior.

A data race happens when two or more threads simultaneously access a variable (i.e., one of the counters), and at least one of those accesses writes to the variable. Data races can be avoided using *atomic operations*---by definition, multiple atomic operations cannot overlap. For example, an atomic LOAD and an atomic STORE cannot overlap and thus cannot cause a data race. However, an atomic LOAD and a normal STORE *can* overlap and cause a data race. The code provides three types of atomic operations: **atomic_store**, **atomic_load**, and **fetch_and_increment**. **atomic_store**(*p*, *v*) takes the address of a variable *p* and a value *v* and atomically stores the value in the variable. **atomic_load**(*p*) atomically loads the value of the variable pointed to by *p* and returns it. **fetch_and_increment**(*p*), in a single atomic operation, increments the value stored at the variable pointed at by *p* and returns the old value stored there.

The threads consistently use one of the 3 acquire methods to enter a critical section and one of the 3 release methods to leave a critical section (always with the same lock), and so there are 9 possible combinations. In the "data race free" column, specify using a checkmark (✓) if the combination is free of data races, and an X (✗) if the combination suffers from data races (and thus the behavior is undefined). If you use an X, leave the other box blank (as it would be undefined). If, however, you think the combination is free of data races, mark with a checkmark if you think the combination is correct (implements *mutual exclusion* and *progress*). If not, enter an X. Missing and ambiguous marks will be counted as wrong. Thus, a correct acquire/release combination should contain two checkmarks.

Q	**acquire method**	**release method**	**data race free**	**correct**
1	acquire1	release1		
2	acquire1	release2		
3	acquire1	release3		
4	acquire2	release1		
5	acquire2	release2		
6	acquire2	release3		
7	acquire3	release1		
8	acquire3	release2		
9	acquire3	release3		

6. To Lock or Not To Lock

Consider the following Harmony program:

```
from synch import Lock, acquire, release

const L = True;    # whether to lock or not
const U = True;    # whether to unlock or not

mutex = Lock()
sequential accu
accu = 3

def cs_enter():    # enter critical section
    if L: acquire(?mutex)

def cs_exit():     # leave critical section
    if U: release(?mutex)

def T0():
    cs_enter(); accu += 2; cs_exit()

def T1():
    cs_enter(); accu *= 3; cs_exit()

spawn T0()
spawn T1()
```

Note that *accu* has sequential consistency, which means that load and store operations on it happen atomically. Now answer the following questions (there is no partial credit for partial answers):

(a)	After both threads T0 and T1 have terminated, what are the possible values of *accu*? Enter one or more integer numbers or enter 'none' if you think one or both threads may never terminate	
(b)	Now suppose L and U are set to False. After both threads T0 and T1 have terminated, what are the possible values of *accu*?	
(c)	Now suppose L is set to True and U is set to False. After both threads T0 and T1 have terminated, what are the possible values of *accu*?	

7. Ithaca is One Lane Bridges

The Ithaca campus is home to various one-lane bridges. Cars can only go in one direction (represented by 0 or 1) on such a bridge. The following program simulates 3 cars crossing in direction 0 and 3 cars crossing in direction 1. Each car is represented by a thread. To enter the bridge in direction d (0 or 1), the car must call **bridge_enter**(*d*), and to exit it must call **bridge_exit**(*d*). MAX_CARS is the maximum number of cars allowed on the bridge at any time. *ncars*[*d*] represents the number of cars on the bridge going in direction *d*. The following invariants must hold for any number of cars trying to cross the bridge:

- $ncars[0] = 0 \lor ncars[1] = 0$
- $0 \le ncars[0] + ncars[1] \le \text{MAX\_CARS}$

Also, if a car *can* enter the bridge, it should be allowed to. (The code cannot simply allow just one car at a time on the bridge.)

```
from synch import *

const MAX_CARS = 2

mutex = Lock()
condition = [ Condition(), Condition() ]
ncars = [ 0, 0 ]

def bridge_enter(direction):
    acquire(?mutex)
    while (ncars[1 − direction] > 0) or (ncars[direction] == MAX_CARS):
        wait(?condition[direction], ?mutex)
    ncars[direction] += 1
    release(?mutex)

def bridge_exit(direction):
    acquire(?mutex)
    notifyAll(?condition[direction])
    ncars[direction] −= 1
    if ncars[direction] == 0:
        notifyAll(?condition[1 − direction])
    release(?mutex)

def car(direction):
    bridge_enter(direction)
    @onbridge: pass
    bridge_exit(direction)

for i in {1..3}:
    spawn car(0)
for i in {1..3}:
    spawn car(1)
```

In the following questions, “at line X” means “just before executing the statement at line X”. Answer the following questions with ‘yes’, ‘no’, or ‘maybe’. For example, in the first question, answer ‘yes’ if the statement is an invariant, ‘no’ if the negation of the statement is an invariant, or ‘maybe’ if neither is the case. Assume that MAX CARS is 2 for all questions.

		yes	no	maybe
(a)	At line 11, $ncars[direction] < \mathrm{MAX\_CARS}$			
(b)	At line 13, $ncars[direction] = 0$			
(c)	At line 13, $ncars[1 - direction] = 0$			
(d)	At line 13, $ncars[direction] < \mathrm{MAX\_CARS}$			
(e)	At line 18, $ncars[1 - direction] = 0$			
(f)	In line 18, it would be correct to replace **notifyAll** by **notify**			
(g)	In line 21, it would be correct to replace **notifyAll** by **notify**			

8. Off to the races

Below find 8 Harmony programs with two threads that either read or write the shared variable *x*. The programs use one or two reader/writer locks. For each of the programs, indicate in the table whether the program may suffer a data race and/or a deadlock.

V1

```
import RW

rw = RW.RWlock()
x = 0

def f(self):
    RW.read_acquire(?rw)
    x = self        // write x
    RW.read_release(?rw)

spawn f(0)
spawn f(1)
```

V2

```
import RW

rw = RW.RWlock()
x = 0

def f(self):
    RW.write_acquire(?rw)
    x = self        // write x
    RW.write_release(?rw)

spawn f(0)
spawn f(1)
```

V3

```
import RW

rw = RW.RWlock()
x = 0

def f(self):
    RW.read_acquire(?rw)
    result = x      // read x
    RW.read_release(?rw)

spawn f(0)
spawn f(1)
```

V4

```
import RW

rw = [ RW.RWlock(), RW.RWlock() ]
x = 0

def f(self):
    RW.read_acquire(?rw[self])
    RW.read_acquire(?rw[1 − self])
    x = self        // write x
    RW.read_release(?rw[1 − self])
    RW.read_release(?rw[self])

spawn f(0)
spawn f(1)
```

V5

```
import RW

rw = [ RW.RWlock(), RW.RWlock() ]
x = 0

def f(self):
    RW.write_acquire(?rw[self])
    RW.write_acquire(?rw[1 − self])
    x = self        // write x
    RW.write_release(?rw[1 − self])
    RW.write_release(?rw[self])

spawn f(0)
spawn f(1)
```

```
import RW

rw = [ RW.RWlock(), RW.RWlock() ]
x = 0

def f(self):
    RW.write_acquire(?rw[0])
    RW.write_acquire(?rw[1])
    x = self          // write x
    RW.write_release(?rw[1])
    RW.write_release(?rw[0])

spawn f(0)
spawn f(1)
```

V6

```
import RW

rw = [ RW.RWlock(), RW.RWlock() ]
x = 0

def f(self):
    RW.write_acquire(?rw[self])
    RW.read_acquire(?rw[1 − self])
    x = self          // write x
    RW.read_release(?rw[1 − self])
    RW.write_release(?rw[self])

spawn f(0)
spawn f(1)
```

V7

```
import RW

rw = [ RW.RWlock(), RW.RWlock() ]
x = 0

def f(self):
    RW.write_acquire(?rw[0])
    RW.read_acquire(?rw[1])
    x = self          // write x
    RW.write_release(?rw[0])
    RW.read_release(?rw[1])

spawn f(0)
spawn f(1)
```

V8

Fill in the following table with "Y" (Yes) or "N" (No):

	V1	V2	V3	V4	V5	V6	V7	V8
Has a data race								
May deadlock								

(there should be a Y or N in each of the 16 boxes above)

9. What a ride it has been!

Given is a rollercoaster with a single car. Safety precautions require the following:

- each ride requires that all seats on the car are filled. That is, partially filled cars are not allowed to ride
- after a ride, the car must completely empty out before new riders are allowed to enter the car

Below find an implementation of a rollercoaster. RollerCoaster(*N*) returns the initial state of a rollercoaster with *N* seats to a car. enter(*b*) takes a pointer *b* to a rollercoaster variable. A thread that calls enter(*b*) should block until 1) all previous riders have left the car and 2) the car has filled up again. After "doing the ride", the thread calls exit(*b*).

```
from synch import *

def RollerCoaster(nseats): result = {
    .mutex: Lock(), .nseats: nseats, .entered: 0, .left: nseats,
    .empty: Condition(), .full: Condition()
}

def enter(b):
    acquire(?b→mutex)
    while b→entered == b→nseats: # wait for car to empty out
        wait(?b→empty, ?b→mutex)
    b→entered += 1
    if b→entered != b→nseats: # wait for car to fill up
        while b→entered < b→nseats:
            wait(?b→full, ?b→mutex)
    else: # car is ready to go
        b→left = 0
        notifyAll(?b→full) # wake up others waiting in car
    release(?b→mutex)

def exit(b):
    acquire(?b→mutex)
    b→left += 1
    if b→left == b→nseats: # car is empty
        b→entered = 0
        notifyAll(?b→empty) # wake up riders wanting to go
    release(?b→mutex)
```

To the right is a simple test program for the rollercoaster. It create a rollercoaster variable with 3 seats in the car. It then creates 3x4 = 12 rider threads that each try to get a ride. Each rider only tries to ride the rollercoaster once.

```
import rollercoaster

const N = 3 # seats in car
const T = 4 # number of rides

rc = rollercoaster.RollerCoaster(N)

def rider():
    rollercoaster.enter(?rc)
    ride: assert 1 <= countLabel(ride) <= N
    rollercoaster.exit(?rc)

for i in {1..N*T}: spawn rider()
```

Fill in the following tables.

Place one checkmark **in each row** in *one* of the three columns *("neither" means sometimes True and sometimes False)*		always True	always False	neither
a)	if a thread is executing at line 38 (at the "ride" label), then there are at most N threads executing at line 38			
b)	if a thread is executing at line 38 (at the "ride" label), then *rc.entered* = N			
c)	if a thread is executing at line 38 (at the "ride" label), then *rc.left* = 0			
d)	after all threads are done (have terminated), *rc.entered* = 0			
e)	after all threads are done (have terminated), *rc.left* = N			
f)	$(0 < rc.left < \mathrm{N}) \Rightarrow (rc.entered = \mathrm{N})$			
g)	$rc.entered + rc.left > 0$			
h)	$rc.entered + rc.left = \mathrm{N}$			
i)	$rc.entered + rc.left < 2\mathrm{N}$			
j)	all threads eventually terminate			
k)	if Line 41 is changed to spawn 10 threads exactly, then all threads eventually terminate			

Place one checkmark **in each row** in *one* of the two columns		True	False
l)	it's ok to replace notifyAll() in Line 18 by notify()		
m)	it's ok to replace notifyAll() in Line 26 by notify()		

10. Playing a Waiting Game

Professor W. W. Walker at Worwell University runs a game design program. In it they form teams of 3 students from across campus. Each team has 1 illustrator, 1 musician, and 1 hacker (no disrespect intended). As students sign up, Professor WWW creates teams as soon as possible. There is a maximum number of teams in the program. As the program is incredibly popular, Professor WWW never has to worry about not being able to form teams. Below find a Harmony model of team forming:

```
from synch import *

const NTEAMS = 3

mutex = Lock()
cond = Condition()
ready = { .illustrator: [], .musician: [] }
matched = {}
```

the global variables consist of a lock, a Mesa condition variable, a FIFO list each for illustrators and musicians, and a set of matched teams

```
def member(role, id):
    acquire(?mutex)
    if role == .hacker:
        while (ready.illustrator == []) or (ready.musician == []):
            wait(?cond, ?mutex)
```

a hacker waits for at least one illustrator and one musician to line up

```
        let match = {
                .hacker: id,
                .musician: ready.musician[0],
                .illustrator: ready.illustrator[0] }:
            del ready.illustrator[0] # remove the first illustrator
            del ready.musician[0] # remove the first musician
            matched |= { match }
```

match consists of the hacker along with the first musician & illustrator on the ready lists.
('|' is the union operator.)

```
    else:
        ready[role] += [ id, ]
        if (ready.illustrator != []) and (ready.musician != []):
            notifyAll(?cond)
    release(?mutex)
```

illustrators and musicians line up and, if both lines are non-empty, notify all hackers

```
for i in {1..NTEAMS}:
    spawn member(.illustrator, i)
    spawn member(.musician, i)
    spawn member(.hacker, i)
```

spawn all illustrators, musicians, and hackers

```
# Check that each student is in exactly one match at the end
finally all(
    (len { m for m in matched where m[role] == id }) == 1
        for role in { .illustrator, .musician, .hacker }
        for id in {1..NTEAMS}
)
```

check that each illustrator, musician, and hacker is in exactly one team in the end

Note that there is an asymmetry in the Harmony program: hackers wait for musicians and illustrators, while musicians and illustrators notify hackers. Answer the following questions about this Harmony program.

		True	False
a)	The program is free of data races		
b)	All threads are guaranteed to terminate eventually		
c)	If we replace notifyAll by notify in Line 25, some threads may never terminate		
d)	If we replace the if statement in Line 24 by “if True”, all threads are guaranteed to terminate eventually		
e)	Right before Line 15, it is guaranteed that there is at least one musician and one illustrator on the respective ready lists		
f)	Right before Line 21, it is guaranteed that both ready lists will be empty		
g)	In Line 13, it’s ok to replace “while” by “if” assuming that there are no “spurious wakeups” (i.e., wait() only returns if notified)		
h)	It is ok (or even better) to replace “or” by “and” in the while condition in Line 13.		
i)	If we didn’t care about busy waiting, it would be ok to replace the wait() statement in Line 14 by “release(?*mutex*); acquire(?*mutex*)”		
j)	If the program spawned fewer musicians than hackers, then some threads would never terminate		
k)	The “del” statements that delete the first musician and illustrator from the respective ready list are there just to reduce memory usage and can be safely removed		

11. RVR Really Needs a Haircut

RVR occasionally (not often enough) gets his hair cut at Big Red Barber Shop in Collegetown. A typically barbershop has a number of barbers (and the same number of barber chairs) and a waiting area with some seats. Customers periodically have their hair cut until, for some reason, they no longer need to get their hair cut. However, if they get to the barbershop and all waiting seats are taken, they skip the cut. Below find a model of a barbershop with NBARBERS barbers, NSEATS waiting seats, and NCUSTOMERS customers using a lock and two Mesa condition variables.

```
from synch import *

const NSEATS = 2
const NCUSTOMERS = 3
const NBARBERS = 2

mutex = Lock()
customer_cond = Condition()
barber_cond = Condition()
customers_waiting = {}
customers_ready = {}

def barber(self):
    while True:
        # Wait for a customer
        acquire(?mutex)
        while customers_waiting == {}:
            wait(?barber_cond, ?mutex)
        var c = choose customers_waiting
        customers_waiting -= { c }
        release(?mutex)

        cut: pass # Cut hair of customer c

        # Cut is done. Tell customer to go.
        acquire(?mutex)
        customers_ready |= { c }
        notifyAll(?customer_cond)
        release(?mutex)
```

```
def customer(id):
    while choose { False, True }: # while alive
        acquire(?mutex)
        if len customers_waiting < NSEATS:
            # Take seat and wake up a barber
            customers_waiting |= { id }
            notify(?barber_cond)

            # Wait for some barber to cut my hair
            while id not in customers_ready:
                wait(?customer_cond, ?mutex)
            customers_ready -= { id }
        release(?mutex)

for _ in {1..NBARBERS}:
    spawn eternal barber()
for i in {1..NCUSTOMERS}:
    spawn customer(i)
```

Each customer has a unique identifier. The model maintains two sets of customer identifiers: *customers_waiting,* the set of customers waiting for their hair to be cut, and *customers_ready,* the set of customers who just had their hair cut but haven't left the shop yet.

A customer checks to see if there are available waiting seats and, if so, takes one of the waiting seats. ('|' is the union operator.). The customer also notifies a barber. The customer then waits until their hair is cut.

A barber, in an eternal loop, waits for a customer and cuts their hair. Cutting hair takes some time. In the code above, a barber waits until there is at least one customer waiting, then chooses one of the customers non-deterministically, cuts their hair, and adds the customer to the *customer_ready* set. The barber finally notifies all customers.

Answer the following questions:

		True	False
a)	The program is free of data races		
b)	Deadlock is possible		
c)	To prevent deadlock, it is important that a barber releases the lock (in Line 21) before cutting hair and re-acquire the lock (in Line 25) afterwards		
d)	It is possible for $customer\_waiting \cap customer\_ready$ to be non-empty, i.e., some customer identifier may be in both sets		
e)	Right after acquiring the lock in Line 33, it is guaranteed that *id* is neither in $customer\_waiting$ nor in $customer\_ready$		
f)	Right after releasing the lock in Line 43, it is guaranteed that *id* is neither in $customer\_waiting$ nor in $customer\_ready$		
g)	Right after notifying a barber in Line 37, it may be that the barber runs next and adds *id* to $customer\_ready$ so the **wait** call in Line 41 is not executed.		
h)	Right before executing Line 19, $customer\_ready$ is guaranteed to be empty		
i)	Right before executing Line 19, $customer\_ready$ is guaranteed to be non-empty		
j)	Right before executing Line 26, $customer\_ready$ is guaranteed to be empty		
k)	It is ok to replace **notifyAll** by **notify** in Line 28.		

The code to the right is how one might test the ClubHouse code in Question q.3. You do not need to look at this code unless you think it might help you.

The code models a clubhouse that can accommodate at most 2 people at a time, and three clubs of 3, 3, and 1 members respectively.

The code checks that the capacity of the clubhouse is not exceeded and that there cannot be members of multiple clubs in the clubhouse simultaneously.

```
from club import ClubHouse, enter, exit

const CAPACITY = 2 # max number of people in club house
const CLUBS = [ 3, 3, 1 ] # number of members in each club
theclub = ClubHouse(CAPACITY)
counters = [0,] * len(CLUBS)

def member(myclub):
    while choose { True, False }:
        enter(?theclub, myclub)
        atomically:
            counters[myclub] += 1
            assert 1 <= counters[myclub] <= CAPACITY
            assert all(counters[c] == 0
                for c in { 0 .. len(CLUBS) - 1} where c != myclub)
        # linger a while
        atomically counters[myclub] -= 1
        exit(?theclub)

for c in { 0 .. len(CLUBS) - 1 }
for _ in { 1 .. CLUBS[c] }:
    spawn member(c)
```

12. Join the Club

Cornell University has a clubhouse that is shared by several clubs. Fire codes require that the clubhouse cannot be occupied by more than a certain maximum number of students. To enter the clubhouse, a student must be a member of a club. There are no students that are members of more than one club. Moreover, the clubs agreed that the clubhouse cannot be used by more than one club at a time. This means that there can never be two students of different clubs in the clubhouse. Every student in the clubhouse eventually exits.

On this and the following page you find four implementations (A, B, C, and D) of a "ClubHouse" abstraction that try to model such a clubhouse. Students are modeled as threads. The interface has three methods:

ClubHouse(*n*): returns the initial state of a clubhouse with a maximum occupancy of n ($n > 1$) students.

enter(*ch, club*): *ch* points to a clubhouse variable, and *club* is a unique identifier for some club. This method models a member of the given club trying to enter the clubhouse. The method waits until the conditions above are met but must not stop a student if they can enter.

exit(ch): this models a student leaving the clubhouse.

A

```
from synch import *

def ClubHouse(n) returns init:
    init = {
        .mutex: Lock(),
        .capacity: n, .occupancy: 0, .club: None
    }

def enter(ch, club):
    acquire(?ch→mutex)
    while (ch→occupancy == ch→capacity) or
            ((ch→occupancy > 0) and (ch→club != club)):
        release(?ch→mutex)
        acquire(?ch→mutex)
    ch→club = club
    ch→occupancy += 1
    release(?ch→mutex)

def exit(ch):
    acquire(?ch→mutex)
    ch→occupancy −= 1
    release(?ch→mutex)
```

Each of the implementations keeps track of the state of the clubhouse. The implementations have the following state in common:

> *mutex*: a lock that protects the state under concurrent access
> *capacity*: the maximum number of students allowed in the clubhouse
> *occupancy*: the current number of students in the clubhouse
> *club*: the club the students belong to (if there are any students in the clubhouse)

a) Put ✓ or ✗ in each of the following boxes (one box per property and implementation):

Property	**A**	**B**	**C**	**D**
Does *not* allow students of different clubs to enter at the same time				
Does *not* allow more students to enter than the capacity allows				
Does *not* prevent students from entering if they are allowed into the clubhouse				
Does *not* use *busy waiting*				

(Waiting on a lock or on a condition variable is not considered busy waiting.)

B

```
1   from synch import *
2
3   def ClubHouse(n) returns init:
4       init = {
5           .mutex: Lock(),
6           .conflict: Condition(), .full: Condition(),
7           .capacity: n, .occupancy: 0, .club: None
8       }
9
10  def enter(ch, club):
11      acquire(?ch→mutex)
12      while (ch→occupancy == ch→capacity) or
13              ((ch→occupancy > 0) and (ch→club != club)):
14          if ch→occupancy == ch→capacity:
15              wait(?ch→full, ?ch→mutex)
16          if (ch→occupancy > 0) and (ch→club != club):
17              wait(?ch→conflict, ?ch→mutex)
18      ch→club = club
19      ch→occupancy += 1
20      release(?ch→mutex)
21
22  def exit(ch):
23      acquire(?ch→mutex)
24      ch→occupancy −= 1
25      if ch→occupancy == 0:
26          notifyAll(?ch→conflict)
27      notifyAll(?ch→full)
28      release(?ch→mutex)
```

b) Answer the following questions using ✓ or ✗ only for implementation B.

Property	**B**
It's ok to remove the **if** statement in line 25 and always **notifyAll** *ch->conflict*	
It's ok to replace **notifyAll** in line 26 with **notify**	
It's better to replace **notifyAll** in line 27 with **notify**	

C

```
from synch import *

def ClubHouse(n) returns init:
    init = {
        .mutex: Lock(),
        .conflict: Condition(), .full: Condition(),
        .capacity: n, .occupancy: 0, .club: None
    }

def enter(ch, club):
    acquire(?ch→mutex)
    while ch→occupancy == ch→capacity:
        wait(?ch→full, ?ch→mutex)
    while (ch→occupancy > 0) and (ch→club != club):
        wait(?ch→conflict, ?ch→mutex)
    ch→club = club
    ch→occupancy += 1
    release(?ch→mutex)

def exit(ch):
    acquire(?ch→mutex)
    ch→occupancy −= 1
    if ch→occupancy == 0:
        notifyAll(?ch→conflict)
    notifyAll(?ch→full)
    release(?ch→mutex)
```

D

```
from synch import *

def ClubHouse(n) returns init:
    init = Lock()

def enter(ch, club):
    acquire(ch)

def exit(ch):
    release(ch)
```

Did you fill out the table in the top right?

13. Running out of Steam

Below, on the left, find a Harmony program that models a "bounded resource." A bounded resource has a certain number of instances, called its "capacity." A resource is modeled by a "resource variable" initialized using Resource(*capacity*). Given a pointer *r* to a resource variable, the method alloc(*r, n*) allocates *n* instances of the resource. The method waits until *n* instances are available. The method free(*r, n*) releases *n* previously allocated instances of the resource.

```
from synch import *

def Resource(capacity) returns init:
    init = { .mutex: Lock(), .cond: Condition(),
             .cap: capacity, .avail: capacity }

def alloc(r, n):
    acquire(?r→mutex)
    assert 0 <= r→avail <= r→cap
    while n > r→avail:
        wait(?r→cond, ?r→mutex)
    r→avail -= n
    release(?r→mutex)

def free(r, n):
    acquire(?r→mutex)
    assert 0 <= r→avail <= (r→avail + n) <= r→cap
    r→avail += n
    notifyAll(?r→cond)
    release(?r→mutex)
```

```
rx = Resource(N_X)
ry = Resource(N_Y)

def app1():
    alloc(?rx, N_X_1)
    alloc(?ry, N_Y_1)
    free(?rx, N_X_1)
    free(?ry, N_Y_1)

def app2():
    alloc(?ry, N_Y_2)
    alloc(?rx, N_X_2)
    free(?ry, N_Y_2)
    free(?rx, N_X_2)

spawn app1()
spawn app2()
```

The code on the right models two resources *x* and *y* by resource variables *rx* and *ry*, as well as two threads app1() and app2() that allocate resources. There are N_X instances of resource *x* and N_Y instances of resource *y*. app1() first allocates N_X_1 instances of resource *x* and then N_Y_1 instances of resource *y*. app2(), instead, allocates N_Y_2 instances of resource *y* first and then N_X_2 instances of resource *x*. Both threads immediately release the resources they acquired.

Unfortunately, for certain values of these 6 constants, deadlock can occur.

q.1) Fill in the following table using ✓ or ✗ in each box:

N_X	N_Y	N_X_1	N_Y_1	N_X_2	N_Y_2	deadlock possible?
1	1	1	1	1	1	
2	2	1	1	1	1	
2	2	2	2	2	2	
2	2	1	2	1	2	
2	2	1	2	2	1	

14. The Dining Philosophers Return

The Dining Philosophers are dining again. Below find an (incomplete) program for the Dining Philosophers. The state of each fork is represented by a boolean. False means the fork is available; True means the fork is taken. The code uses a Mesa monitor approach.

```
from synch import *

const N = 5

mutex = Lock()
forks = [False,] * N
conds = [Condition(),] * N

def pickup_fork(f):
    assert not forks[f]
    forks[f] = True

def replace_fork(f):
    assert forks[f]
    forks[f] = False
    notify(?conds[f]) # wake up someone waiting for this fork

def get_forks(left, right):
    pass # replace with code to pick up forks

def diner(which):
    let left, right = (which, (which + 1) % N):
        while choose({ False, True }):
            acquire(?mutex)
            get_forks(left, right)
            release(?mutex)
            #
            # dine
            #
            acquire(?mutex)
            replace_fork(left)
            replace_fork(right)
            release(?mutex)

for i in {0..N−1}:
    spawn diner(i)
```

The state consists of a monitor lock, the status of each fork, and a condition variable for each fork.

A fork should only be picked up when it is not already taken.

A fork can only be replaced if it's taken. After doing so, wake up a thread waiting for the fork, if any.

to be replaced

Each diner is a thread that first determines the identifiers of its left and right forks. Then it runs a loop until it chooses False. In the loop body, the thread first picks up the forks, then dines, then replaces the forks.

The method get_forks(left, right) must be completed. Below find five different ways of implementing get_forks(). Some are correct; some are not. You are to determine if one of the assertions on the previous page might fail and if the implementation might deadlock if the assertions are ignored.

V1

```
def get_forks(left, right):
    while forks[left]:
        wait(?conds[left], ?mutex)
    pickup_fork(left)
    while forks[right]:
        wait(?conds[right], ?mutex)
    pickup_fork(right)
```

V2

```
def get_forks(left, right):
    while forks[left] or forks[right]:
        if forks[left]:
            wait(?conds[left], ?mutex)
        if forks[right]:
            wait(?conds[right], ?mutex)
    pickup_fork(left)
    pickup_fork(right)
```

V3

```
def get_forks(left, right):
    let f1, f2 = min(left, right), max(left, right):
        while forks[f1]:
            wait(?conds[f1], ?mutex)
        pickup_fork(f1)
        while forks[f2]:
            wait(?conds[f2], ?mutex)
        pickup_fork(f2)
```

V4

```
def get_forks(left, right):
    while forks[left]:
        wait(?conds[left], ?mutex)
    while forks[right]:
        wait(?conds[right], ?mutex)
    pickup_fork(left)
    pickup_fork(right)
```

V5

```
def get_forks(left, right):
    let f1, f2 = min(left, right), max(left, right):
        while forks[f1]:
            wait(?conds[f1], ?mutex)
        while forks[f2]:
            wait(?conds[f2], ?mutex)
        pickup_fork(f1)
        pickup_fork(f2)
```

Fill in the following table with "Y" (Yes) or "N" (No):

	V1	V2	V3	V4	V5
Assertion may fail					
May deadlock					

(there should be a Y or N in each of the 10 boxes above)

15. Hit the gas!

The following Harmony program uses a Mesa monitor approach to simulate carbon and hydrogen atoms (each modeled using threads) combining to form methane (CH_4) molecules. Each methane molecule consists of 4 hydrogen atoms and 1 carbon atom.

```
from synch import *

const nCarbon = 1000
const nHydrogen = 4 * nCarbon

waitingHydrogen = waitingCarbon = 0
matchedHydrogen = matchedCarbon = 0
mutex = Lock()
condHydrogen = condCarbon = Condition()

def hydrogen():
    acquire(?mutex)
    if (waitingHydrogen >= 3) and (waitingCarbon >= 1):
        waitingHydrogen -= 3; matchedHydrogen += 3
        waitingCarbon -= 1; matchedCarbon += 1
        notifyAll(?condHydrogen)
        notify(?condCarbon)
    else:
        waitingHydrogen += 1
        while matchedHydrogen == 0:
            wait(?condHydrogen, ?mutex)
        matchedHydrogen -= 1
    release(?mutex)

def carbon():
    acquire(?mutex)
    if waitingHydrogen >= 4:
        waitingHydrogen -= 4; matchedHydrogen += 4
        notifyAll(?condHydrogen)
    else:
        waitingCarbon += 1
        while matchedCarbon == 0:
            wait(?condCarbon, ?mutex)
        matchedCarbon -= 1
    release(?mutex)

for _ in {1..nHydrogen}: spawn hydrogen()
for _ in {1..nCarbon}: spawn carbon()
```

The state consists of the number of hydrogen and carbon threads waiting to be combined, and the remaining number of such threads that have been matched to form methane.

When there’s a match, notify all who might be able to continue.

Wait until a hydrogen atom has been made part of a methane molecule.

When there’s a match, notify hydrogen threads only.

Wait until a carbon atom has been made part of a methane molecule.

Fill in the following tables.

Place one checkmark **in each row** in *one* of the three columns *("neither" means sometimes True and sometimes False)*		always True	always False	neither
a)	all threads eventually terminate			
b)	if all threads have terminated, then $waitingHydrogen$ and $waitingCarbon$ are both 0			
c)	if all threads have terminated, then $matchedHydrogen$ and $matchedCarbon$ are both larger than 0			
d)	If no lock is held, then ($waitingHydrogen < 4$ or $waitingCarbon = 0$)			
e)	If no lock is held, then ($matchedHydrogen < 4$ or $matchedCarbon = 0$)			
f)	when a thread resumes right after the wait() call in line 21, $matchedHydrogen$ is larger than 0			
g)	if, in line 4, constant *nHydrogen* is set to a value larger than 4 times *nCarbon*, then some hydrogen threads will never terminate			
h)	if no lock is held, then $4 \times matchedHydrogen = matchedCarbon$			

Place one checkmark **in each row** in *one* of the two columns		True	False
i)	it's ok to replace notifyAll() in Line 16 by notify()		
j)	it's ok to replace notifyAll() in Line 16 by 3 calls to notify() on the same condition variable		
k)	it's ok to replace notifyAll() in Line 29 by notify()		
l)	it's ok to replace notifyAll() in Line 29 by 3 calls to notify() on the same condition variable		

16. Chez Platopus

The famous philosopher Platopus lives in the underwater world of the multipi. A multipus is an elegant creature with one or more arms. E.g., an octopus has eight arms, while a pentopus has only 5. Multipi love to eat escargot (snails), but they require a fork for each arm before they can eat. Platopus decides to open an escargot place. There's a single large communal table with a glass of forks in the center. When a multipus arrives at the restaurant, they go to the table, eagerly take forks from the glass until they have a fork for each arm (waiting if the glass is empty), eat all escargot they can eat (and there's an infinite supply of those), and then replace the forks. The Harmony program below models the restaurant.

```
from synch import *

const NFORKS = 18
const MULTIPI = [ 8, 5, 8 ]
```

The parameters. NFORKS is the number of forks initially in the glass. In this scenario there are three multipi: two octopi and one pentopus, and 18 forks in the glass initially.

```
mutex = Lock()
cond = Condition()
avail = NFORKS
```

Here are the variables. The integer *avail* keeps track of how many forks there are left in the glass.

```
def take_fork():
    acquire(?mutex)
    while avail == 0:
        wait(?cond, ?mutex)
    avail −= 1
    release(?mutex)
```

This code waits until there's at least one fork in the glass and then takes it.

```
def replace_fork():
    acquire(?mutex)
    avail += 1
    notifyAll(?cond)
    release(?mutex)
```

This code replaces a fork in the glass and notifies the multipi that are waiting for a fork.

```
def multipus(n):
    for i in {1..n}:
        take_fork()
    # eat
    for i in {1..n}:
        replace_fork()
```

A multipus with *n* arms eagerly takes *n* forks, one at a time, and then eats. After eating, they replace the forks and leave.

```
for n in MULTIPI:
    spawn multipus(n)
```

Here the various multipi get spawned.

Initially, Platopus didn't think things through very carefully. They noticed that sometimes there are a bunch of platipi at the table, each holding some but not enough forks, while the glass is empty. Platopus called this idea "deadlock" and spent the rest of their highly influential life trying to figure out under what circumstances deadlock may occur.

a) Fill in the table below, putting a checkmark in each row in one of the columns:
   Always: deadlock is unavoidable. It always occurs.
   Never: deadlock cannot happen.
   Sometimes: deadlock may or may not occur depending on circumstances

NFORKS	MULTIPI	Always	Never	Sometimes
18	[8, 5, 8]			
15	[8, 8]			
21	[8, 8, 8]			
5	[2, 2, 2, 2]			
2	[3, 1]			

Now answer the following questions about the code:

		True	False
b)	Just after executing Line 11, it is invariant that *avail* = 0		
c)	Just before executing Line 14, it is invariant that *avail* > 0		
d)	Just before executing Line 19, it is invariant that *avail* < NFORKS		
e)	Line 20 can be replaced with **notify**(?*cond*) with no ill effect (i.e., if deadlocks were not possible, they are still not possible)		
f)	Line 26 is a "critical section". That is, it is impossible for multiple multipi to eat at the same time		

17. The Water is Wide

The Harmony code below models a ferry between locations 0 and 1 (representing the east side and the west side of a river respectively). The ferry can carry a maximum of CAPACITY passengers.

```
from synch import *

const CAPACITY = 10

mutex = Lock()
location = 0 # 0 = east, 1 = west
avail = [ Condition(), Condition() ]
full = [ Condition(), Condition() ]
entered = 0
left = CAPACITY

def embark(side):
    acquire(?mutex)
    # wait for ferry to arrive and empty out
    while (location != side) or (left < CAPACITY):
        wait(?avail[side], ?mutex)
    entered += 1
    if entered < CAPACITY:
        # wait for ferry to fill up
        while entered != CAPACITY:
            wait(?full[side], ?mutex)
    else:
        # ferry is full and can cross
        left = 0
        notifyAll(?full[side])
    release(?mutex)

def disembark(side):
    acquire(?mutex)
    left += 1
    if left == CAPACITY:
        # everybody has arrived and left the ferry
        entered = 0
        location = 1 - location
        notifyAll(?avail[location])
    release(?mutex)

```

```
const EAST = 40
const WEST = 30

def passenger(side):
    embark(side)
    # Enjoy the view
    disembark(side)

for i in {1 .. EAST}:
    spawn passenger(0)
for i in {1 .. WEST}:
    spawn passenger(1)
```

The ferry starts out on the east side (0) and can only run when it's filled up with passengers. There are a total of EAST passengers on the east side and WEST passengers on the west side. The passengers are spawned in Lines 49-60 in the box on the right-hand side of the page. Each passenger embarks on either the east or west side and disembarks on the other. For the code to work, both EAST and WEST must be a multiple of CAPACITY, and either EAST = WEST or EAST = WEST + CAPACITY.

The ferry code uses the following variables:

- *mutex*: a global lock
- *location*: the current location of the ferry (0 is east side, 1 is west side)
- *avail*: a condition variable for each side, to wait until ferry is ready for boarding
- *full*: a condition variable for each side, to wait until the ferry filled up and can sail
- *entered*: the number of passengers that have boarded the ferry
- *left*: the number of passengers that have disembarked

Answer the following questions about the code:		True	False
a)	The number of threads executing in Line 54 is always either 0 or CAPACITY		
b)	If some thread is executing at Line 54, then *entered* = CAPACITY		
c)	If some thread is executing at Line 54, then *left* < CAPACITY		
d)	If all threads terminate, then in the end *left* = 0		
e)	The following predicate is invariant after initialization: $entered + left \geq \text{CAPACITY}$		
f)	The **notifyAll** call in Line 25 is inefficient and can be replaced with just **notify**		
g)	The **notifyAll** call in Line 35 is inefficient and can be replaced with just **notify**		
h)	Using the constants as given, this program will eventually terminate (all threads will eventually terminate)		

18. Cross Lock

A "cross lock" is a bit like a reader/writer lock, but it is more symmetric. As in reader/writer locks, there are two kinds of thread. The kinds are 0 and 1. Multiple threads can acquire the lock, but there can never be more than one of each kind that has simultaneously acquired the lock. So, it's ok for one thread of kind 0 and three threads of kind 1 to have the lock simultaneously, but it's not ok for two threads of kind 0 and two threads of kind 1 to have the lock simultaneously. Below find a specification and an implementation.

Specification

```
def CrossLock() returns init:
    init = { .count: [0,] * 2 }

def ok(c, kind) returns success:
    success = (c→count[kind] == 0) or (c→count[1 − kind] in { 0, 1 })

def cl_acquire(c, kind):
    atomically when ok(c, kind):
        c→count[kind] += 1

def cl_release(c, kind):
    atomically c→count[kind] −= 1
```

Implementation

```
from synch import *

def CrossLock() returns init:
    init = {
        .mutex: Lock(),
        .count: [0,] * 2,
        .cond: [Condition(),] * 2
    }

def ok(c, kind) returns success:
    success = (c→count[kind] == 0) or (c→count[1 − kind] in { 0, 1 })

def cl_acquire(c, kind):
    acquire(?c→mutex)
    while not ok(c, kind):
        wait(?c→cond[kind], ?c→mutex)
    c→count[kind] += 1
    release(?c→mutex)

def cl_release(c, kind):
    acquire(?c→mutex)
    c→count[kind] −= 1
    if c→count[kind] == 0:
        notify(?c→cond[kind])
    if c→count[kind] == 1:
        notifyAll(?c→cond[1 − kind])
    release(?c→mutex)
```

Method CrossLock() returns the initial value of a cross lock.

Method cl_acquire() takes a pointer to a cross lock variable and a kind as arguments. It blocks if it is not currently possible to acquire the lock for that kind of thread.

Method cl_release() takes the same two arguments. It releases the lock once for that kind of thread, possibly allowing other threads to acquire the lock.

The implementation uses Mesa condition variables, one for each kind.

Note that the expression "1 – *kind*" computes "the other kind": 0 becomes 1 and 1 becomes 0.

To the right find a test program. It starts four threads of each kind. Each thread tries to acquire and release the lock zero or more times. There are 4 threads of kind 0 and 4 threads of kind 1 configured in this test program. The variable *in_cs* keeps track of how many threads of each kind are in the "critical section" (Lines 15-17). If a thread of kind 0 is in the critical section, then in_cs[0] must be at least 1. However, this critical section allows multiple threads of different kinds to be in the critical section constrained by the rules described on the previous page.

One of your tasks is to find suitable invariants for this test program for Line 9. Indicate in the table below with ✓ if the predicate is invariant and with ✗ if not.

```
1   from crosslock import *
2
3   const N = [ 4, 4 ]
4
5   thelock = CrossLock()
6
7   in_cs = [0,] * 2
8
9   invariant [redacted]
10
11  def thread(kind):
12      while choose { False, True }:
13          cl_acquire(?thelock, kind)
14          atomically in_cs[kind] += 1
15
16          assert in_cs[kind] > 0
17
18          atomically in_cs[kind] -= 1
19          cl_release(?thelock, kind)
20
21  for _ in { 1 .. N[0] }:
22      spawn thread(0)
23  for _ in { 1 .. N[1] }:
24      spawn thread(1)
```

a)	(*in_cs*[0] > 0) **or** (*in_cs*[1] > 0)	
b)	(*in_cs*[0] >= 0) **and** (*in_cs*[1] >= 0)	
c)	(*in_cs*[0] **in** { 0, 1 }) **and** (*in_cs*[1] **in** { 0, 1 })	
d)	(*in_cs*[0] **in** { 0, 1 }) **or** (*in_cs*[1] **in** { 0, 1 })	
e)	(*in_cs*[0] < *in_cs*[1]) **or** (*in_cs*[0] > *in_cs*[1])	

Answer the following questions about the cross lock implementation with ✓ or ✗:

f)	Just before Line 17 (after the **while** loop finishes), *c->count*[1 – *kind*] must be either 0 or 1.	
g)	Between Line 21 and 22 (just after acquiring the mutex), *c->count*[*kind*] must be larger than 0.	
h)	It is correct to replace Lines 23 and 24 with **notify**(?*c->cond*[*kind*]), that is, to remove Line 23.	
i)	There is no disadvantage to replacing **notifyAll**() in Line 26 with just **notify**().	

19. Dining Western Philosophers

A group of N western philosophers get together to eat at a hip local restaurant. Western philosophers eat using a knife and a fork. In the center of their table is a mug with the same number of knives and forks. Left-handed philosophers first take a fork and then a knife before they can eat; right-handed philosophers first take a knife and then a fork. If the utensil they are looking for is not available, they wait. Once a philosopher has a knife and a fork, they eat and replace the utensils. Each philosopher eats zero or more times before they leave the restaurant.

```
const N_LEFT = 2 # number of left-handed (fork-first) philosophers
const N_RIGHT = 2 # number of right-handed (knife-first) philosophers
const N_PAIRS = 3 # number of pairs of forks and knives

nforks = nknives = N_PAIRS

def take(utensil):
    atomically when !utensil > 0:
        !utensil -= 1

def replace(utensil):
    atomically !utensil += 1

def left_handed():
    while choose { False, True }:
        take(?nforks)
        take(?nknives)
        # eat
        replace(?nforks)
        replace(?nknives)

def right_handed():
    while choose { False, True }:
        take(?nknives)
        take(?nforks)
        # eat
        replace(?nknives)
        replace(?nforks)

for _ in { 1 .. N_LEFT }:
    spawn left_handed()
for _ in { 1 .. N_RIGHT }:
    spawn right_handed()
```

The code on the left models left-handed and right-handed western philosophers in Harmony. The variable *nforks* keeps track of how many forks are left in the mug, while *nknives* keeps track of the number of knives in the mug.

The method take(*utensil*) waits for a utensil of a particular type to be available and then takes one. The method replace(*utensil*) replaces a utensil of a particular type.

Each of the philosophers eats zero of more times.

For particular combinations of N_LEFT, N_RIGHT, and N_PAIRS deadlock may or may not be possible. In the table to the right, use ✓ to indicate that deadlock is possible, and ✗ to indicate that deadlock is not possible.

N_LEFT	N_RIGHT	N_PAIRS	deadlock possible?
3	3	6	
3	3	4	
3	3	3	
3	3	1	
3	0	3	
3	0	1	
2	3	4	
2	3	3	
2	3	2	
2	3	1	