School of Computing and Information Systems
SWEN90004 Modelling Complex Software Systems
Practice Exam – Sample Answers / Suggestions
Semester 1, 2019

© University of Melbourne

Question 1

(a)

Think about two threads each executing swap on two shared objects, and the four necessary and sufficient conditions (the Coffman conditions) that must occur for deadlock to happen:

- serially reusable resources: each thread shares the two objects, and (due to the synchronized keyword) accesses these objects using mutual exclusion;
- incremental acquisition: each thread obtains a lock on each of the two objects in turn;
- no pre-emption: once a thread has obtained a lock on an object, it only releases it once swap has completed executing;
- wait-for cycle: each thread may hold the lock on one of the two shared objects, and be waiting on the lock for the other.

So, deadlock can happen when two threads execute swap on two shared objects and each thread holds the lock for one object and is waiting for the lock on the other.

(b)

We can use the identity hash codes of objects to order them in linear fashion. If we always give higher priority to an object with a smaller hash value then we cannot have one thread holding the lock for a while waiting for the lock for b and at the same time have another thread holding the lock for b while waiting for a. The threads will ask for locks in the same order, hence breaking the symmetry.

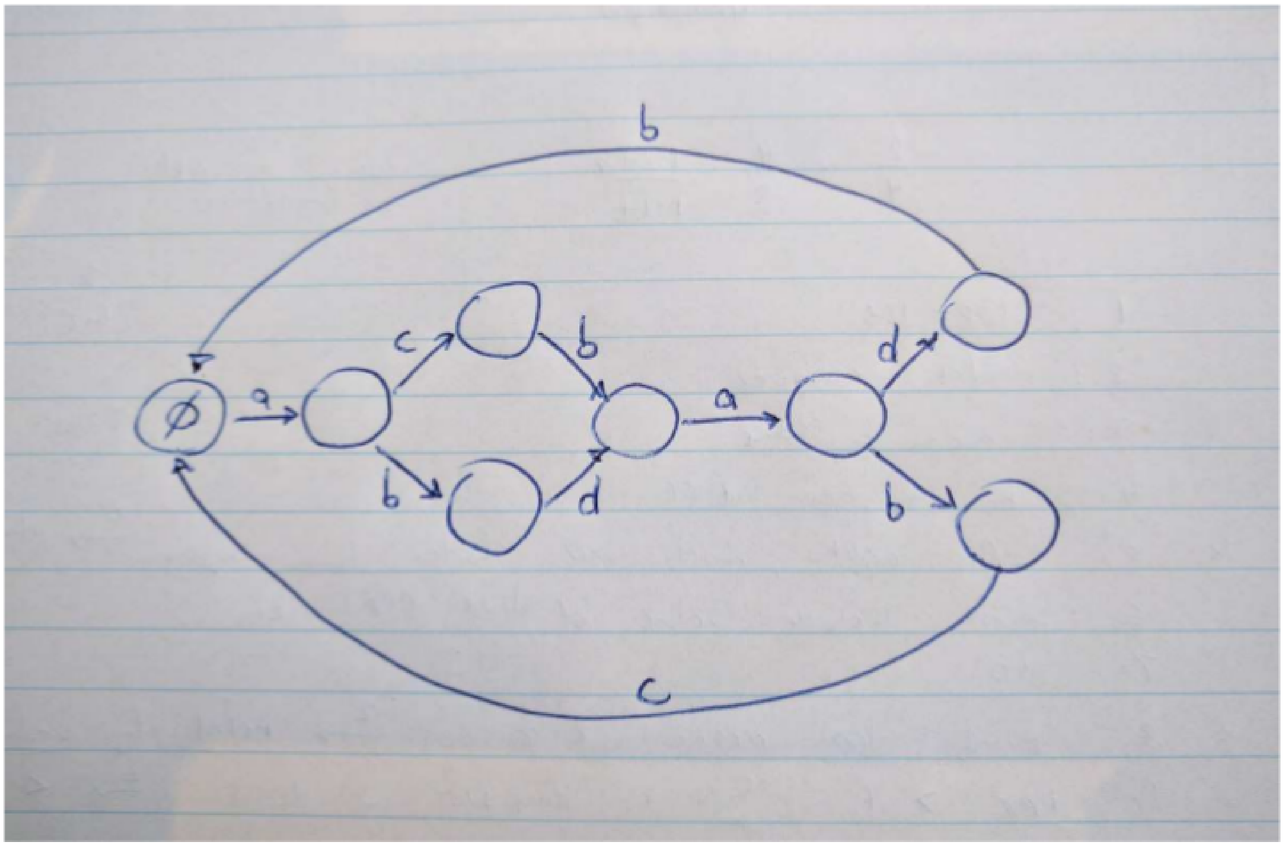
Here is an implementation:

```
public void swap(ThreadSafeValue v) {
  if (v == this)
    return;
  else if (System.identityHashCode(v) <
    System.identityHashCode(this))
    v.safeSwap(this);
  else
    this.safeSwap(v);
}

private synchronized void safeSwap(ThreadSafeValue v) {
  Object temporaryValue = this.value;
  this.value = v.getValue();
  v.setValue(temporaryValue);
}
```

Question 2

(a)



(b)

```
S = ( a -> b -> d -> END ).
```

(c)

```
ROUTINE =
   ( eat -> drink -> sleep -> ROUTINE
   | drink -> R
   ),
R =
   ( eat -> sleep -> ROUTINE
   | run -> drink -> R
   ).
```

Question 3

(a)

```
const N = 10        // number of resources
range T = 1..N      // resources requested
const U = 5         // number of users

RESOURCE_USER =
  ( get[n:T] -> use[n] -> put[n] -> RESOURCE_USER ).

RESOURCE_ALLOC = RESOURCE_ALLOC[N],
RESOURCE_ALLOC[r:0..N] =
  // note: index can be zero!
  ( when (r > 0) get[n:1..r] -> RESOURCE_ALLOC[r-n]
  | put[n:T] -> RESOURCE_ALLOC[r+n]
  ).
```

(b)

```
||SYSTEM =
  ( t[u:1..U]:RESOURCE_USER
  || {t[u:1..U]}::RESOURCE_ALLOC
  ).
```

(c)

Given that in part (b), we distinguish each `RESOURCE_USER` with label `t[u]`, we can assert the following property:

```
  assert USE = forall[u:1..U] <>t[u].use[n:T]
```

(ie, **for all** values of u in the range `1..U`, the process labelled `t[u]` will **eventually** execute the action `use`)

Question 4

```
public class Carpark {
  protected capacity;
  protected boolean open = false;
  protected int count == 0;

  Carpark(Int n) {
    capacity = n;
  }

  public synchronized void arrive()
    throws InterruptedException
  {
    while(!open || count == capacity)
      wait();
    count = count + 1;
  }

  public synchronized void depart() {
    if (count == 0)
      error();
    if (count == capacity)
      notifyAll();
    count = count - 1;
  }

  public synchronized void open() {
    open = true;
    notifyAll();
  }

  public synchronized void close() {
    open = false;
  }
}
```

While minor syntax errors (missing semicolon, etc.) are tolerated, marks would be deducted for: not using `synchronized` correctly; not using the `while/wait` pattern correctly; not throwing appropriate exceptions; not using `wait/notify` (or `notifyAll`) correctly; incorrect project logic (ie, unlikely to do the correct thing!); poor or missing declarations; blatantly incorrect syntax, etc.

Question 5

(a)

Upon terminating, the program will have printed:

```
6
12
18
24
30
36
42
```

(b)

(Almost certainly) nothing would be printed: as all goroutines are executed concurrently with the calling function (in this case `main`), `main` would finish executing and the program would terminate before any output had been printed.

(c)

An unbuffered channel will only execute sends (`chan <-`) when there is a corresponding receive (`<- chan`) ready to receive the sent value. Buffered channels can accept a finite number of values before a corresponding receiver is needed.

Question 6

(a)

Your answer could mention any of the properties discussed in introduction to complex systems lectures or after, such as many parts that interact with one another; emergent behaviour; self-organisation; etc.

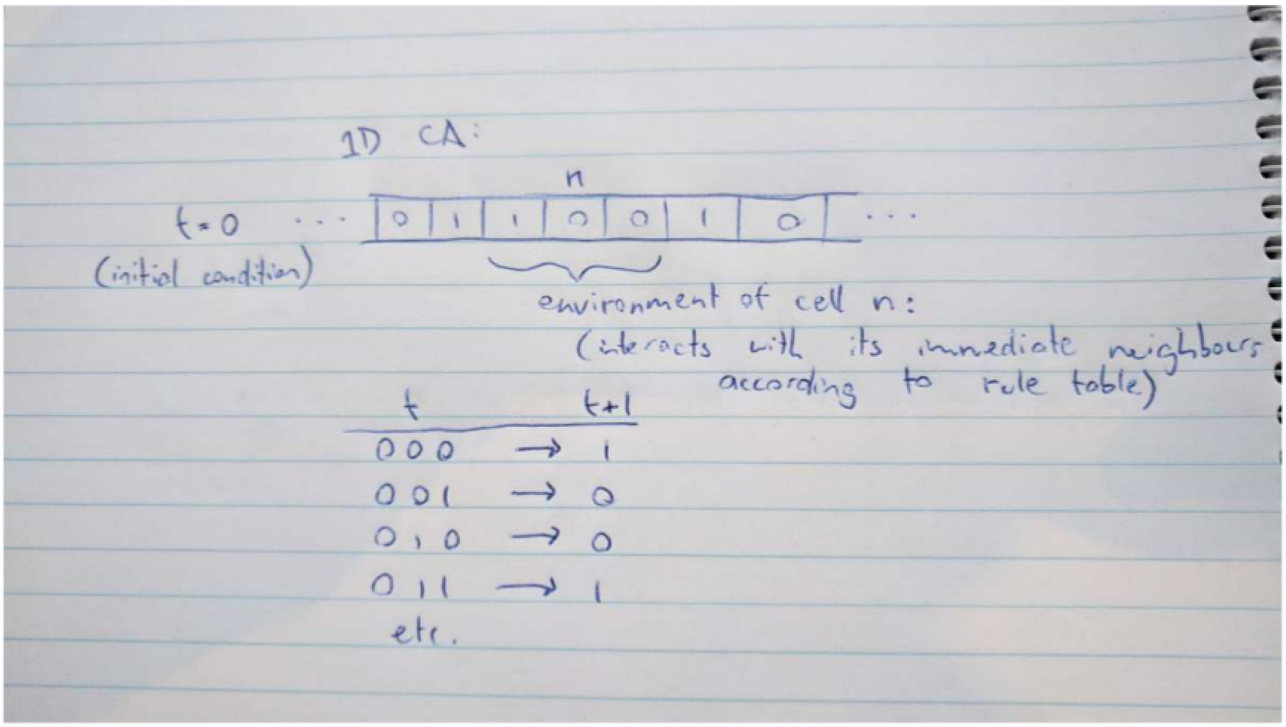
(b)

1. deterministic: logistic map maps each state to a *single* successor state.
2. aperiodic: (for appropriate values of r) logistic map produces orbits that do not visit the same state more than once.
3. bounded: state (x) of logistic map is bounded between 0 and 1; ie, can't take on values outside this range.
4. sensitive to initial conditions: (for appropriate values of r) if x' and x'' are very close at time 0, their future trajectories will diverge rapidly.

Question 7

(a)

- *interactions*: the future state of an individual cell in the CA is defined in terms of its own state and that of the neighbours with which it can be considered to interact.
- *environment*: the environment of an individual cell is the set of neighbours with which it interacts.
- *initial conditions*: the initial condition of a system is the pattern of states of each of its cells at time *t*=0.



(b)

A behaviour that is displayed at the level of a system which arises as a result of interactions between its constituent parts, but which is not evident in those individual parts. eg, “gliders” in the Game of Life.

Question 8

(a)

(i)

- a deterministic environment changes over time in a predictable fashion (eg, a seasonal temperature that follows a sine curve); if we (or the agents) know the function and current value, we can predict the future value.
- a non-deterministic environment changes over time in an unpredictable ("random" or stochastic) fashion; we cannot necessarily predict the future state of the environment.

(ii)

- a discrete environment has a finite number of distinct locations (eg a grid) that an agent may occupy;
- a continuous environment has a (potentially) infinite number of locations that an agent may occupy (to the limit of precision of the computer!)
- using a discrete or continuous environment will have implications for:
  - whether two agents can occupy the same location at the same time
  - how agent neighbourhoods, movement, etc are defined

(b)

eg ant colony:

- agents are ants
- ants are located at a particular point in (continuous) space, and may move in any direction
- ants sense the presence of chemical signals/pheromones (information) from their local environment
- pheromones may modify the decisions an ant chooses to move
- ants may deposit pheromones in their environment as they move
- hence ants interact indirectly, via signals places and received in their spatial environment (known as stigmergy)

(c)

- The neighbourhood defines how far a predator or prey is able to sense (or see) or in order to obtain information on its surroundings; alternatively, it might be the range at which the predator could `eat' prey.
- Increasing the size of the neighbourhood would increase the range at which predator and prey agents could detect each other: predator agents would be able to see (and chase) more distant prey; prey agents would be able to see (and flee from) more distant predators. Under alternative interpretation, predators would be able to "eat" more distant prey.
- The species with the larger neighbourhood would have an advantage.

Question 9

(a)

(i) Starting with a regular lattice; randomly rewire a small fraction of edges

(ii) Small world networks tend to have equivalent clustering coefficient to regular lattice (relatively high compared to a random network), but lower average path length compared to a regular lattice (equivalent to a random network).

(b)

When new nodes are added to the network, they are connected to existing nodes with a probability proportional to the existing node's current number of neighbours, so nodes with many neighbours tend to preferentially get attached to even more neighbours. This results in a highly skewed degree distribution in which some nodes have many neighbours but most nodes have few neighbours.

(c)

- Nodes = people
- Edges = contact between people that could allow infection to spread.
- What an edge means depends on the nature of the disease (eg, a sexually-transmitted infection and respiratory infection such as flu will spread along two very different networks).
- Need data on people's patterns of social contact (eg, how many contacts, are contacts “clustered”, etc.)

Question 10

(a)

Your answer could mention aspects of the scenario that may be difficult to capture in other types of models (such as ODEs): eg, space, decision making, potential heterogeneity between individuals (eg, different trappers may walk shorter or longer distances, or may require different amounts of money to keep working).

(b) (examples only, other responses are possible)

- agents: trappers, foxes, native species
- interactions: foxes eat native species; trappers catch foxes
- update rules:
  - trappers decide where to set traps, perhaps on basis of past experience;
  - foxes can get caught in traps, and can predate upon native species;
  - native species can get eaten by foxes, etc.
  - [use your imagination]

(c) (example only, other responses are possible)

- question: what is the minimal payment per fox required to attract and retain sufficient trappers to control the fox population?
- experiments:
  - assuming we don't know how much money each trapper needs to earn in order to keep coming to work, or how fast they can move through the park (and hence how far apart they can set traps), we will run a series of scenarios (a parameter sweep) in which systematically vary each of these key parameters (payment_per_fox, trapper_income_needs and trapper_speed).
  - For each scenario, we will measure the size of the population after a given period of time (eg, 6 months).
  - Our model is stochastic, therefore we will run each scenario 100 times and calculate the mean (average) size of the fox population for each parameter combination.
  - This will help us to determine, for particular trapper income requirements and movement speeds, how much money the NPWS needs to offer per fox in order to control the population
  - [again, use your imagination]

(d) (examples only, other responses are possible)

To keep the model simpler, we may assume that:

- all trappers move at the same speed
- native species are only predated upon by foxes
- native species have an unlimited source of food (ie, assumptions of standard predator-prey models)
- park is shaped like a square, with entrances (for trappers) on each side
- all part of park are equally densely covered by bush, and hence take the same amount of time to move through
- each trap can only catch a single fox (ie the first fox who encounters it)
- etc.

Possible limitations of a model that makes these assumptions are that they may not hold in the real world; think about how and what this would mean: eg, if the actual park is long and narrow and only has an entrance at one end, then our model may not reflect the fact that trappers may not be able to reach, and set traps in, very distant parts of the park.

NB: I have given quite a few examples for each of the questions above; in a real exam question, you would probably be asked for some specific number (eg two or three examples). You don't need to write paragraphs of text. Lists are fine (but make sure that you are precise/descriptive enough that the marker can understand what you intend)

Question 11

(Looking at it again, this example is actually pretty straightforward for a final question; expect the final question to be a bit trickier in the exam!)

```
const N = 6    // max number of trappers
const T = 3    // max number of tokens
range Token = 0..T

AGENT
  = AGENT[T],
AGENT[t:Token]
  = ( when (t > 0) requestToken -> AGENT[t-1]
    | when (t < T) returnToken  -> AGENT[t+1]
    ).

TRAPPER
  = (requestToken -> enter -> leave -> returnToken -> TRAPPER).

||SYSTEM
  = (trapper[1..N]:TRAPPER || {trapper[1..N]}::AGENT).
```