**INF214 – Eksamen Haust 2020**

**1)**

1. Yes
2. No
3. Yes
4. Yes
5. Yes
6. Yes
7. Yes
8. ?
9. Yes
10. Yes
11. Yes
12. No
13. No
14. Yes
15. No
16. Yes
17. No
18. No ?
19. No
20. Yes
21. Yes
22. No
23. No
24. No

**2)**

2.1) A volatile variable is a special variable in Java that is used when you want to notify the program about a variable that is updated by multiple threads in the program.

2.2) PoisonPill in Java is a built in tool that helps shutting down a program safely when needed. We have seen something similar when talking about sentinel values.

2.3) The waiting time of a thread is increased and the performance of the system is affected.

2.4) An actor will receive a message and do something depending on which message it received. Actors holds control of their own state and cannot be modified by other actors. An actor has a mailbox that messages are stored if he received multiple.

2.5) Channels is something that’s helpful when trying to code asynchronous programs. It works as a queue and is used when processes needs to communicate.

**3)**

3.1) The P operation reduces the semaphore by 1, saying that this semaphore is busy now. The V operation increments it by 1 again, saying that this semaphore is ready to perform a new task.

3.2) The purpose of a barrier is to have all threads waiting at the barrier until the last thread reaches the barrier as well.

3.3) In synchronous message passing, the sender blocks until the receiver picks up the message. In asynchronous message passing the sender does not block, it just leaves the message and continue working.

**4)**

Replaced if statements with await statements.

// global variables

int buf;

bool buf\_full = false;

const int N = 100, Ps = 10, Cs = 10;

process Producer[i = 1 to Ps] {

int a[N];

// here we fill array 'a' with data

int p = 0;

while (p < N) {

< await ( buf\_full == false ) ; buf = a[p]; buf\_full = true;>

p = p + 1;

}

}

process Consumer[i = 1 to Cs] {

int b[N];

int c = 0;

while (c < N) {

< await ( buf\_full == true ) ; b[c] = buf; >

c = c + 1;

}

// here we use array b

}

**5)**

5.1)

**Int x = 2;**

**Int y = 3;**

co

< x = x + y; >

||

< y = x \* y; >

oc

Possible values for x and y:

x = 2 + 3 = 5

y = 5 \* 3 = 15

y = 2 \* 3 = 6

x = 2 + 6 = 8

5.2)

{x>=4}

< x = x - 4; >

x>=0

{x >= 0}

< x = x + 5; >

x >= 5

They do not interfer, its all good.

**6)**

This is very similar to the bear and bees problem, but instead of the parent eating everything in the dish, he will now fill it up.

sem mutex = 1, # mutual exlusion

dish\_empty = 0; # synchr bear/bees

int portions = F; # portions in the dish

process baby [i=1 to N] {

while (true) {

P (mutex);

portions--; # eat one portion

if (portions == 0)

V (dish\_empty); # let the parent fill dish, pass mutex baton

else

V (mutex); # let other babies eat out of dish

}

}

process parent {

while (true) {

P (dish\_empty); # wait until the dish is empty

portions = F; # fill up dish

V (mutex); # let babies start eating again

}

}