UNIVERSITETET I BERGEN

KANDIDAT

111

PRØVE

INF214 0 Multiprogrammering

Emnekode	INF214
Vurderingsform	Skriftlig eksamen
Starttid	21.11.2022 14:00
Sluttid	21.11.2022 17:00
Sensurfrist	--
PDF opprettet	07.11.2023 12:18

Structure of the exam

Oppgave	Tittel	Oppgavetype
i	Exam structure	Informasjon eller ressurser

Atomic blocks

Oppgave	Tittel	Oppgavetype
1	1	Langsvar

Semaphores

Oppgave	Tittel	Oppgavetype
2	2	Langsvar
3	3	Programmering

Monitors

Oppgave	Tittel	Oppgavetype
4	4	Programmering
5	5	Paring

Communicating Sequential Processes

Oppgave	Tittel	Oppgavetype
6	6	Programmering

JavaScript Promises

Oppgave	Tittel	Oppgavetype
7	7	Programmering

8	8	Langsvar
9	9	Fyll inn tekst
ℹ	Cheat sheet about the semantics of promises	Informasjon eller ressurser
10	10.1	Flervalg
11	10.2	Flervalg
12	10.3	Flervalg
13	10.4	Flervalg

1 1

Consider the following program:

```
int x = 2;
int y = 3;
co
   < x = x + y; >
   ||
   < y = x * y; >
oc
```

What are the possible final values for **x** and **y**?
Explain how you got those values.

Fill in your answer here

Here there are 6 possible ways for this program to execute. For simplification we can say that both processes does two things. It begins by reading, and then writes to the variable. We can short this to R_1, W_1, R_2, W_2. In the chart the first value is X and the second is Y.

	R_1	R_1	R_1	R_2	R_2	R_2	
	W_1	R_2	R_2	W_2	R_1	R_1	
	R_2	W_1	W_2	R_1	W_1	W_2	
	W_2	W_2	W_1	W_1	W_2	W_1	
Value:	5, 15	5, 6	5, 6	8, 6	5, 6	5, 6	

Ord: 91

Maks poeng: 7

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

6 1 2 3 9 6 9

2 2

A semaphore is a program variable that holds an integer value. It can be manipulated only by the operations **P** and **V**. Describe the semantics of these operations.

Fill in your answer here

The semantics of the operations P() & V(), are that P() decrement the integer value of the semaphore by one, and if this value is 0 when the operation P() gets to it, the process waits. The process waits until the the operation V() comes, this operation increments the value of the semaphore by one, and then if a process is waiting for the semaphore to increment V() lets this process continue.

Ord: 72

Maks poeng: 10

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

1 3 7 8 0 6 7

3 3

Three persons *A*, *B*, and *C*, who like tea very much, have gathered to play the following game at the home of their friend *D*.

To drink a portion of tea, each of the persons *A*,*B*,*C* needs three "ingredients": the tea leaves, some boiled water, and a mug. Person *A* has the tea leaves, *B* has boiled water, and *C* has mugs. We assume that persons *A*,*B*, and *C* each has an unlimited supply of these ingredients (i.e., tea leaves, boiled water, mugs), respectively.

The host of the party (person *D*) also has an unlimited supply of the ingredients.

The situation unfolds as follows: *D* puts two random ingredients on the table. The person who has the third ingredient picks up the other two, makes the drink (i.e., puts tea leaves into a mug and adds boiled water), and then drinks it.

Person *D* waits for that person to finish.

This "cycle" is then repeated.

Write code in the AWAIT language that simulates this situation.
Represent the persons *A*,*B*,*C*,*D* as processes.
Use semaphores for synchronization.
Make sure that your solution avoids deadlock.

Fill in your answer here

```
object A = teaLeaves
object B = boiledWater
object C = mugs

sem host = 1
sem guest = 0
list ingredients = []

process guest[i = A to C]{
    while(true){
        P(guest)
        if(!ingredients.contains(this.item())){
            ingredients.add(this.item())
```

```
14              makeTea(ingredients); //This method clears the list
15              V(host)
16          }
17          else{
18              V(guest)
19          }
20
21      }
22
23  }
24
25  process host{
26      while(true){
27          P(host)
28          ingredients.addTwo(randomIng());
29          V(guest)
30      }
31  }
```

Maks poeng: 25

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

7 3 8 6 2 0 2

4 4

Recall the Readers/Writers problem: readers processes query a database and writer processes examine and modify it. Readers may access the database concurrently, but writers require exclusive access. Although the database is shared, we cannot encapsulate it by a monitor, because readers could not then access it concurrently since all code within a monitor executes with mutual exclusion. Instead, we use a monitor merely to arbitrate access to the database. The database itself is global to the readers and writers.

In the Readers/Writers problem, the *arbitration monitor* grants permission to access the database. To do so, it requires that processes inform it when they want access and when they have finished. There are two kinds of processes and two actions per process, so the monitor has four procedures: **request_read**, **release_read**, **request_write**, **release_write**. These procedures are used in the obvious ways. For example, a reader calls **request_read** before reading the database and calls **release_read** after reading the database.

To synchronize access to the database, we need to record how many processes are reading and how many processes are writing. In the implementation below, **nr** is the number of readers, and **nw** is the number of writers; both of them are initially 0. Each variable is incremented in the appropriate request procedure and decremented in the appropriate release procedure.

```
monitor ReadersWriters_Controller {
  int nr = 0;
  int nw = 0;
  cond OK_to_read; // signalled when nw == 0

  procedure request_read() {
    wait(OK_to_read);
    nr = nr + 1;
  }

  procedure release_read() {
    nr = nr - 1;
  }

  procedure request_write() {
    nw = nw + 1;
  }

  procedure release_write() {
    nw = nw - 1;
  }
}
```

A beginner software developer has implemented this code, but has unfortunately missed a lot of details related to synchronization. Help the beginner developer fix this code.

Note: Your solution does not need to arbitrate between readers and writers.

Fill in your answer here

```
1 // Since we dont want people to both read and write at the same time, I created a ne
```

```
2  // keeps track over when it is ready to write, and the one that the junior developer
3  // to read. And since a monitor only executes one procedure at a time mutal exclutio
4  // need to keep track of this.
5  monitor ReadersWriters_Controller {
6
7     int nr = 0;
8     int nw = 0;
9
10    cond OK_to_read; // signalled when nw == 0
11    cond OK_to_write; // signalled when nr == 0
12
13    procedure request_read() {
14       if(nw > 0){
15          wait(OK_to_read);
16       }
17       nr = nr + 1;
18    }
19
20    procedure release_read() {
21       nr = nr - 1;
22       if(nr == 0){
23         signal_all(OK_to_write)
24       }
25    }
26
27    procedure request_write() {
28       if(nw > 0){
29          wait(OK_to_write);
30       }
31       nw = nw + 1;
32    }
33
34    procedure release_write() {
35       nw = nw - 1;
36       if(nw == 0){
37         signal_all(OK_to_read)
38       }
39    }
40 }
```

Maks poeng: 20

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

4 4 8 5 0 1 2

5 5

There is duality between monitors and message passing. What is that duality exactly?

In the table, the rows represent notions about monitors, and the columns represent notions about message passing.

Click the circle in a cell to represent that a notion about monitors is dual to a notion about message passing.

Please match the values:

	`send reply()`	retrieve and process pending request	arms of case statement on operation kind	`send request(); receive reply()`	`receive request()`	save pending request	local server variables
monitor entry	◉	○	○	○	○	○	○
procedure return	○	○	○	◉	○	○	○
permanent variables	○	○	○	○	○	○	◉
`signal`	◉	○	○	○	○	○	○
`wait`	○	○	○	○	◉	○	○
procedure bodies	○	○	◉	○	○	○	○
procedure identifiers	○	○	○	○	○	◉	○
procedure call	○	◉	○	○	○	○	○

Maks poeng: 6

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

1 9 1 2 3 1 4

6 6

Using Communicating Sequential Processes, define a process **Copy** that copies a character from process **Vestland** to process **Bergen**.

Fill in your answer here

```
process Vestland(){
    send(randomChar());
}

process Bergen(){
    recive(listen());
}
```

Maks poeng: 5

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

3 1 0 4 8 6 5

7 7

Using JavaScript, define a promise which is immediately resolved. Use **console.log** to print out the value of the promise.

Fill in your answer here

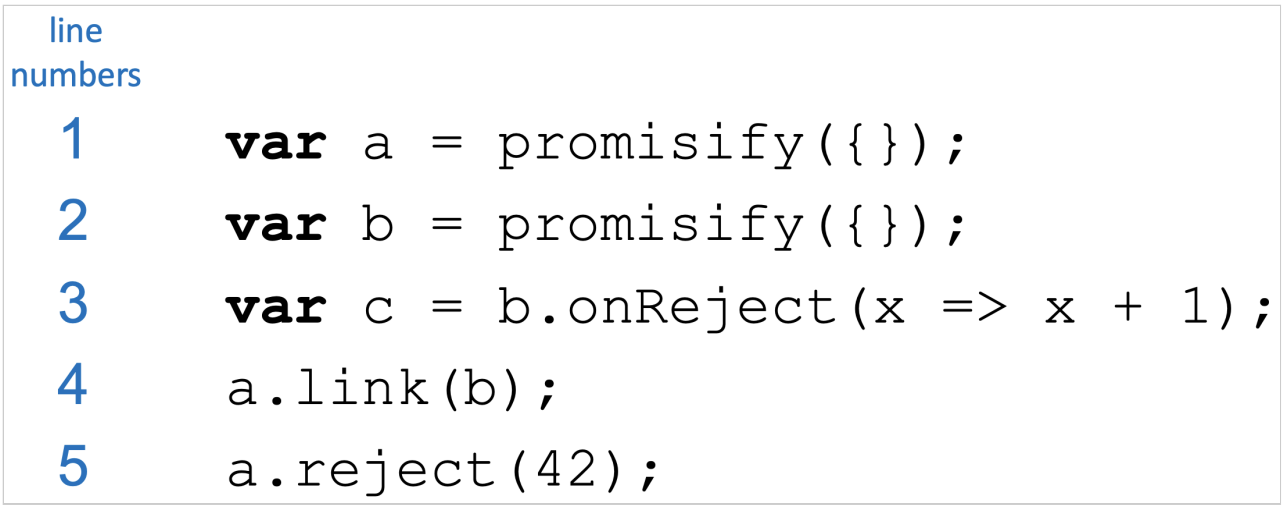
```
function* promise(){
    yield 1
}
const p = new promise()
console.log(p.next())
```

Maks poeng: 4

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

7 8 9 2 3 2 7

8 **8**



Consider the JavaScript code on the image.

Note the syntax here is a blend of JavaScript and λ_p, which uses:

- **promisify** *to create a promise,*
- **onReject** *to register a reject reaction,*
- **link** *to link to promises (linking means that when the original promise is resolved/rejected, then the linked promise will be resolved/rejected with the same value)*

Draw a promise graph for this code.

Remember to use the names of nodes in that graph that represent the "type" of node:
v for value
f for function
p for promise

with a subscript that represents the **line number** where that particular value/function/promise has been **declared / where it appears first**.

For example, the value 42 on line 5 will be denoted by $\mathbf{v_5}$ in the promise graph.

Please draw the promise graph on a piece of paper that you will get during the exam.
Please draw the promise graph on a piece of paper that you will get during the exam.

Paper

Ord: 1

Maks poeng: 10

Knytte håndtegninger til denne

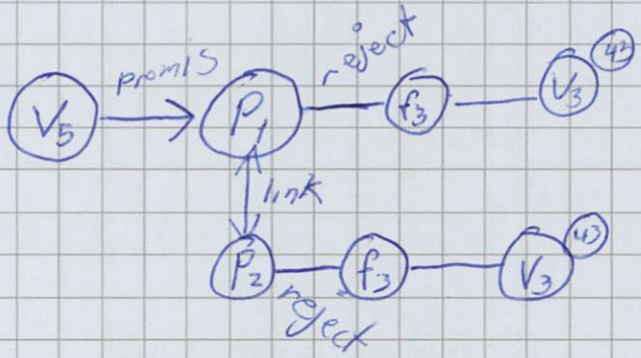
oppgaven?
Bruk følgende kode:

8 1 9 2 1 7 9

Håndtegning 1 av 1

Oppgavekode Question code	Dato Date	Emnekode Subject code	Kandidatnummer Candidate number	Oppgavenummer Question number	Sidetall Page number
81921 79	11/21/22	INF214	111	8	1 av/of 1

Tegneområde Drawing area



9 9

Consider the following HTML/JavaScript attached in the PDF file to this question.

This code runs on a computer of a super-user, who clicks the button **myButton** 6 milliseconds after the execution starts.

What happens at particular time points?
Write an integer number in each of the text boxes. If something mentioned in the left column on the table does not happen, then write "-1" (negative one) in the corresponding right column.

what happens	at what time
`**clickHandler**` finishes	28 milliseconds
`**clickHandler**` starts	14 milliseconds
interval fires for the first time	24 milliseconds
interval fires for the second time	34 milliseconds
interval fires for the third time	44 milliseconds
interval fires for the fourth time	-1 milliseconds
`**intervalHandler**` starts	6 milliseconds
`**intervalHandler**` finishes	14 milliseconds
mainline execution starts	***0*** milliseconds
mainline execution finishes	46 milliseconds
promise handler starts	14 milliseconds
promise handler finishes	18 milliseconds
promise resolved	a tiny bit after 18 milliseconds
`**timeoutHandler**` starts	0 milliseconds
`**timeoutHandler**` finishes	6 milliseconds
timer fires	16 milliseconds
user clicks the button	***6*** milliseconds

Maks poeng: 9

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

8 3 6 5 8 5 4

10 10.1

$$\frac{\begin{array}{c} a \in Addr \qquad a \in \mathrm{dom}(\sigma) \qquad \psi(a) = \mathsf{P} \\ a' \in Addr \qquad a' \notin \mathrm{dom}(\sigma) \qquad \psi' = \psi[a' \mapsto \mathsf{P}] \qquad \sigma' = \sigma[a' \mapsto \{\}] \\ f' = f[a \mapsto f(a) ::: (\lambda, a')][a' \mapsto Nil] \qquad r' = r[a' \mapsto Nil] \end{array}}{\langle \sigma, \psi, f, r, \pi, E[a.\mathtt{onResolve}(\lambda)] \rangle \rightarrow \langle \sigma', \psi', f', r', \pi, E[a'] \rangle}$$

What does this rule describe?
Select one alternative:

- This rule extracts a fulfill reaction from the queue, executes it with the promise's value, and uses the returned value to resolve the dependent promise.
- This rule states that resolving a settled promise has no effect.
- This rule registers a fulfill reaction on a pending promise.
- This rule handles the case when a pending promise is resolved.
- This rule handles the case when a fulfill reaction is registered on a promise that is already resolved.

Maks poeng: 1

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

4 4 1 4 6 5 7

11 **10.2**

$$\frac{a \in Addr \qquad a \in \mathsf{dom}(\sigma) \qquad \psi(a) \in \{\mathsf{F}(v'),\ \mathsf{R}(v')\}}{\langle \sigma, \psi, f, r, \pi, E[a.\mathtt{resolve}(v)] \rangle \rightarrow \langle \sigma, \psi, f, r, \pi, E[\mathsf{undef}] \rangle}$$

What does this rule describe?
Select one alternative:

- This rule turns an address into a promise
- This rule handles the case when a pending promise is resolved.
- This rule handles the case when a fulfill reaction is registered on a promise that is already resolved.
- This rule states that resolving a settled promise has no effect.

Maks poeng: 1

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

9 3 7 6 1 8 3

12 10.3

$$\frac{a_1 \in Addr \qquad a_1 \in \mathsf{dom}(\sigma) \qquad a_2 \in Addr \qquad a_2 \in \mathsf{dom}(\sigma) \qquad \psi(a_1) = \mathsf{F}(v) \qquad \pi' = \pi :::: (\mathsf{F}(v), \mathsf{default}, a_2)}{\langle \sigma, \psi, f, r, \pi, E[a_1.\mathtt{link}(a_2)] \rangle \rightarrow \langle \sigma, \psi, f, r, \pi', E[\mathsf{undef}] \rangle}$$

What does this rule describe?

Select one alternative:

- This rule causes a promise to be "linked" to another, with no regards to the state of that original promise.
- This rule causes a non-settled promise to be "linked" to another.
- This rule causes a pending promise to be "linked" to another.
- This rule causes an already settled promise to be "linked" to another.

Maks poeng: 1

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

2 9 7 8 7 1 3

13 **10.4**

$$\frac{a \in Addr \qquad a \in \mathrm{dom}(\sigma) \qquad a \notin \mathrm{dom}(\psi) \qquad \psi' = \psi[a \mapsto \mathsf{P}] \qquad f' = f[a \mapsto Nil] \qquad r' = r[a \mapsto Nil]}{\langle \sigma, \psi, f, r, \pi, E[\texttt{promisify}(a)] \rangle \rightarrow \langle \sigma, \psi', f', r', \pi, E[\mathsf{undef}] \rangle}$$

What does this rule describe?
Select one alternative:

- This rule clears fulfill and reject reactions of a settled promise.
- This rule registers a reject reaction on a pending promise.
- This rule enables evaluation and recomposition of expressions according to the evaluation contexts.
- This rule registers a fulfill reaction on a pending promise.
- This rule turns an address into a promise.

Maks poeng: 1

Knytte håndtegninger til denne oppgaven?
Bruk følgende kode:

3 2 3 2 1 8 6