Exam structure

The exam has 10 questions.

question	topic	points	expected time needed to solve
1	atomic blocks in AWAIT language	7	10 minutes
2	semaphores in AWAIT language	10	10 minutes
3	semaphores in AWAIT language	25	40 minutes
4	monitors in AWAIT language	20	30 minutes
5	monitors in AWAIT language	6	10 minutes
6	CSP	5	10 minutes
7	JavaScript promises	4	5 minutes
8	promise graph (JavaScript)	10	25 minutes
9	Event loop (JavaScript)	9	20 minutes
10.1	semantics of promises (JavaScript)	1	3 minutes
10.2		1	3 minutes
10.3		1	3 minutes
10.4		1	3 minutes

Total number of points: 100.
Total time: a bit less than 3 hours.

This is INF214 H22 exam!
The exam H23 might have a different structure!

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 Help

Format | B I U x_2 x^2 I_x | Ω | Σ

This is INF214 H22 exam!
The exam H23 might have a different structure!

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 Help

Format | B I U x_2 x^2 T_x | Ω | Σ

Words: 0

This is INF214 H22 exam!
The exam H23 might have a different structure!

3 **3**

Three persons, who like gløgg very much, have gathered to play the following game in a bar.
To drink a portion of gløgg, each of them obviously needs three "ingredients": the gløgg (liquid in the decanter -- *see image*), a mug, and almonds. One player has the gløgg (liquid in the decanter), the second player has mugs, and the third has almonds. Assume that each of the players has an unlimited supply of these ingredients (i.e., gløgg in the decanter, mugs, almonds), respectively.
The barista, who also has an unlimited supply of the ingredients, puts two random ingredients on the table.
The player who has the third ingredient picks up the other two, makes the drink (i.e., pours the gløgg liquid from the decanter into a mug and adds almonds), and then drinks it.
The barista waits for the player to finish.
This "cycle" is then repeated.

"Simulate" this behaviour in the AWAIT language.
Represent the players and the barista as processes.
Use semaphores for synchronization.
Make sure that your solution avoids deadlock.
Fill in your answer here

1

This is INF214 H22 exam!
The exam H23 might have a different structure!

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;
  }
```

This is INF214 H22 exam!
The exam H23 might have a different structure!

```
  }

  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
```

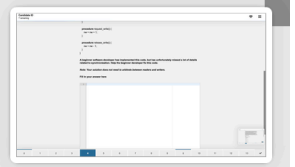
This is INF214 H22 exam!
The exam H23 might have a different structure!

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:

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

This is INF214 H22 exam!
The exam H23 might have a different structure!



Check answer

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

```
1
```

This is INF214 H22 exam!
The exam H23 might have a different structure!

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

This is INF214 H22 exam!
The exam H23 might have a different structure!

8 **8**

line numbers

```
1   var a = promisify({});
2   var b = promisify({});
3   var c = b.onReject(x => x + 1);
4   a.link(p2);
5   a.reject(42);
```

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.

This is INF214 H22 exam!
The exam H23 might have a different structure!

```
<button id="myButton"></button>
<script>

   setTimeout(function timeoutHandler() {
      /* code that runs for 6 ms */
    }, 10);

    setInterval(function intervalHandler() {
      /* code that runs for 8 ms */
    }, 10);

    const myButton = document.getElementById("myButton");

    myButton.addEventListener("click", function clickHandler() {
       Promise.resolve().then( () => {
          /* some promise handling code that runs for 4 ms */ }
       );

       /* click-handling code that runs for 10 ms */
    });

    /* code that runs for 18 ms */
</script>
```

This is INF214 H22 exam!
The exam H23 might have a different structure!

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?

what happens	at what time
`**clickHandler**` finishes	milliseconds
`**clickHandler**` starts	milliseconds
interval fires for the first time	milliseconds
interval fires for the second time	milliseconds
interval fires for the third time	milliseconds
interval fires for the fourth time	milliseconds
`**intervalHandler**` starts	milliseconds
`**intervalHandler**` finishes	milliseconds
mainline execution starts	***0*** milliseconds
mainline execution finishes	milliseconds
promise handler starts	milliseconds
promise handler finishes	milliseconds
promise resolved	a tiny bit after milliseconds
`**timeoutHandler**` starts	milliseconds
`**timeoutHandler**` finishes	milliseconds
timer fires	milliseconds
user clicks the button	***6*** milliseconds

Check answer

Cheat sheet about the semantics of promises

The following four questions will ask to explain in natural language the formal semantics of promises in JavaScript.

Please note that each of those four questions is worth 1 point.
The detailed description below is only provided for the purposes of the completeness of the question formulation.

<u>Do NOT spend time reading this, but rather look at each of the four questions first, and only come back to the description below to clarify the relevant notation used in the question itself!</u>

This is INF214 H22 exam!
The exam H23 might have a different structure!

Recall the paper on "*A Model for Reasoning About JavaScript Promises*", which defines the calculus λ_p and formal semantics for JavaScript promises.

The **syntax** of the calculus λ_p has the following expressions:

- $\mathrm{promisify}(e)$ turns an object into a promise
- $e.\,\mathrm{resolve}(e)$ fulfills a promise with a value and causes its resolve reactions to be scheduled for execution by the event loop
- $e.\,\mathrm{reject}(e)$ rejects a promise with a value and causes its reject reactions to be scheduled for execution by the event loop
- $e.\,\mathrm{onResolve}(e)$ registers a resolve reaction on a promise and returns a dependent promise
- $e.\,\mathrm{onReject}(e)$ registers a reject reaction on a promise and returns a dependent promise
- $e.\,\mathrm{link}(e)$ registers a dependency between two promises such that when the former is resolved (or rejected) the latter is resolved (or rejected) with the same value

The calculus λ_p has a small-step reduction operational semantics with several rules that express how the expressions of λ_p are interpreted.
The **runtime** of λ_p has:

- promise state ψ that maps each address to an algebraic data type $\psi \in \mathrm{PromiseValue}$, which is one of: pending P, fulfilled $\mathrm{F}(v)$, rejected $\mathrm{R}(v)$, where v is the result value of the promise
- fulfill reactions f that map an address to a list of *reaction* and *dependent promise* pairs for a pending promise
- reject reactions r that map an address to a list of *reaction* and *dependent promise* pairs for a pending promise
- reaction c: either a lambda function or a special default reaction (this special default reaction can be thought

- $\text{promisify}(e)$ turns an object into a promise
- $e.\text{resolve}(e)$ fulfills a promise with a value and causes its resolve reactions to be scheduled for execution by the event loop
- $e.\text{reject}(e)$ rejects a promise with a value and causes its reject reactions to be scheduled for execution by the event loop
- $e.\text{onResolve}(e)$ registers a resolve reaction on a promise and returns a dependent promise
- $e.\text{onReject}(e)$ registers a reject reaction on a promise and returns a dependent promise
- $e.\text{link}(e)$ registers a dependency between two promises such that when the former is resolved (or rejected) the latter is resolved (or rejected) with the same value

The calculus λ_p has a small-step reduction operational semantics with several rules that express how the expressions of λ_p are interpreted.
The **runtime** of λ_p has:

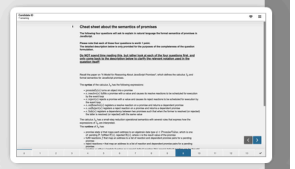
- promise state ψ that maps each address to an algebraic data type $\psi \in \text{PromiseValue}$, which is one of: pending P, fulfilled $\text{F}(v)$, rejected $\text{R}(v)$, where v is the result value of the promise
- fulfill reactions f that map an address to a list of *reaction* and *dependent promise* pairs for a pending promise
- reject reactions r that map an address to a list of *reaction* and *dependent promise* pairs for a pending promise
- reaction ρ: either a lambda function or a special default reaction (this special default reaction can be thought of as the identity function)
- queue π : a queue of scheduled reactions, i.e., promises that have been settled for which the reactions are awaiting asynchronous computation by the event loop

In addition, σ represents the heap, and $Addr$ represents an address in the heap.
A program **state** is represented as a tuple: $\langle \sigma, \psi, f, r, \pi \rangle$ (i.e., the *heap*, the *promise state*, the *list of fulfill reactions*, the list of *reject reactions*, the *queue of the event loop*).

Other notation used:

- Nil represents an empty list
- $x :: xs$ is list concatenation
- $xs ::: ys$ represents the append operation

This is INF214 H22 exam!
The exam H23 might have a different structure!



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

Maximum marks: 1 Check answer

This is INF214 H22 exam!
The exam H23 might have a different structure!

11 **10.2**

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

What does this rule describe?
Select one alternative:

- ○ This rule handles the case when a pending promise is resolved.
- ○ This rule turns an address into a promise
- ○ 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.

Maximum marks: 1 Check answer

This is INF214 H22 exam!
The exam H23 might have a different structure!

12 **10.3**

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

What does this rule describe?

Select one alternative:

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

Maximum marks: 1 Check answer

This is INF214 H22 exam!
The exam H23 might have a different structure!

13 **10.4**

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

What does this rule describe?
Select one alternative:

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

Maximum marks: 1 Check answer

This is INF214 H22 exam!
The exam H23 might have a different structure!