UNIVERSITETET I BERGEN

KANDIDAT

93

PRØVE

INF214 0 Multiprogrammering

Emnekode	INF214
Vurderingsform	Skriftlig eksamen
Starttid	26.11.2025 08:00
Sluttid	26.11.2025 11:00
Sensurfrist	--
PDF opprettet	17.12.2025 12:09

Exam Structure

Oppgave	Tittel	Status	Poeng	Oppgavetype
i	Exam Structure			Informasjon eller ressurser

Shared memory concurrency

Oppgave	Tittel	Status	Poeng	Oppgavetype
1.1	Dining Philosophers	Riktig	3/3	Sammensatt
1.2	Ticket Algorithm	Delvis riktig	2/6	Sammensatt
1.3	Barrier synchronization	Riktig	6/6	Sammensatt
1.4	Savings Account	Riktig	8/8	Nedtrekk
1.5	Monitors (Java)	Riktig	8/8	Sammensatt
1.6	Properties of concurrent programs	Delvis riktig	1/2.5	Paring

JavaScript promises

Oppgave	Tittel	Status	Poeng	Oppgavetype
2.1	Promise graph	Delvis riktig	6.5/7	Plasser i bilde
2.2	Finding a bug	Riktig	10/10	Sammensatt
2.3	Semantics of promises: reading a rule (question 1 of 2)	Riktig	1.5/1.5	Feltvalg
2.4	Semantics of promises: reading a rule (question 2 of 2)	Delvis riktig	0.5/1.5	Feltvalg
2.5	Semantics of promises: writing a rule	Delvis riktig	1/1.5	Dra og slipp

Concurrency in web browsers

Oppgave	Tittel	Status	Poeng	Oppgavetype
3.1	Running C++ code in browser	Riktig	10/10	Plasser i bilde

Points from Portfolio Sets

Oppgave	Tittel	Status	Poeng	Oppgavetype
4.1	Portfolio Sets	Ubesvart	35/35	Muntlig

1.1 Dining Philosophers

Five philosophers sit around a circular table. Between each adjacent pair is one fork (so 5 forks total). To eat, a philosopher must hold both adjacent forks; otherwise they think. Forks are exclusive (one philosopher at a time). Each philosopher first attempts to pick up their **left** fork, then their **right** fork, blocking if a fork is unavailable.

Question 1:

What is the *maximum* number of philosophers that can be eating simultaneously?

- One
- Five
- Three
- Four
- [✓ Two] ✓

Question 2:

Suppose philosopher P_0 always picks up **right** fork first, while all others pick **left** first.

Is deadlock possible in this scenario?

- Yes, deadlock is possible.
- [✓ No, deadlock is not possible.] ✓

Maks poeng: 3

1.2 Ticket Algorithm

Consider the following code for the Ticket Algorithm.

```
int number = 1;
int next = 1;
int turn[1:n] = (0, ..., 0);

process CS[i=1 to n] {
  while(true) {

    <turn[i] = number;
     number = number + 1;>

    <await (turn[i] == next);>

    // critical section;

    <next = next + 1;>

    // non-critical section;
  }
}
```

Note that this code uses explicit atomic statements **<...>** from the A.W.A.I.T. language. Rewrite this code so that it does not use the explicit atomic statements.

```
int number = 1;
int next = 1;
int turn[1:n] = (0,...,0);

process CS[i=1 to n] {
  while(true) {
```

turn[i] = TestAndTestAndSet(number, 1); ✗ (turn[i] = FetchAndAdd(number, 1);, turn[i] = TestAndSet(number, 1);, turn[i] = number + 1;, turn[i] = TestAndTestAndSet(number, 1);)

while (turn[i] != next) { }; ✓ (while (next) { };, while (turn[i] != next) { };, while (turn[i] == next) { };, while (turn[i] > next) { };, while (turn[i] != turn[i-1]) { };)

// critical section;

FetchAndAdd(next, next + 1); ✗ (next = next + 1;, FetchAndAdd(next, next + 1);, while(TestAndSet(next)) { next = next + 1; })

```
    // non-critical section;
  }
}
```

Maks poeng: 6

1.3 Barrier synchronization

Recall the Coordinator-Worker technique to used implement barrier synchronization. Below is the code in the A.W.A.I.T. language, which contains some missing parts. Fill in the missing parts.

```
int arrive[1:n] = (0, ..., 0);
int continue[1:n] = (0, ..., 0);

process Worker[i=1 to n] {
    while(true) {
        /* do the business logic of task i; */
        [arrive[i] = 1] ✓ ;
        < [await (continue[i] == 1)] ✓ ; >
        continue[i] = [0] ✓ ;
    }
}

process Coordinator {
    while(true) {
        for[i=1 to n] {
            < await (arrive[i] == [1] ✓ ) >;
            [arrive[i] = 0] ✓ ;
        }
        for[i=1 to n] [continue[i] = 1] ✓ ;
    }
}
```

Maks poeng: 6

1.4 Savings Account

A savings account is shared by several people (processes). Each person may deposit or withdraw funds from the account. The current balance in the account is the sum of all deposits to date minus the sum of all withdrawals to date. The balance must never become negative. A deposit never has to delay (except for mutual exclusion), but a withdrawal has to wait until there are sufficient funds.

A software developer was asked to implement a monitor to solve this problem, using Signal-and-Continue discipline. Below is the code the developer has written so far.
Help the developer finish the implementation.

```
monitor Account {
   int balance = 0;
   cond cv;

   procedure deposit(int amount) {

      [// nothing is needed here] ✓ (if (balance < 0) signal(cv), // nothing is needed here, signal(cv), if (balance > 0) wait(cv), if (balance < 0) wait(cv), if (balance > 0) signal(cv), wait(cv), signal_all(cv));
      balance = balance + amount;

      [signal_all(cv)] ✓ (// nothing is needed here, wait(cv), if (empty(cv)) wait(cv), signal_all(cv));
   }

   procedure withdraw(int amount) {

      [while (amount > balance) wait(cv)] ✓ (// nothing is needed here, while (amount > balance) signal(cv), while (balance > amount) signal(cv), while (amount > balance) wait(cv), signal(cv), while(empty(cv)) wait(cv);, signal_all(cv), while(empty(cv)) signal(cv);, wait(cv), while (balance > amount) wait(cv));
      balance = balance - amount;

      [// nothing is needed here] ✓ (signal(cv), wait(cv), // nothing is needed here, signal_all(cv), empty(cv));
   }
}
```

Maks poeng: 8

1.5 Monitors (Java)

Fill in the blanks in the Java code below.

```
import java.util.concurrent.locks.Condition;
public class ExamCounter {

 // 1. we can't decrement the counter if `count` is 0
 private int count = 0;

 // 2. we can't increment the counter if `count` is at `limit`
 private final int limit = 50;

 private final Lock lock = new ReentrantLock();

 private final Condition notAtLimit = lock ✓ (this, Condition, lock).newCondition();

 private final Condition notAtZero = lock ✓ (this, lock, Condition).newCondition();

 public int increment() throws InterruptedException {
  lock.lock();
  try {

   while ( count == limit ✓ (count == 0, count != 0, count != limit, count == limit)) {

    notAtLimit. await ✓ (await, signal, signalAll)();
   }
   count++;

   notAtZero. signalAll ✓ (await, signalAll)();

   return count;
  }
  finally {

   lock. unlock ✓ (lock, unlock)();
  }

 }
 public int decrement() throws InterruptedException {
  lock.lock();
  try {

   while ( count == 0 ✓ (count > limit, count < 0, count == limit, count != 0, count == 0, count >= 0, count
!= limit)) {

    notAtZero. await ✓ (await, signalAll, signal)();
    }
    count--;


    notAtLimit.[signalAll] ✓ (await, signalAll)();
    return count;
  }
  finally {


    lock.[unlock] ✓ (lock, unlock)();
  }

 }
}
```

Maks poeng: 8

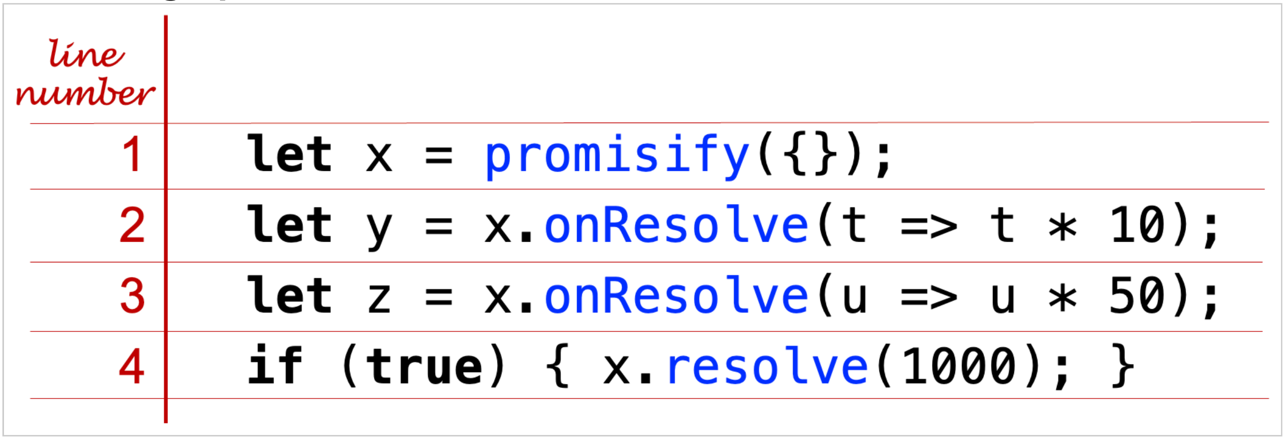
1.6 Properties of concurrent programs

For each of the properties given in the rows of the table, tell whether it is a **safety** property, a **liveness** property, or **neither.**

	safety property	neither safety nor liveness property	liveness property
Absence of deadlock	● ✓	○	○
Eventual entry	● ✗	○	○ ✓
Absence of unnecessary delay	○ ✓	○	● ✗
Absence of livelock	● ✓	○	○
Mutual exclusion	○ ✓	● ✗	○

Maks poeng: 2.5

2.1 Promise graph



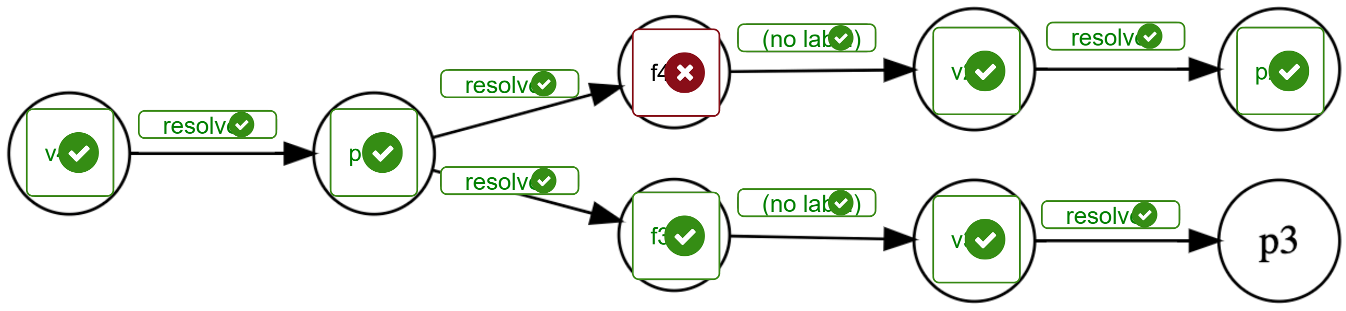
```
line
number
let x = promisify({});
let y = x.onResolve(t => t * 10);
let z = x.onResolve(u => u * 50);
if (true) { x.resolve(1000); }
```

Consider the code on the image. Note the syntax here is a blend of JavaScript and λ_p.

By drag-and-dropping, draw a promise graph for this code.

Hjelp

v3	v2	resolve	p1	f2
reject	p2	v1	p4	f1
f4	(no label)	f3	v4	



Maks poeng: 7

2.2 Finding a bug

Consider the code and the explanations in the attached PDF.
The bug can be fixed by modifying exactly a single line in the attached code.

Write below the number of that line, and what the correct code on that line should be.

Answer:

To avoid the bug from happening, line [22] ✓ (2, 3, 17, 18, 19, 21, 22, 23, 27, 28, 29, 31, 32, 33) should be have been written in the following way:

[return bcrypt.compare(password, user.password_digest)] ✓

Maks poeng: 10

2.3 Semantics of promises: reading a rule (question 1 of 2)

Consider the following inference rule from the semantics definition of λ_p.

Which part of the rule expresses the fact that *after calling* **.resolve(v)** *on some promise* **a***, the state of that promise will be "Fulfilled"*?

Click on the correct part of the inference rule.

$$\frac{\boxed{a \in Addr} \quad \boxed{a \in \mathrm{dom}(\sigma)} \quad \boxed{\psi(a) = \mathrm{P}} \quad \boxed{f(a) = (\lambda_1, a_1) \cdots (\lambda_n, a_n)} \quad \boxed{f' = f[a \mapsto \mathrm{Nil}]} \quad \boxed{r' = r[a \mapsto \mathrm{Nil}]} \quad \boxed{\psi' = \psi[a \mapsto \mathrm{F}(v)]}\ ✓ \quad \boxed{\pi' = \pi :::: (\mathrm{F}(v), \lambda_1, a_1) \cdots (\mathrm{F}(v), \lambda_n, a_n)}}{\boxed{\langle \sigma, \psi, f, r, \pi, E[a.\mathrm{resolve}(v)] \rangle} \rightarrow \boxed{\langle \sigma, \psi', f', r', \pi', E[\mathrm{undef}] \rangle}}$$

Maks poeng: 1.5

2.4 Semantics of promises: reading a rule (question 2 of 2)

We continue considering the inference rule from the previous task.

Which part(s) of the rule extract(s) the fulfill reactions of the promise and schedule(s) them for execution?

Click on the correct part or the correct parts of the inference rule.

$a \in Addr$ $\quad$ $a \in \mathrm{dom}(\sigma)$ $\quad$ $\psi(a) = \mathrm{P}$

$f(a) = (\lambda_1, a_1) \cdots (\lambda_n, a_n)$

$f' = f[a \mapsto \mathrm{Nil}]$ $\quad$ $r' = r[a \mapsto \mathrm{Nil}]$

$\psi' = \psi[a \mapsto \mathrm{F}(v)]$

[✓ $\pi' = \pi :::: (\mathrm{F}(v), \lambda_1, a_1) \cdots (\mathrm{F}(v), \lambda_n, a_n)$]

$\langle \sigma, \psi, f, r, \pi, E[a.\mathrm{resolve}(v)] \rangle \rightarrow \langle \sigma, \psi', f', r', \pi', E[\mathrm{undef}] \rangle$

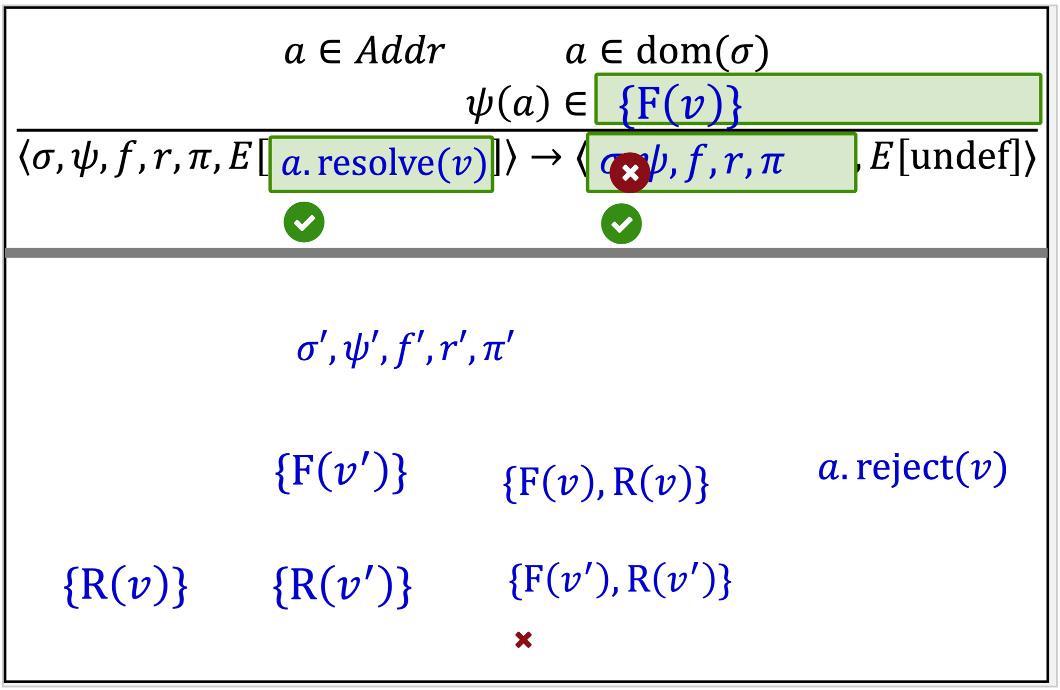
Maks poeng: 1.5

2.5 Semantics of promises: writing a rule

Consider an inference rule that states that resolving a settled promise has no effect. The rule below has 3 missing parts. Fill in those parts by drag-and-dropping the parts from the bottom of the image to their correct places in the inference rule.

Please note that there are 10 parts available in the bottom of the image, but only 3 of them should be used. This means that 7 of those parts should not be used in the inference rule.

Drag-and-drop parts from the bottom part of the image to their correct places in the inference rule.



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

$\sigma', \psi', f', r', \pi'$

$\{\mathrm{F}(v')\}$ $\{\mathrm{F}(v), \mathrm{R}(v)\}$ $a.\mathrm{reject}(v)$

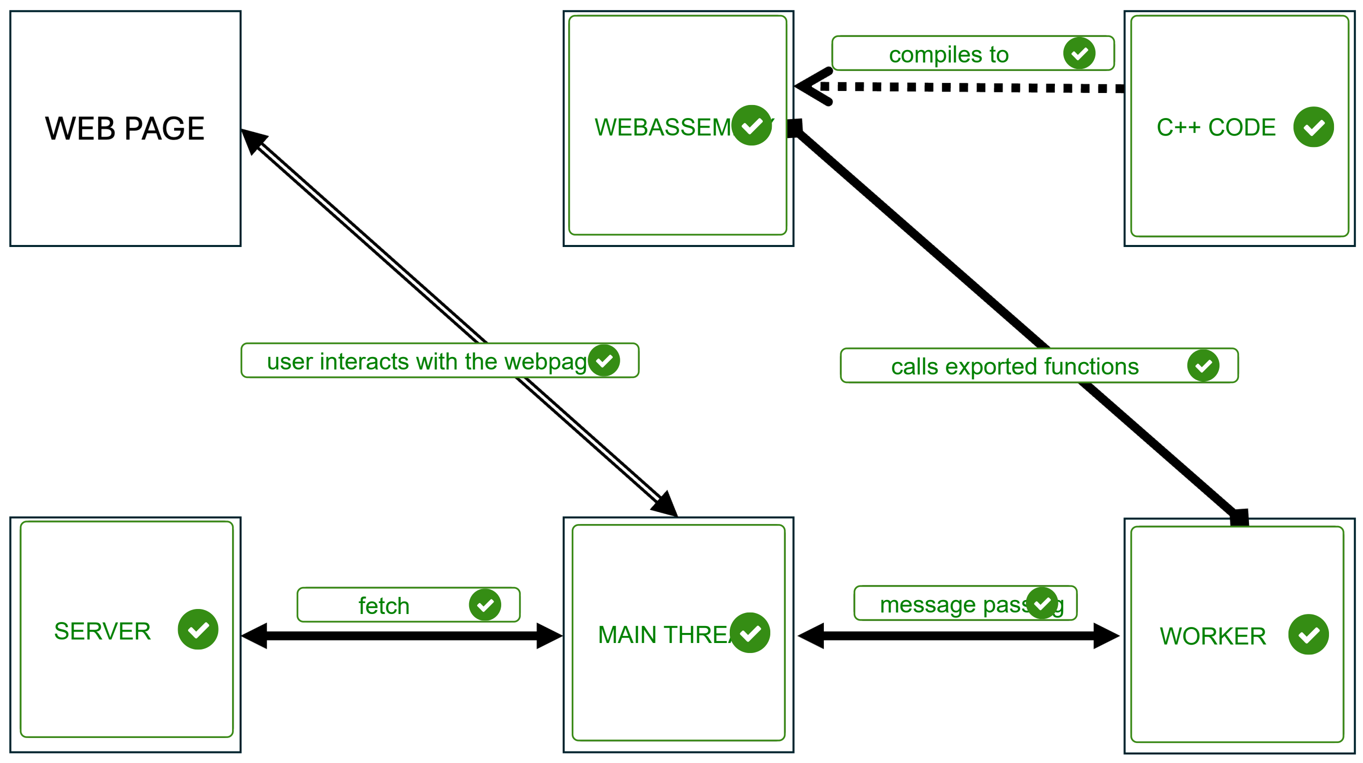
$\{\mathrm{R}(v)\}$ $\{\mathrm{R}(v')\}$ $\{\mathrm{F}(v'), \mathrm{R}(v')\}$

Maks poeng: 1.5

3 Running C++ code in browser

The task description is shown in the PDF for convenience.

Hjelp



Maks poeng: 10

4 Portfolio Sets

Points from Portfolio Sets will be added here.

Maks poeng: 35