Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT392

17 August 2023

Exam supervisors: N. Piterman (piterman@chalmers.se, 073 856 49 10) and G. Schneider (gersch@chalmers.se, 072 974 49 64)

(Exam set by N. Piterman and G. Schneider, based on the course given in Aug-Oct 2022 and Jan-Mar 2023)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes (single or double-sided); English dictionary (no smart phones allowed).

Grading: You can score a maximum of 70 points. Exam grades are: between 28–41 (3), between 42–55 (4), 56 or more (5).

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows: between 40–59 (3), between 60-79 (4), 80 or more (5).

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will receive no points!
- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.
- Answer each question on a new page. Glance through the whole paper first; five questions, numbered Q1 through Q5. Do not spend more time on any question or part than justified by the points it carries.
- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1 (15p). In what follows you will get 5 assertions concerning locks and semaphores. For each assertion, you need to say whether it is correct (true) or not (false). You need to justify your answer in each case (an answer without a justification will not be granted full points).

1 A semaphore with capacity 2 can be implemented by using exactly one lock. That is, everything you can do with a semaphore with capacity 2 can be done with one lock. **(2p)**

2 The pseudocode of the program shown in Fig. 1 guarantees mutual exclusion, it is deadlock-free and it is starvation-free. If you answer *true*, then explain how the semaphore guarantees all those properties. If you answer *false*, explain, which of the three properties are violated and why (you should also provide the right code that guarantees the three properties). **(4p)**

```
int counter; Semaphore sem = new Semaphore(1);
```

thread t

```
int c,d;

1        sem.down();
2        c = counter - 1;
3        d = c * 2;
4        counter = c + d + 1;
5        sem.up();
```

thread u

```
int c;

         sem.up();              6
         c = counter + 1;       7
         counter = c - 1;       8
         sem.down();            9
```

Figure 1: Q1-2: Pseudocode of program `someCounting`.

3 In the correct version of the pseudocode of the program shown in Fig. 1, the semaphore could be replaced by a lock. If you answer *true*, then explain how this is done (provide the replacements to be done in the program). If you answer *false*, explain why this cannot be done. NOTE: Your answer should be based on the *correct* version of the program, meaning that if you answered *true* in the previous question then you should use the original program; if you answered *false* then you should use your corrected version. **(3p)**

4 The capacity of a Java semaphore puts a limit on the number of times one may call `up()`. **(2p)**

5 The pseudocode of the program in Fig. 2 shows the use of two semaphores on two threads. The programmer wanted to implement a barrier. That is, ensure deadlock-freedom and that

`thread t` cannot execute `code #2` unless `thread u` has finished executing `code #3` (and similarly for the other thread: `thread u` cannot execute `code #4` unless `thread t` has finished executing `code #1`). Does the program guarantee those properties? If you answer *true* explain how the properties are guaranteed. If you answer *false* explain why this is not the case and provide the right code so the properties are guaranteed (you should use exactly 2 semaphores). NOTE: You can assume that all the pieces of code are before and after the use of the semaphores make progress, that is, there is no starvation nor deadlocks occurring in those parts of the program. **(4p)**

```
Semaphore done0 = new Semaphore(0), done1 = new Semaphore(1);
```

	thread t		thread u	
1	`// some code #1`		`// some code #3`	5
2	`done0.down();`		`done1.up();`	6
3	`done0.up();`		`done1.down();`	7
4	`// some code #2`		`// some code #4`	8

Figure 2: Q1-5: Pseudocode of program `someSemaphore`.

Q2 (14p). (Part a). (3p)

Here is the implementation of the `add` function of a Sequential Set data structure. The set stores items in increasing key order. Recall that the find function returns a pair of nodes *pred* and *curr* such that $pred.key() < node.key() \leq curr.key()$.

```
public boolean add(T item) {
  Node<T> node = new Node(item);
  Node<T> pred = null, curr = null;
  (pred, curr) = find(head, node.key());
  if (curr.key() == node.key())
    return false;  // item already in set
  else { // insert node between pred and curr
    node.setNext(curr);
    pred.setNext(node);
    return true;
  }
}
```

Can this implementation be used by multiple threads simultaneously (thread safe)? If it is, explain why. If it is not, give an example of how things could go wrong.

(Part b). (4p)

Here are parts of the implementation of a set with lazy node removal, which reduces the amount of locking required.

```
class ValidatedNode<T> extends ReadWriteNode<T>
{
  private volatile boolean valid;

  boolean valid() { return valid; }  // is node valid?
  void setValid() { valid = true; }  // mark valid
  void setInvalid() { valid = false; }  // mark invalid
}

public boolean has(T item) {
  // find position without locking
  Node<T> (pred, curr) = find(head, item.key());
  // check validity and item without locking
  return curr.valid() && curr.key() == item.key();
}
```

```
public boolean remove(T item) {
  do {
    Node<T> (pred, curr) = find(head, item.key()); // no locking
    pred.lock(); curr.lock(); // now lock position
    try { // if position still valid, while locking:
      if (valid(pred, curr)) {
        if (curr.key() > item.key())
          return false; // item not in the set
        else { // item in the set at curr: remove it
          curr.setInvalid(); // logical removal
          pred.setNext(curr.next()); // physical removal
          return true;
        }
     }
    } finally { pred.unlock(); curr.unlock(); }// done: unlock
  } while (true); // if not valid: try again!
}
```

Explain the role of the `volatile` keyword in the definition of `ValidateNode`. What is the role it plays in the correctness of the `has` function? Why is locking still required for removal?

(Part c). (4p)

Here is an implementation of the `remove` function of a lock-free version of the Set data structure. The `find` function, is as before.

```
public boolean remove(T item) {
  boolean done;
  do {
    Node<T> (pred, curr) = find(head, item.key());
    if (curr.key() > item.key()) return false; // item not in set
    else
    // try to remove curr by setting pred.next using compareAndSet
      done = pred.next().compareAndSet(pred.next(), curr.next());
  } while (!done); return true;
}
```

Is this implemetation thread safe? If it is, explain why. If it is not, give an example of how things could go wrong.

(Part d). (3p)

What advantages does lock-free programming offer? Under what conditions is it a good idea to use it?

Q3 (12p). The following code shows part of a telephone system implemented in Erlang:

```
ringing(A, B) ->
       receive
            {A, on_hook} ->
                 A ! {stop_tone, ring},
                 B ! terminate,
                 idle();
            {B, answered} ->
                  A ! {stop_tone, ring},
                  switch ! {connect, A, B},
                  conversation(A, B)
       end.
```

The provided Erlang program defines the function `ringing/2` that handles the ringing phase of a telephone call between two parties, represented by processes A and B. This is what the program does:

1 The function `ringing/2` expects two parameters: A and B, i.e., the processes representing the parties involved in the call.
2 The receive statement is used to listen for incoming messages sent to the process running `ringing/2`.
3 If the received message matches the tuple `{A, on_hook}`, it means that process A has gone on-hook (hung up) before process B answered. In this case:
   i. Process A is sent a message `{stop_tone, ring}` to inform it that the ringing tone should stop;
   ii. Process B is sent a message `terminate` to inform it that the call has ended;
   iii. The function `idle/0` (not defined in the provided code) is called. This function handles the idle state of the server.
4 If the received message matches the tuple `{B, answered}`, it means that process B has answered the call. In this case:
   i. The process A is sent a message `{stop_tone, ring}` to inform it that the ringing tone should stop;
   ii. A process named `switch` handles the telephone switching system and in particular can receive a message `{connect, A, B}` to establish a connection between A and B. This message triggers further actions in the system to set up the call.
   iii. The function `conversation/2` (not defined in the provided code) is called with A and B as arguments. This function handles the conversation phase between the two parties.

(Part a). (3p)

Extend the code such that the function `ringing/2` can receive a message matching `{A, timeout}`, in which case the connection is not established due to a timeout. That is: i) A should get a message that the ringing tone should stop; ii) B should be informed that the call has ended; iii) the server should be idle.

(Part b). (5p)

Assume that the code is extended with the reception of a message for `{A, timeout}`, as in Part a), but that in this case process A should wait for 2000 time units and then call again automatically. In order to do so, a programmer extended the function `ringing/2` with the following additional message in `receive`:

```
{A, timeout} ->
     A ! {stop_tone, ring},
     B ! terminate,
     timer:sleep(2000),
     ringing(A, B);
```

Is this correct? If you answer positively then explain why it works. If you answer that it is not correct, then explain why this is wrong and suggest how this should be done (you do not need to provide code on how this should be done, but just explain the idea on how you would add the additional functionality).

(Part c). (4p)

The `ringing/2` function does not seem to be symmetric: it only handles calls from the first parameter A to the second parameter B. Is this general enough? Would you need to change the code of the function so we can also handle calls from the process being instantiated by B to the one instantiated by A?

Q4 (14p). The four necessary conditions for a deadlock are:

- Mutual exclusion - threads may have exclusive access to shared resources.
- Hold and wait - a thread may request one resource while holding another one.
- No preemption - resources cannot forcibly be released from threads that hold them.
- Circular wait - two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.

Here is an implementation of a protocol for the *dining philosophers problem*. There are n philosophers and n forks that are implemented by locks. Every philosopher has an identifier in the range $1, \ldots, n$, and the `left_fork` of philosopher i is fork i and the `right_fork` of philosopher i is fork $(i \ mod \ n) + 1$.

```
entry  () {
  left_fork.acquire(); // pick up left fork
  right_fork.acquire();// pick up right fork
}
critical section { eat(); }
exit   () {
left_fork.release(); // release left fork
right_fork.release();// release right fork
}
```

(Part a). (4p)

Explain how this program satisifes all four necessary conditions for a deadlock.

(Part b). (4p)

The following solution to the dining philosophers problem adds one more lock. The `eat()` and `release()` functions are as before.

```
entry  () {
  global_lock.acquire();
  left_fork.acquire(); // pick up left fork
  right_fork.acquire();// pick up right fork
  global_lock.release();
}
```

Explain how this solution ensures that there is no circular waiting.

(Part c). (2p)

Does this solution work? That is, if all locks are fair and philosophers eat and release fairly, will the protocol ensure mutual exclusion of eating by adjacent philosophers and lack of starvation for all philosophers?

(Part d). (4p)

Here is the implementation of the `fork` and the functions `get_fork` and `put_fork` from the Erlang solution to dining philosophers that was studied in class.

```
% a fork not held by anyone
fork() ->
  receive
    {get, From, Ref} -> From ! {ack, Ref},
                        fork(From) % fork held
  end.

% a fork held by Owner
fork(Owner) ->
  receive
    {put, Owner, _Ref} -> fork()  % fork not held
  end.

get_fork(Fork) ->
  Ref = make_ref(),
  Fork ! {get, self(), Ref},
  receive {ack, Ref} -> ack end.

put_fork(Fork) ->
  Ref = make_ref(),
  Fork ! {put, self(), Ref}.
```

Suggest an implementation of `get_fork_preempt` that will allow to preempt a process waiting for a fork by the following philosopher:

```
philosopher(Forks) ->
  think(),
  get_fork(Forks#forks.left),  % pick up left fork
  get_fork_preempt(Forks#forks.left,Forks#forks.right), % pick up right
  %fork
  eat(),
  put_fork(Forks#forks.left),  % put down left fork
```

```
put_fork(Forks#forks.right), % put down right fork
philosopher(Forks).
```

Explain how this function would be used.

Partial marks will be given for implementations that require further changes to `philosopher` or `fork` (explain what changes would be required, no need to implement them). There is no need to create changes that apply the preemption mechanism on other philosophers.

Q5 (15p). In what follows you have 5 subquestions (Parts a to e) concerning different topics seen in the course. Each part a multiple-choice question. The grading for each question is as follows:

- For a right answer you will get 3 points;
- If you do not answer the question, you will get 0 points;
- If you answer wrongly, you will get -1 (a negative point).

The total amount of points will be done summing all the points for each part. So, if you answer correctly Part a, incorrectly Part b, and do not answer any of the rest, you will get 2 points (3 points for Part a minus 1 point for Part b, and 0 for the rest). Note that no negative points for the whole question (Q5) will be given (the minimum number of points you may get for the whole question is 0). That is, if you do answer wrongly in each part, you will not get -5 but 0.

(Part a). (3p)

You can see in Fig. 3 for implementations of the function `up()` of a semaphore using `notify()` and the internal counter of the semaphore, `count`. The implementation of `down()` is shown in Fig. 4. Which one of the four implementations of `up()` does NOT work?

a) `up1()`

b) `up2()`

c) `up3()`

d) `up4()`

```
public synchronized void up1() {
    ++count;
    notify();
}


public synchronized void up2() {
    notify();
    ++count;
}
```

```
28  public void up3() {
29      notify();
30      synchronized (this) {
31          ++count;
32      }
33  }
34
35  public void up4() {
36      synchronized (this) {
37          ++count;
38      }
39      notify();
40  }
```

Figure 3: Q5 - Part a: Four semaphore implementations of up().

(Part b). (3p)

This question concerns the use of monitors with different signalling policies. In Fig. 5 you see the pseudocode of a monitor program that prints the value of `counter`. Threads may execute `inc()` and `print()`

```
public synchronized void down() throws InterruptedException {
    while (count == 0) { wait(); }
    count = count - 1;
}
```

Figure 4: Q5 - Part a: Semaphore implementation of down().

```
monitor class PrintCounter {
   private int counter = 0;
   private Condition isOne = new Condition();

   public void print() {
        while (counter != 1) isOne.wait();
        System.out.println(counter);
   }

   public void inc() {
         counter += 1;
         if (counter == 1) isOne.signal();
   }
}
```

Figure 5: Q5 - Part b: Pseudocode of program `PrintCounter`.

in every order and as many times as they want. What are the possible printed values when calling method `print()`?

a) It may print "1" or it may block forever

b) It may print any number equal or bigger than "1"

c) It never prints "1"

d) It always prints "1" if the monitor uses a *signal and continue* discipline

e) It always blocks if the monitor uses a *signal and wait* discipline

(Part c). (3p)

This question is about parallelisation. In Fig. 6 you get a parallel version of a program that computes the multiplication of numbers from m to n.

Note 1: The function has two special cases: it gives 1 if $m > n$, and it gives m when $m == n$).

Note 2: the division operation ("/") truncates the decimal part (e.g., 1/2 gives 0).

```
class ParallelMul extends RecursiveTask<Integer> {
     int m, n;

     protected Integer compute() {
         if (m > n) return 1;
         if (m == n) return m;
         int mid = m + (n-m)/2;                 // mid point
         ParallelMul lower = new ParallelMul(m, mid);
         ParallelMul upper = new ParallelMul(mid+1, n);
         lower.fork();
         upper.fork();
         return lower.join() * upper.join();
    }
}
```

Figure 6: Q5 - Part c: Java program `ParallelMul`.

The program is run to compute the product of integers from $m = 1$ to $n = k$ (with $k > 1$).

What is the maximal number of cores that would run this code effectively? That is, adding more cores will not speed up the computation.

a) 1, there practically is no parallelism

b) 2^k (that is, 2 to the power of k)

c) k

d) $k!$ (that is, the factorial of k)

e) k^2 (that is, $k * k$)

(Part d). (3p)

According to Amdahl's law, if the fraction p of a program can be parallelized, then, the maximum speedup that can be achieved by n processes is $\frac{1}{(1-p)+\frac{p}{n}}$. You have a program where 10% of the program must be done sequentially and 90% of the program can be parallelized. What is the maximum speedup that you can achieve given unlimited resources (i.e., increase the number of processes as you wish).

a) One cannot achieve speedup at all.

b) With a very large number of processes, one can achieve every wanted speedup.

c) The program cannot run more than 10 times faster.

d) The program cannot run more than 5 times faster.

(Part e). (3p)

Which of the following programs does not have data races? In all cases, $t1$ and $t2$ are threads executing at the same time sharing the variables at the top. The programs are numbered 1, 2, and 3 from left to right.

```
boolean x=false, y=false;

t1 {
    if (x)
        y=true;
}

t2 {
    if (y)
        x=true;
}
```

```
int cnt = 0;

t1 {
    lock.lock();
    cnt++;
    cnt++;
    lock.unlock();
}

t2 {
    cnt++;
}
```

```
volatile boolean flag = false;
int cnt = 0;

t1 {
    if (flag) {
        cnt++;
    }
}

t2 {
    cnt++;
    flag = true;
}
```

Figure 7: Q5 - Part e: Programs with or without data races.

a) Program 2.

b) Program 3.

c) Program 1 and program 2.

d) Program 1 and program 3.

Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT392

17 August 2023

Exam supervisors: N. Piterman (piterman@chalmers.se, 073 856 49 10) and G. Schneider (gersch@chalmers.se, 072 974 49 64)

(Exam set by N. Piterman and G. Schneider, based on the course given in Aug-Oct 2022 and Jan-Mar 2023)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes (single or double-sided); English dictionary (no smart phones allowed).

Grading: You can score a maximum of 70 points. Exam grades are: between 28–41 (3), between 42–55 (4), 56 or more (5).

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows: between 40–59 (3), between 60-79 (4), 80 or more (5).

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will receive no points!
- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.
- Answer each question on a new page. Glance through the whole paper first; five questions, numbered Q1 through Q5. Do not spend more time on any question or part than justified by the points it carries.
- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1 (15p). In what follows you will get 5 assertions concerning locks and semaphores. For each assertion, you need to say whether it is correct (true) or not (false). You need to justify your answer in each case (an answer without a justification will not be granted full points).

1 A semaphore with capacity 2 can be implemented by using exactly one lock. That is, everything you can do with a semaphore with capacity 2 can be done with one lock. **(2p)**

Answer: False. A semaphore is a more general synchronisation structure than a lock. In particular, a semaphore with capacity 2 can allow certain synchronisation behaviours not allowed if you use one lock.

2 The pseudocode of the program shown in Fig. 1 guarantees mutual exclusion, it is deadlock-free and it is starvation-free. If you answer *true*, then explain how the semaphore guarantees all those properties. If you answer *false*, explain, which of the three properties are violated and why (you should also provide the right code that guarantees the three properties). **(4p)**

```
int counter; Semaphore sem = new Semaphore(1);
```

	thread t		thread u	
	`int c,d;`		`int c;`	
1	`sem.down();`		`sem.up();`	6
2	`c = counter - 1;`		`c = counter + 1;`	7
3	`d = c * 2;`		`counter = c - 1;`	8
4	`counter = c + d + 1;`		`sem.down();`	9
5	`sem.up();`			

Figure 1: Q1-2: Pseudocode of program `someCounting`.

Answer: False. It is not mutual exclusive (e.g., if thread u calls `sem.up()` *first, then thread u can call* `sem.down()` *and both will be in the critical section). It is deadlock-free and starvation-free (it is never the case that both get stuck because they cannot execute* `sem.up()`*). The right code is to exchange lines 6 and 9 (i.e., thread u should execute* `sem.down()` *at the beginning and* `sem.up()` *at the end).*

3 In the correct version of the pseudocode of the program shown in Fig. 1, the semaphore could be replaced by a lock. If you answer *true*, then explain how this is done (provide the replacements to be done in the program). If you answer *false*, explain why this

cannot be done. NOTE: Your answer should be based on the *correct* version of the program, meaning that if you answered *true* in the previous question then you should use the original program; if you answered *false* then you should use your corrected version. **(3p)**

Answer: True. A semaphore with capacity 1 used in the way it is used in the correct version of the program acts as a lock. The replacements to be done (in the correct coce) are: a) replace `sem.down()` *by* `lock()`*; b) replace* `sem.up()` *by* `unlock()`*.*

4 The capacity of a Java semaphore puts a limit on the number of times one may call `up()`. **(2p)**

Answer: False. The capacity is just an initial value of the internal counter of the semaphore and there is no limit on how many times one may call `up()`*.*

5 The pseudocode of the program in Fig. 2 shows the use of two semaphores on two threads. The programmer wanted to implement a barrier. That is, ensure deadlock-freedom and that `thread t` cannot execute `code #2` unless `thread u` has finished executing `code #3` (and similarly for the other thread: `thread u` cannot execute `code #4` unless `thread t` has finished executing `code #1`). Does the program guarantee those properties? If you answer *true* explain how the properties are guaranteed. If you answer *false* explain why this is not the case and provide the right code so the properties are guaranteed (you should use exactly 2 semaphores). NOTE: You can assume that all the pieces of code are before and after the use of the semaphores make progress, that is, there is no starvation nor deadlocks occurring in those parts of the program. **(4p)**

```
Semaphore done0 = new Semaphore(0), done1 = new Semaphore(1);
```

	thread t		thread u	
1	`// some code #1`		`// some code #3`	5
2	`done0.down();`		`done1.up();`	6
3	`done0.up();`		`done1.down();`	7
4	`// some code #2`		`// some code #4`	8

Figure 2: Q1-5: Pseudocode of program `someSemaphore`.

Answer: False. It is deadlock-free since none of the threads blocks the other from proceeding but it is not starvation-free: `thread t` *is blocked after executing* `code #1` *given that cannot call* `done0.down()`

since the semaphore has been initialised to 0. Also, the program doesn't guarantee the properties stated: `thread u` *can execute* `code #4` *after calling* `done1.down()` *without having to wait for* `thread t` *finishing the execution of* `code #1` *since they each handle a separate semaphore independently of each other. The correct code is shown in Fig. 3.*

```
Semaphore done0 = new Semaphore(0), done1 = new Semaphore(0);
```

	thread t		thread u	
1	`// some code #1`		`// some code #3`	5
2	`done0.up();`		`done1.up();`	6
3	`done1.down();`		`done0.down();`	7
4	`// some code #2`		`// some code #4`	8

Figure 3: Q1-5: Corrected pseudocode of program `someSemaphore`.

Q2 (14p). (Part a). (3p)

Here is the implementation of the `add` function of a Sequential Set data structure. The set stores items in increasing key order. Recall that the find function returns a pair of nodes *pred* and *curr* such that $pred.key() < node.key() \leq curr.key()$.

```
public boolean add(T item) {
  Node<T> node = new Node(item);
  Node<T> pred = null, curr = null;
  (pred, curr) = find(head, node.key());
  if (curr.key() == node.key())
    return false;  // item already in set
  else { // insert node between pred and curr
    node.setNext(curr);
    pred.setNext(node);
    return true;
  }
}
```

Can this implementation be used by multiple threads simultaneously (thread safe)? If it is, explain why. If it is not, give an example of how things could go wrong.

Answer: The implementation is not *thread safe. For example, if the list contains elements with keys 1 and 4, and two threads try to add 2 and 3 simultaneously, one addition could over-write the other. In particular, if both execute the line* `pred.setNext(node)` *one after the other, the second will over-write the first.*

(Part b). (4p)

Here are parts of the implementation of a set with lazy node removal, which reduces the amount of locking required.

```
class ValidatedNode<T> extends ReadWriteNode<T>
{
  private volatile boolean valid;

  boolean valid() { return valid; }  // is node valid?
  void setValid() { valid = true; }  // mark valid
  void setInvalid() { valid = false; }  // mark invalid
}

public boolean has(T item) {
  // find position without locking
```

```
  Node<T> (pred, curr) = find(head, item.key());
  // check validity and item without locking
  return curr.valid() && curr.key() == item.key();
}

public boolean remove(T item) {
  do {
    Node<T> (pred, curr) = find(head, item.key()); // no locking
    pred.lock(); curr.lock(); // now lock position
    try { // if position still valid, while locking:
      if (valid(pred, curr)) {
        if (curr.key() > item.key())
          return false; // item not in the set
        else { // item in the set at curr: remove it
          curr.setInvalid(); // logical removal
          pred.setNext(curr.next()); // physical removal
          return true;
        }
     }
    } finally { pred.unlock(); curr.unlock(); }// done: unlock
  } while (true); // if not valid: try again!
}
```

Explain the role of the `volatile` keyword in the definition of `ValidateNode`. What is the role it plays in the correctness of the `has` function? Why is locking still required for removal?

Answer: The volatile keyword means that changes to the field are visible immediately to other threads that may access the same field.

It means that in this implementation of has, the value of curr.valid() is the latest value and that the node has not been invalidated by other threads while the thread running has was locating the current position.

For removal, locking is still required because, for example, otherwise several threads could remove the same node successfully.

(Part c). (4p)

Here is an implementation of the `remove` function of a lock-free version of the Set data structure. The `find` function, is as before.

```
public boolean remove(T item) {
  boolean done;
  do {
    Node<T> (pred, curr) = find(head, item.key());
    if (curr.key() > item.key()) return false; // item not in set
```

```
    else
    // try to remove curr by setting pred.next using compareAndSet
      done = pred.next().compareAndSet(pred.next(), curr.next());
  } while (!done); return true;
}
```

Is this implemetation thread safe? If it is, explain why. If it is not, give an example of how things could go wrong.

Answer: The implementation is not *thread safe. If pred is being removed by some other thread then the removal is ignored.*

(Part d). (3p)

What advantages does lock-free programming offer? Under what conditions is it a good idea to use it?

Answer: Lock-free programming may reduce the amount of time that is spent on locking and hence accelerate the execution of a multi-threaded implementation. However, it requires to repeat some operations. In a situation where there is a lot of competition it may actually cause a delay due to repetition.

It should be used when there is little competition (and the case of this Set, little changes compared to many searches).

Q3 (12p). The following code shows part of a telephone system implemented in Erlang:

```
ringing(A, B) ->
       receive
            {A, on_hook} ->
                 A ! {stop_tone, ring},
                 B ! terminate,
                 idle();
            {B, answered} ->
                  A ! {stop_tone, ring},
                  switch ! {connect, A, B},
                  conversation(A, B)
       end.
```

The provided Erlang program defines the function `ringing/2` that handles the ringing phase of a telephone call between two parties, represented by processes A and B. This is what the program does:

1 The function `ringing/2` expects two parameters: A and B, i.e., the processes representing the parties involved in the call.
2 The receive statement is used to listen for incoming messages sent to the process running `ringing/2`.
3 If the received message matches the tuple `{A, on_hook}`, it means that process A has gone on-hook (hung up) before process B answered. In this case:
   i. Process A is sent a message `{stop_tone, ring}` to inform it that the ringing tone should stop;
   ii. Process B is sent a message `terminate` to inform it that the call has ended;
   iii. The function `idle/0` (not defined in the provided code) is called. This function handles the idle state of the server.
4 If the received message matches the tuple `{B, answered}`, it means that process B has answered the call. In this case:
   i. The process A is sent a message `{stop_tone, ring}` to inform it that the ringing tone should stop;
   ii. A process named `switch` handles the telephone switching system and in particular can receive a message `{connect, A, B}` to establish a connection between A and B. This message triggers further actions in the system to set up the call.
   iii. The function `conversation/2` (not defined in the provided code) is called with A and B as arguments. This function handles the conversation phase between the two parties.

(Part a). (3p)

Extend the code such that the function `ringing/2` can receive a message matching `{A, timeout}`, in which case the connection is not established due to a timeout. That is: i) A should get a message that the ringing tone should stop; ii) B should be informed that the call has ended; iii) the server should be idle.

Answer: The code is extended with the following under `receive`*:*

```
{A, timeout} ->
     A ! {stop_tone, ring},
     B ! terminate,
     idle();
```

(Part b). (5p)

Assume that the code is extended with the reception of a message for `{A, timeout}`, as in Part a), but that in this case process A should wait for 2000 time units and then call again automatically. In order to do so, a programmer extended the function `ringing/2` with the following additional message in `receive`:

```
{A, timeout} ->
     A ! {stop_tone, ring},
     B ! terminate,
     timer:sleep(2000),
     ringing(A, B);
```

Is this correct? If you answer positively then explain why it works. If you answer that it is not correct, then explain why this is wrong and suggest how this should be done (you do not need to provide code on how this should be done, but just explain the idea on how you would add the additional functionality).

Answer: The code is wrong since the timer should not be applicable to the server defining `ringing/2` *but to the process making the call. Also, the recursive call* `ringing(A, B)` *will keep waiting for messages matching one of the three options, not producing the desired effect that process A will call again. A correct implementation would add the timer to process A, who will then act as "normal" by calling again after the timeout.*

(Part c). (4p)

The `ringing/2` function does not seem to be symmetric: it only handles calls from the first parameter A to the second parameter B. Is this general enough? Would you need to change the code of the function

so we can also handle calls from the process being instantiated by B to the one instantiated by A?

Answer: Yes, the code is general: you just need to call the function interchanging the two parameters in order to make the second process to call the first one.

Q4 (14p). The four necessary conditions for a deadlock are:

- Mutual exclusion - threads may have exclusive access to shared resources.
- Hold and wait - a thread may request one resource while holding another one.
- No preemption - resources cannot forcibly be released from threads that hold them.
- Circular wait - two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.

Here is an implementation of a protocol for the *dining philosophers problem*. There are n philosophers and n forks that are implemented by locks. Every philosopher has an identifier in the range $1, \ldots, n$, and the `left_fork` of philosopher i is fork i and the `right_fork` of philosopher i is fork $(i \ mod \ n) + 1$.

```
entry  () {
  left_fork.acquire(); // pick up left fork
  right_fork.acquire();// pick up right fork
}
critical section { eat(); }
exit   () {
left_fork.release(); // release left fork
right_fork.release();// release right fork
}
```

(Part a). (4p)

Explain how this program satisifes all four necessary conditions for a deadlock.

Answer:

- *Mutual exclusion - each lock is held by at most one philosopher at a time.*
- *Hold and Wait - each philosopher requires two forks to eat. They take one and try to acquire the second.*
- *No preemption - the only way to release the locks is by executing the exit protocol.*
- *Circular wait - in a scenario where all philosophers reach for their left fork one by one we have the all philosophers are waiting for a resouce that the next thread in the chain is holding.*

(Part b). (4p)

The following solution to the dining philosophers problem adds one more lock. The `eat()` and `release()` functions are as before.

```
entry  () {
  global_lock.acquire();
  left_fork.acquire(); // pick up left fork
  right_fork.acquire();// pick up right fork
  global_lock.release();
}
```

Explain how this solution ensures that there is no circular waiting.

Answer: The main lock ensures that no two philosophers try to ackquire locks at the same time. Furthermore, the only way to release the main lock is when acquiring both forks. If some process is locking the main lock and waiting for one of their forks, by implication, the philosopher it is waiting for is not waiting for anything becuase it released the main lock.

(Part c). (2p)

Does this solution work? That is, if all locks are fair and philosophers eat and release fairly, will the protocol ensure mutual exclusion of eating by adjacent philosophers and lack of starvation for all philosophers?

Answer: Yes, the solution works. There cannot be a deadlock as explained above. If the main lock is fair, then lack of starvation is ensured by fairness of the lock.

(Part d). (4p)

Here is the implementation of the `fork` and the functions `get_fork` and `put_fork` from the Erlang solution to dining philosophers that was studied in class.

```
% a fork not held by anyone
fork() ->
  receive
    {get, From, Ref} -> From ! {ack, Ref},
                        fork(From) % fork held
  end.

% a fork held by Owner
fork(Owner) ->
  receive
    {put, Owner, _Ref} -> fork()  % fork not held
```

```
  end.

get_fork(Fork) ->
  Ref = make_ref(),
  Fork ! {get, self(), Ref},
  receive {ack, Ref} -> ack end.

put_fork(Fork) ->
  Ref = make_ref(),
  Fork ! {put, self(), Ref}.
```

Suggest an implementation of `get_fork_preempt` that will allow to preempt a process waiting for a fork by the following philosopher:

```
philosopher(Forks) ->
  think(),
  get_fork(Forks#forks.left),  % pick up left fork
  get_fork_preempt(Forks#forks.left,Forks#forks.right), % pick up right
  %fork
  eat(),
  put_fork(Forks#forks.left),  % put down left fork
  put_fork(Forks#forks.right), % put down right fork
  philosopher(Forks).
```

Explain how this function would be used.

Partial marks will be given for implementations that require further changes to `philosopher` or `fork` (explain what changes would be required, no need to implement them). There is no need to create changes that apply the preemption mechanism on other philosophers.

Answer:

```
get_fork_preempt(Heldfork, Fork) ->
  Ref = make_ref(),
  Fork ! {get, self(), Ref},
  receive
    {ack, Ref} -> ack;
    preempt -> Heldfork ! { put, self(), Ref},
               preemepted(Fork,Ref,Heldfork)
  end.

preempted(Forkwait,Ref,Nextfork)
  receive
    {ack, Ref} -> get_fork_preempt(Forkwait,Nextfork)
  end.
```

With this implementation no further changes are required.

Q5 (15p). In what follows you have 5 subquestions (Parts a to e) concerning different topics seen in the course. Each part a multiple-choice question. The grading for each question is as follows:

- For a right answer you will get 3 points;
- If you do not answer the question, you will get 0 points;
- If you answer wrongly, you will get -1 (a negative point).

The total amount of points will be done summing all the points for each part. So, if you answer correctly Part a, incorrectly Part b, and do not answer any of the rest, you will get 2 points (3 points for Part a minus 1 point for Part b, and 0 for the rest). Note that no negative points for the whole question (Q5) will be given (the minimum number of points you may get for the whole question is 0). That is, if you do answer wrongly in each part, you will not get -5 but 0.

(Part a). (3p)

You can see in Fig. 4 for implementations of the function `up()` of a semaphore using `notify()` and the internal counter of the semaphore, `count`. The implementation of `down()` is shown in Fig. 5. Which one of the four implementations of `up()` does NOT work?

a) `up1()`

b) `up2()`

c) `up3()`

d) `up4()`

```
public synchronized void up1() {
    ++count;
    notify();
}


public synchronized void up2() {
    notify();
    ++count;
}
```

```
28    public void up3() {
29        notify();
30        synchronized (this) {
31            ++count;
32        }
33    }
34
35    public void up4() {
36        synchronized (this) {
37            ++count;
38        }
39        notify();
40    }
```

Figure 4: Q5 - Part a: Four semaphore implementations of up().

Answer: Option c). Reason: `up3()` *does not work because the notify() might wake a thread that will go ahead, take the lock, see that count is* 0 *and then go back to waiting. The thread calling* `up3()` *will then increase the counter but the waiting thread will not be woken up leading to a deadlock.*

```
public synchronized void down() throws InterruptedException {
    while (count == 0) { wait(); }
    count = count - 1;
}
```

Figure 5: Q5 - Part a: Semaphore implementation of down().

(Part b). (3p)

This question concerns the use of monitors with different signalling policies. In Fig. 6 you see the pseudocode of a monitor program that prints the value of `counter`. Threads may execute `inc()` and `print()` in every order and as many times as they want. What are the possible printed values when calling method `print()`?

```
monitor class PrintCounter {
   private int counter = 0;
   private Condition isOne = new Condition();

   public void print() {
        while (counter != 1) isOne.wait();
        System.out.println(counter);
   }

   public void inc() {
         counter += 1;
         if (counter == 1) isOne.signal();
   }
}
```

Figure 6: Q5 - Part b: Pseudocode of program `PrintCounter`.

a) It may print "1" or it may block forever

b) It may print any number equal or bigger than "1"

c) It never prints "1"

d) It always prints "1" if the monitor uses a *signal and continue* discipline

e) It always blocks if the monitor uses a *signal and wait* discipline

Answer: Option a). Reason: the only way to print is by not executing the while when the counter is 1 in which case it will print 1. The other possibility is that some of the threads waiting in the `wait` *condition wakes up and then proceed to print, but this would only happen when a* `signal` *is executed, which may only happen once (given the condition*

`counter==1`*). Thus, all the process waiting to wake up will be blocked as the signalling will happen only once.*

(Part c). (3p)

This question is about parallelisation. In Fig. 7 you get a parallel version of a program that computes the multiplication of numbers from m to n.

Note 1: The function has two special cases: it gives 1 if $m > n$, and it gives m when $m == n$).

Note 2: the division operation ("/") truncates the decimal part (e.g., $1/2$ gives 0).

```
class ParallelMul extends RecursiveTask<Integer> {
     int m, n;

     protected Integer compute() {
         if (m > n) return 1;
         if (m == n) return m;
         int mid = m + (n-m)/2;                    // mid point
         ParallelMul lower = new ParallelMul(m, mid);
         ParallelMul upper = new ParallelMul(mid+1, n);
         lower.fork();
         upper.fork();
         return lower.join() * upper.join();
     }
}
```

Figure 7: Q5 - Part c: Java program `ParallelMul`.

The program is run to compute the product of integers from $m = 1$ to $n = k$ (with $k > 1$).

What is the maximal number of cores that would run this code effectively? That is, adding more cores will not speed up the computation.

a) 1, there practically is no parallelism

b) 2^k (that is, 2 to the power of k)

c) k

d) $k!$ (that is, the factorial of k)

e) k^2 (that is, $k * k$)

Answer: Option c). Reason: the program forks till reaching the base case in which case there are k parallel threads that will then be joined performed the multiplication. All other threads are waiting.

(Part d). (3p)

According to Amdahl's law, if the fraction p of a program can be parallelized, then, the maximum speedup that can be achieved by n processes is $\frac{1}{(1-p)+\frac{p}{n}}$. You have a program where 10% of the program must be done sequentially and 90% of the program can be parallelized. What is the maximum speedup that you can achieve given unlimited resources (i.e., increase the number of processes as you wish).

a) One cannot achieve speedup at all.

b) With a very large number of processes, one can achieve every wanted speedup.

c) The program cannot run more than 10 times faster.

d) The program cannot run more than 5 times faster.

Answer: Option c). Reason: The denominator is at least 1/10. So the fraction is always smaller than 10.

(Part e). (3p)

Which of the following programs does not have data races? In all cases, $t1$ and $t2$ are threads executing at the same time sharing the variables at the top. The programs are numbered 1, 2, and 3 from left to right.

```
boolean x=false, y=false;

t1 {
    if (x)
        y=true;
}

t2 {
    if (y)
        x=true;
}
```

```
int cnt = 0;

t1 {
    lock.lock();
    cnt++;
    cnt++;
    lock.unlock();
}

t2 {
    cnt++;
}
```

```
volatile boolean flag = false;
int cnt = 0;

t1 {
    if (flag) {
        cnt++;
    }
}

t2 {
    cnt++;
    flag = true;
}
```

Figure 8: Q5 - Part e: Programs with or without data races.

a) Program 2.

b) Program 3.

c) Program 1 and program 2.

d) Program 1 and program 3.

Answer: Option d). Reason: Program 1 the write are never accessible. So it cannot have a data race. Program 2 has a write in t2 that is not ordered with respect to the writes in t1. Program 3 As flag is volatile, an access to the if in t1 happens only after the flag is set to true in t2. This means that the increment in t1 happens after the increment in t2.