INF214 Concurrent Programming

December 12th, 2018

Please write clearly! Pay special attention to marking question numbers clearly on your answer sheets. Exam support materials: *none*.

Some questions ask for program code. If you do not remember the syntax of whatever language is being used, it is no big deal. Use pseudo-code that makes your intent clear, and explain with additional comments if necessary.

Hvis du vil, du kan svare på norsk i stedet for engelsk.

There are 5 questions altogether and 4 pages.

---

1. Atomic operations and interleavings

Consider the program below, where x is a shared variable, initially 0. P_1 and P_2 are names of the two processes, not part of the program.

co $x \overset{P_1}{=} x + 1 \parallel x \overset{P_2}{=} x - 1$ **oc**

a. What are all the atomic memory operations in the program.

ANSWER:

10 points

P_1 first reads x from the shared memory (call this operation R1), then increments it, writing possibly to a local (non-shared) temporary, then writes the result to x (call this operation W1). The atomic memory operations are R1 and W1. P_2 has analogous atomic memory operations, which we name R2 and W2.

b. List all possible interleavings of these operations. Determine the final value of x for each case. UPDATE: It was clarified during the exam that one can assume sequential consistency.

ANSWER:

10 points

	x		x		x		x		x		x
R1	0	R1	0	R1	0	R2	0	R2	0	R2	0
R2	0	R2	0	W1	1	R1	0	R1	0	W2	-1
W1	1	W2	-1	R2	1	W1	1	W2	-1	R1	-1
W2	-1	W1	1	W2	0	W2	-1	W1	1	W1	0
	-1		1		0		-1		1		0

---

2. Concepts

a. 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.

ANSWER:

3 points

Assume `s` is a semaphore.

- `P(s)` atomically checks if the value of `s` is at least one, and if so decrements the value by one. If not, it blocks until `s` is at least one (and retries the atomic check/decrement).
- `P(v)` increments the value of `s` by one.

a. What is the purpose of a *barrier*?

ANSWER:

3 points

A purpose of a barrier is to make sure that all threads in a given group of threads have reached a particular point in the program before any thread can proceed past that point.

b. Assume the following two codes are executed in separate threads in a C++ program (assuming its *relaxed* memory model). Further assume that `v` and `go` are regular (non-atomic) variables, and that initially `v == 0` and `go == false`.

```
// Thread 1                    // Thread 2
while (!go);                   v = 1;
print(v);                      go = true;
```

Can the program print 0? Why or why not? Is there a *data race* in the program? (If yes, where?)

ANSWER:

4 points

Yes, the program can pring 0. Thread 1 can observe the writes Thread 2 to variables `v` and `go` in any order (because of compiler rewrites the operations of Thread 2 or because the hardware executes them concurrently). In particular, Thread 1 can observe `go == true` and still after that `v == 0`.

There are two data races, on `go` and `v`. Thread 1 and Thread 2 both access these variables concurrently, the accesses of Thread 2 are writes, and there is no synchronization between the accesses.

d. What is a *future*? Describe the three possible states of a future.

ANSWER:

4 points

Future is a placeholder for a result of a computation; a delayed value, a promise that eventually the future will hold a value.

The states of the future are:

- pending (future has no value yet)
- fulfilled (future has a value)
- rejected (future is in an error state, it will never get a value)

Valid state transitions are from pending to fulfilled and from pending to rejected.

Several alternative names for the states are fine.

- fulfilled: resolved, ready
- rejected: error

e. Describe the difference between a *synchronous* and *asynchronous* message passing.

ANSWER:

3 points

In synchronous message passing, the sender process blocks at message send until the receiver process is ready to receive a message. In asynchronous message passing, the sender process does not block but rather can continue, regardless of whether a receiver is ready or not. Practical limitations (reaching the maximum allowed buffer size) may cause the sender to block even in asynchronous message passing.

f. Describe the idea of *Software Transactional Memory* in a few sentences.

ANSWER:

3 points

Software transactional memory (STM) is a way to implement (coarse-grained) atomic blocks. When starting to execute an atomic block, a thread starts a transaction. During the transaction, all memory reads and writes are logged. At the end of the transaction (and during, in some implementations) the read log is compared with the current state of the memory. If the memory is unchanged from when the transaction started, the transaction is committed and the log of writes are written to the memory. If the memory was changed, the logs are cleared and the transaction is restarted. Determining whether memory has been changed is based on a version number stored with every word/object/block in the memory.

STM implementations guarantee that at least one transaction of many that access shared memory locations will succeed.

3. Monitors

In the dining philosophers problem, n philosophers alternate between eating and thinking at random intervals. To eat, a philosopher needs two forks. There are n forks altogether. In the traditional formulation of the problem, each philosopher uses the forks to the left and right of him/her. In this assignment, however, every time a philosopher decides to eat, he/she chooses two forks at random to use for eating, regardless of whether they are currently in use or not. UPDATE: If one or both of the chosen forks are in use, the philosopher must wait until they are free to begin eating.

a. Implement a *monitor* that synchronizes the philosophers' use of forks. Use some pseudo-code that resembles C++/alang or the language used in the book. The monitor should implement the procedures:

```
void get_forks(int fork1, int fork2);
void release_forks(int fork1, int fork2);
```

ANSWER:

12 points

```
class table : monitor {
  private:
    cond fork_released;
    std::vector<bool> forks;
  public:
    table(int n) : forks(n, false) {}

    void get_forks(int fork1, int fork2) {
      SYNC;
      while (forks[fork1] || forks[fork2]) {
        if (forks[fork1]) alang::logl("Waiting for fork ", fork1);
        if (forks[fork2]) alang::logl("Waiting for fork ", fork2);
        wait(fork_released);
      }
      forks[fork1] = true;
      forks[fork2] = true;
    }
    void release_forks(int fork1, int fork2) {
      SYNC;
      forks[fork1] = false;
      forks[fork2] = false;
      alang::logl("Released forks ", fork1, " and ", fork2);
      signal_all(fork_released);
    }
}
```

b. What is the *monitor invariant* of your monitor?

ANSWER:

2 points

`forks.size() == 10` $\wedge$ the number of `true` elements in `forks` is even.

Other answers can be OK too if it is clear that the the student understands what a monitor invariant is.

c. Your implementation should be such that it avoids deadlock. Explain why it does avoid it. (Or if you did not manage to come up with a non-deadlocking implementation, explain how deadlock could arise).

ANSWER:

3 points

Deadlock cannot arise, because the `get_forks` function atomically checks that both desired forks are available and if so reserves them. If both are not available, `get_forks` releases the monitor's intrinsic lock so that other threads can make progress, and then tries again.

d. Discuss briefly the *fairness* of your implementation. (Can a philosopher starve? Is it likely?)

UPDATE: It was clarified during the exam that the implementation can be fair or not, both are fine. One just needs to discuss its fairness.

ANSWER:

3 points

The conditions for a philosopher to be able to eat become true infinitely often, but do not necessarily stay true once the become true. Fairness would thus require a strongly fair scheduler, so in principle the implementation is not fair in practical schedulers (which are typically weakly fair) and with an adversarial scheduler a philosopher could starve. In practice it would be very unlikely for a philosopher to starve; practical schedulers pick processes for execution in non-adverserial manner.

A skeleton program (for $n = 10$) that shows a use of the monitor is given. Alang's `prandom(a, b)` gives a random integer in the closed interval $[a, b]$ and `sleep_random(t)` makes a process sleep for `t` milliseconds.

```
#include "alang.hpp"
class table : monitor {

  // Your code
};

int main() {
  const int n = 10;
  table tbl(n);
  {
    processes ps;
    for (int i = 0; i<n; ++i) {
      ps += [&tbl]{
        while (true) {
          alang::sleep_random(100);
          auto fork1 = alang::prandom(0, n-1);
          auto fork2 = (fork1 + alang::prandom(0, n-2)) % 10;
            // pick a random fork different from fork1

          tbl.get_forks(fork1, fork2);

          alang::sleep_random(100);
          tbl.release_forks(fork1, fork2);
        }
```

```
        };
      }
    }
}
```

---

4. Coroutines

a. Two *transfer of control* events take place during the lifetime of a *subroutine* (*call* and *return*), whereas *five* transfer of control events can take place during the lifetime of a *coroutine*, some of them more than once.

Describe the five transfer of control events of coroutines. Include explanations of what happens to *activation records* of coroutines at these events. You can assume a stack-based language like C or C++.

ANSWER:

10 points

- *Call*: coroutine call is similar to a function call. An activation record is pushed onto the stack. On some coroutine implementations, a call merely *primes* the coroutine, and immediately *suspends*.
- *Suspend*: The execution of the coroutine is suspeneded. The activation record is saved on the heap, along with the program counter.
- *Resume*: The execution of the coroutine is resumed. The activation record is copied from the heap and pushed on the stack. Program counter is resumed to the saved value.
- *Destroy*: The activation record that has been saved is deallocated (without resuming execution). The coroutine can no longer be suspended.
- *Return*: similar to a return of a function. Activation record is popped of the stack.

b. Consider a graphical user interface that contains a single element `box`. The following piece of JavaScript code sets up the event handlers for being able to *drag and drop* the `box` element with a mouse. A drag and drop sequence starts by a `mousedown` event on the `box` element, followed by any number of `mousemove` events anywhere within `window`, and ends with a `mouseup` event enywhere within `window`.

```
let box = document.getElementById("box");

let dd = dragndrop(box); // prime the 'dragndrop' coroutine

box.onmousedown   = (event) => dd.next(event);
window.onmousemove = (event) => dd.next(event);
window.onmouseup   = (event) => dd.next(event);
```

In this code, the `dragndrop` coroutine, a JavaScript *generator*, implements the drag and drop sequence. The variable `dd` is initialized to the *primed* coroutine. The mouse events each resume `dd` with the current mouse event object.

Your task is to implement the `dragndrop` coroutine, whose skeleton is given below.

```
function* dragndrop(box) {
  // your code
}
```

Note that the event handlers pass an event object to the coroutine. Once your coroutine's `yield` receives the event object, call it `evt`, you can find out the type of the event by examining the `evt.type` property. The relevant values are `'mousedown'`, `'mousemove'`, or `'mouseup'`. To move the box to the correct position (in the case where dragging has been initiated and `evt.type == 'mousemove'`), you can simply call `move(box, evt)`. For the curious reader, here's an implementation of the `move` function.

```
function move(box, event) {
  box.style.left = event.pageX - box.parentNode.offsetLeft;
  box.style.top = event.pageY - box.parentNode.offsetTop;
}
```

ANSWER:
10 points

```
function* dragndrop(box) {
  while (true) {
    let evt = yield;
    if (evt.type == 'mousedown') {
      while (true) {
        let evt = yield;
        if (evt.type == 'mousemove') move(box, evt);
        if (evt.type == 'mouseup') break;
      }
    }
    // ignore all other kinds of events
  }
}
```

5. Program Correctness

a. Describe the meaning of the Hoare triple

$\{P\}\ S\ \{Q\}$

ANSWER:

5 points

The triple is an assertion that assuming the predicate P holds before executing the statement S, Q will hold when S terminates.

b. Provide a proof sketch of the (partial) correctness of the following Hoare triple using the program logic introduced in class. The variables x and y in the program are integers.

$\{x \leq y + 1\}$
while $(x < y)$ {
  co $\langle x = x + 1\rangle \parallel \langle y = y - 1\rangle$ **oc**
}
$\{x == y \ \vee \ x == y + 1\}$

The relevant proof rules are given below. Use these rules to justify your reasoning.

ASSIGNMENT
$\{P[e/x]\}\ x = e\ \{P\}$

SEQUENCING
$$\frac{\{P\}\ s_1\ \{Q\} \qquad \{Q\}\ s_2\ \{R\}}{\{P\}\ s_1;\ s_2\ \{R\}}$$

WHILE
$$\frac{\{I \wedge b\}\ s\ \{I\}}{\{I\}\ \textbf{while}\ (b)\ s\ \{I \wedge \neg b\}}$$

CONSEQUENCE
$$\frac{P' \Rightarrow P \qquad \{P\}\ s\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ s\ \{Q'\}}$$

AWAIT
$$\frac{\{P \wedge B\}\ s\ \{Q\}}{\{P\}\ \langle \textbf{await}\ (b)\ s\rangle\ \{Q\}}$$

CO
$$\frac{\{P_1\}\ s_1\ \{Q_1\} \qquad \cdots \qquad \{P_n\}\ s_n\ \{Q_n\} \qquad s_1, \ldots, s_n \text{ are interference-free}}{\{P_1 \wedge \cdots \wedge P_n\}\ \textbf{co}\ S_1 \parallel \cdots \parallel S_n\ \textbf{oc}\ \{Q_1 \wedge \cdots \wedge Q_n\}}$$

You can also use the usual laws of arithmetic and propositional logic in your proof sketch.

ANSWER:

15 points

We choose the a loop invariant as $I = x \leq y + 1$. The while-rule gives then:

$$\frac{\{x \leq y + 1 \wedge x < y\}\ s\ \{x \leq y + 1\}}{\{x \leq y + 1\}\ \textbf{while}\ (x < y)\ s\ \{x \leq y + 1 \wedge \neg(x < y)\}}\ \text{WHILE}$$

Simplifying

- $x \leq y + 1 \wedge x < y$ to $x < y$ and
- $x \leq y + 1 \wedge \neg(x < y)$ to $x == y \vee x == y + 1$,

the while rule instance becomes the following.

$$\frac{\{x < y\}\ s\ \{x \leq y + 1\}}{\{x \leq y + 1\}\ \textbf{while}\ (x < y)\ s\ \{x == y \vee x == y + 1\}}\ \textsc{While}$$

Substituting the loop body s, we get the proof obligation

$$\{x < y\}\ \textbf{co}\ \langle x = x + 1\rangle \parallel \langle y = y - 1\rangle\ \textbf{oc}\ \{x \leq y + 1\}.$$

Maybe it is clearer to write it as:

$$\{x < y\}\ \textbf{co}\ \langle x = x + 1\rangle \parallel \langle y = y - 1\rangle\ \textbf{oc}\ \{x < y + 2\}.$$

We must be careful to come up with pre and post conditions that are interference free, so that the co-rule can be applied. To this end, we introduce extra variables to stand for the "old" values of x and y, and formulate the pre and post conditions of the branches of the co-statement as follows, both proved with first the await rule and then the assignment axiom (not shown):

- $\{x = x_0 \wedge x_0 < y_0\}\langle x = x + 1\rangle\ \{x = x_0 + 1 \wedge x_0 < y_0\}$
- $\{y = y_0 \wedge x_0 < y_0\}\langle y = y - 1\rangle\ \{y = y_0 - 1 \wedge x_0 < y_0\}$

Then we can instantiate the co-rule as follows. The side condition that processes are interference free holds.

$$\frac{\{x = x_0 \wedge x_0 < y_0\}\langle x = x + 1\rangle\ \{x = x_0 + 1 \wedge x_0 < y_0\} \qquad \{y = y_0 \wedge x_0 < y_0\}\langle y = y - 1\rangle\ \{y = y_0 - 1 \wedge x_0 < y_0\}}{\{x = x_0 \wedge y = y_0 \wedge x_0 < y_0\}\ \textbf{co}\ \langle x = x + 1\rangle \parallel \langle y = y - 1\rangle\ \textbf{oc} \qquad \{x = x_0 + 1 \wedge y = y_0 - 1 \wedge x_0 < y_0\}}\ \textsc{Co}$$

We see that $x = x_0 \wedge x_0 < y_0$ is equivalent with $x < y$ and that $x = x_0 + 1 \wedge y = y_0 - 1 \wedge x_0 < y_0$ is equivalent with $x < y + 2$, which completes the proof.