Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 38 November 8, 1993

Solutions to Problem Set #4

The maximum score for this problem set was 55 points.

Problem 1 (10 points)

Consider the following Spec program:

```
VAR i: Int := 0

PROC P1 () =
     DO i//3#0 => i := i+1 OD

PROC P2 () =
     DO i//3#1 => i := i+1 OD

PROC P3 () =
     DO i//3#2 => i := i+1 OD
```

What is (are) the possible outcome(s) (that is, the final values of the state variables and any exceptions) of running `P1`*,* `P2` *and* `P3` *concurrently? Give reasons.*

This program can continue indefinitely. If it halts, it can halt in a state with `i` equal to any integer greater than `0`.

Most people had no trouble showing that the program could run indefinitely or halt in any state with `i>1`. Showing that it could halt with `i=1` was trickier.

An infinite execution can occur with the processes taking steps as follows (with the value of `i` after the action is taken):

P1	P2	P3	i
			0
	i//3#1=>i:=i+1		1
		i//3#2=>i:=i+1	2
i//3#0=>i:=i+1			3
	i//3#1=>i:=i+1		4
⋮	⋮	⋮	⋮

and so on. Each process executes in turn, repeatedly. Since `i` goes up by `1` when each process takes its turn, after three processes have taken their turns the value of `i//3` is unchanged, so no process will ever fail its test.

To see that the program can terminate in a state with `i=j`, where `j` is any positive integer, we use one of the following cases:

If `i>1`, then we take advantage of the fact that, in any state, exactly one process is capable of failing its loop test. Thus, we can run as in the above execution until `i=j-2`. At this point, exactly one process will be capable of halting; let it do so. Run one of the others for one loop (incrementing `i`, at which point a second process will be capable of halting; let it halt. Then the third process can run and increment `i`, and then it too will halt, and we are done.

As an example, here is the run terminating with `j=5`:

P1	P2	P3	i
			0
	`i//3#1=>i:=i+1`		1
		`i//3#2=>i:=i+1`	2
`i//3#0=>i:=i+1`			3
`i//3#0=>[fail]`			3
	`i//3#1=>i:=i+1`		4
	`i//3#1=>[fail]`		4
		`i//3#2=>i:=i+1`	5
		`i//3#2=>[fail]`	5

Finally the tricky case. To get the program to halt with `i=1`, we take advantage of the fact that an assignment `i:=i+1` takes place in two atomic steps; one to compute `i+1` (represented below by an invented "*compute*" operation) and the other to alter the value of `i`. Now, consider the following execution, in which we include the last value each process has computed for `i+1`.

P1	P1's i+1	P2	P2's i+1	P3	P3's i+1	i
						0
`i//3#0=>[fail]`						0
		`i//3#1=>`*compute*`(i+1)`	1			0
			1	`i//3#2=>i:=i+1`	1	1
			1	`i//3#2=>i:=i+1`	2	2
			1	`i//3#2=>[fail]`	2	2
		`i:=1`	1			1
		`i//3#1=>[fail]`	1			1

Problem 2 (14 points)

Consider the following set of Spec programs. Briefly explain what is (are) the possible outcome(s) (that is, the final values of the state variables and any exceptions) and the possible undesirable behavior(s) of running `P1` *and* `P2` *concurrently in each case.*

The three parts of this problem model different attempts at implementing a mutex between two concurrent procedures:

- (A) is an incorrect implementation, in that the intended mutual exclusion may be violated.
- (B) is a partially correct implementation that may loop forever in a peculiar way. Recall that partial correctness does not guarantee termination.
- (C) is a correct implementation of a mutex.

(It was not necessary to know the above in order to deduce the outcomes.)

(A) (4 points)

```
VAR c1: Bool := true
    c2: Bool := true

PROC P1() =
  DO ~c2 => SKIP OD;
  c1 := false;
  x := x + 1;
  c1 := true

PROC P2() =
  DO ~c1 => SKIP OD;
  c2 := false;
  x := x + 1;
  c2 := true
```

With the assumption that `x` is initially set to 0, if `P1` and `P2` run without interfering with each other (say, if one starts after the other is finished), the final value of `i` is 2 with both `c1` and `c2` reset to `true`.

However, it is possible that both `P1` and `P2` fail their respective guard in their `DO` loop before either one executes its second statement. Thereafter, both `P1` and `P2` may read the value of `x` equal to 0 and set it to 1. This is possible because the evaluation of the RHS and the assigment do not occur atomically. As a result, both procedures will terminate with the possible outcomes being:

```
x = 1, c1 = true, c2 = true
x = 2, c1 = true, c2 = true
```

(B) (5 points)

```
VAR c: Int
    x: Int := 0

PROC T1 () =
  c := 1;
  DO c=1 => SKIP OD
  x := x+1;
  c := 1

PROC T2 () =
  c := 2;
  DO c=2 => SKIP OD;
  x := x+1;
  c := 2
```

Intuitively, the variable `c` points to whichever of `T1` or `T2` set it last. One procedure can go ahead and access `x` if the other procedure set `c` later than itself—so that `c` points to the other procedure. Immediately after one of the procedures fails through the loop, the other must have executed its first statement and is left spinning in its `DO` loop. Thus it is not possible that both `T1` and `T2` are simultaneously executing their third statement, that is, incrementing `x`. Therefore, the final value of `x` is never 1, as is possible in (A).

However, if `T1` and `T2` run without interfering with each other (say, if one starts after the other is finished), then the first one will loop forever waiting for the other to execute. Therefore, the execution of `T1` and `T2` must interleave to avoid deadlock.

The possible outcomes with `T1` and `T2` interleaving are:

```
x = 2, c = 1
x = 2, c = 2
```

(C) (5 points)

```
VAR c1: Bool := true
    c2: Bool := true
    turn: Int := 1

PROC P1() =                          PROC P2() =
  c1 := false;                         c2 := false;
  DO ~c2 =>                            DO ~c1 =>
    turn=2 => c1 := true;                turn=1 => c2 := true;
              DO turn=2=>SKIP OD;                  DO turn=1=>SKIP OD;
              c1 := false                          c2 := false
    [*] SKIP                             [*] SKIP
  OD                                   OD
  x := x + 1;                          x := x + 1;
  turn := 2;                           turn := 1;
  c1 := true                           c2 := true
```

Intuitively, these procedures can be viewed as negotiation to acquire the exclusive access to `x`, followed by the modification of `x`, followed by releasing the exclusive access. `P1` sets variable `c1` to false to indicate that it would like access to `x`, and `P2` uses variable `c2` simlarly. Each procedure first announces its intent, then checks whether the other also wants access. Also, each procedure respects the other's intent reflected by the state of the corresponding `c`. If it is this procedure's turn, it waits for the other procedure to acknowledge the turn by withdrawing its request; otherwise, it immeditately withdraws its request and waits for its turn to be given by the other procedure. Therefore, each procedure can only exit the negotiation phase if it has agreed with the other procedure that it has the exclusive access.[1]

[1] Ideas from Cliff Young.

The following analysis is done with an assumption that x is initially 0. P1 is given a preference by having turn set to 1 initially in this problem. If both P1 and P2 announce their intent to access x by setting c1 and c2 to false respectively, P2 will find out that it is not its turn and immediately withdraw its request by setting c2 to true and wait for its turn to be given by P1. In the meantime, P1 will find out that it is its turn and wait for P2 to acknowledge the turn by withdrawing its request by c2 to true. Then P1 proceeds to increment x to 1 while P2 is waiting for its turn. After P1 exits with c1 reset to true, P2 continues to increment x from 1 to 2.

However, if any one procedure tests the guard in its second statement before the other executes its first statement, the earlier one will run to termination with the corresponding c reset to true before the latter one increments x from 1 to 2.

The possible outcomes are:

```
x = 2, turn = 1, c1 = true, c2 = true
x = 2, turn = 2, c1 = true, c2 = true
```

Problem 3 (16 points)

Consider the following Spec programs:

```
PROC P () -> Int =
     VAR i: Int, j: Int |
          j< 10 => BEGIN i*j>0 => RET 1 END
          [*]
          j>-10 => BEGIN
                         i*j<0 => RET 2
                         [*]
                         BEGIN i>= 0 => RET 3 [*] RET 4 END
                  END
```

Recall the non-atomic semantics of the following commands:

- R => S : If the guard R can be validated, the program counter is advanced to S. The successful validation itself counts as a transition and cannot be backed-off once it is made.

- S1 [*] S2 : If the first transition in S1 is enabled, it is made, irrespective of whether later transitions in S1 are enabled. Otherwise, S2 is tried.

For the sake of explanation, we name the three predicates in P as R1 (that is, j<10), R2 (i*j>0) and R3 (j>-10).

(A) (6 points)

What is (are) the possible outcome(s) (that is, the final values of the state variables and any exceptions) of running P. *Can* P *deadlock?*

Here the main command is of the form S1 [*] S2.

```
j< 10 => BEGIN i*j>0 => RET 1 END   % S1
[*]                                 % [*]
j>-10 => BEGIN ... END              % S2
```

It is always possible to make the first transition in `S1` because choosing `j<10` validates `R1`. Hence, `S2` is never tried, so 2, 3, or 4 is never returned. Any value may be chosen for `i` to accompany the choice of `j<10` for validating `R1`; as a result, `R2` may not be vailid after `R1` is. The state machine may get stuck waiting for `R2` to change status, which will never happen because `i` and `j` are local variables.

The only possible return vaule is therefore 1, and there is a possibilty of a deadlock. Let's use a table to cover all possible outcomes. In this table, `D` denotes a deadlock result and `N` denotes a not possiblly chosen case.

	j<=-10	-10<j<0	j=0	0<j<10	j<=10
i>0	D	D	D	1	N
i=0	D	D	D	D	N
i<0	1	1	D	D	N

(B) (5 points)

How does your answer to part A change if the first `[*]` *in* `P` *is replaced by* `[]`*?*

There will be non-deterministic results if `j<10 /\ j>-10` so that both `R1`, the first transition of `S1`, and `R3`, the first transition of `S2`, are valid. Hence, either `S1` or `S2` may be executed in this case.

The possible return values are `1`, `2`, `3` and `4`, and a deadlock is possible. Let's use a table to cover all possible outcomes. In this table, `D` denotes a deadlock result.

	j<=-10	-10<j<0	j=0	0<j<10	j<=10
i>0	D	D or 2	D or 3	1 or 3	3
i=0	D	D or 3	D or 3	D or 3	3
i<0	1	1 or 4	D or 4	D or 2	2

(C) (5 points)

How does your answer to part A change if `P` *is made an* `APROC` *and atomicity brackets added to its body?*

If `P` is made atomic, then its subparts are atomic too. Also, an atomic command is enabled or disabled as a whole (there is no intermediate state to get stuck in). Therefore, `S1` is executed only if it is fully enabled (that is, `j<10 /\ i*j>0`). Otherwise, the alternative is chosen. This is what is meant in saying that an atomic command can backtrack and choose the alternatives. Note that `APROC P` will not choose `j<=-10 /\ i>=0` such that neither statement can be fully executed.

The possible return values are `1`, `2`, `3` and `4`, without a possible deadlock result. Let's use a table to cover all possible outcomes. In this table, `N` denotes a not possibly chosen case. Note that this table is derived from that of part B.

	j<=10	-10<j<0	j=0	0<j<10	j<=10
i>0	N	2	3	1	3
i=0	N	3	3	3	3
i<0	1	1	4	2	2

Problem 4 (15 points)

Lampson's "Hints for Computer System Design" paper (Handout 6) divides the hints into three categories: functionality, speed, and fault-tolerance. Pick any two hints from two different categories, and describe how the designers of the BSD Log-Structured File System (Handout 10) applied the hints or failed to apply them. Answers should be a few paragraphs long; it is not enough to say ". . . is a good feature" or ". . . is undesirable."

We received many good answers to this question. The worst flaws were generally unclear writing (sometimes making it unclear which hints were being discussed) rather than a clear lack of understanding. (Also, most people seemed reluctant to say that the BSD-LFS team had in any way *failed* to apply any of Lampson's hints.)

The most common hint discussed was also the most obvious: "Log updates"; Since LFS takes this technique as its central theme, this was an easy choice. Other common choices included "Batch processing" (usually with reference to buffering writes), "Compute in background" and "Leave it to the client" (both with reference to the cleaner) and "Separate normal and worst case" (with reference to the dual recovery techniques).