Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

Examination: Tuesday, 17 March 2020

Examiner: K. V. S. Prasad, prasad@chalmers.se, 0736 30 28 22
(based on the course given Jan-Mar 2020)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes; English dictionary.

Grading: You can score a maximum of 70 points. Exam grades are:

points in exam	Grade Chalmers	Grade GU
28–41	3	G
42–55	4	G
56–70	5	VG

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows:

points in exam + labs	Grade Chalmers	Grade GU
40–59	3	G
60–79	4	G
80–100	5	VG

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 not receive any 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; six questions, numbered Q1 through Q6. 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.
- If a part of a question asks you to prove a proposition P, you can use P as a fact in later parts of the question, even if you didn't prove P.

Q1. (8p) Consider the following "Stop the loop" program.

boolean flag = false, b = false	
p	q
p1: while flag == false p2: b = !b	q1: while flag == false q2: if b == false q3: flag = true

(Part a). Give a scenario for which the program terminates. *(2p)*

(Part b). What are the possible values of b when the program terminates? Construct a scenario for each. *(2p)*

(Part c). Construct a non-terminating scenario for the program. *(2p)*

(Part d). Is your non-terminating scenario fair? *(2p)*

Answers:

(a) q1, q2, q3, p1, q1.

(b) The above yields n=0. And p1, q1, q2, p2, q3, p1, q1 yields n=1.

(c) p1, p2 (so that n = 1), then an infinite repetition of (q1, q2, p1, p2, p1, p2). This scenario never terminates.

(d) The scenario in (c) is fair (both processes run infinitely often.)

Q2. (14p) This question is about programming a *rendezvous* using binary semaphores, and then using the rendezvous.

If commands `...; b1; b2` occur in thread `B(oss)`, and commands `...; w1; w2` occur in thread `W(orker)`, then thread `B` is said to rendezvous with thread `W` if `b1` happens before `w2` and `w1` before `b2`. As in everyday use, rendezvous means the one who arrives earlier waits for the later. But in everyday use, the both people might leave at the same time, which is not possible in an interleaving model, so we merely insist on the sequence above. `B` leaves after `W` arrives, and vice-versa.

(Part a) Does this code implement a rendezvous between `B` and `W`?

Semaphore Ba = 0, Wa = 0 // both initialised to 0	
B	W

b1: // command b1
b2: Ba.release();
b3: Wa.acquire();
b4: // command b2
	w1: // command w1
	w2: Ba.acquire();
	w3: Wa.release();
	w4: // command w2

Prove that in any scenario, neither thread leaves until both have arrived. *(3p)*

Answer: b3 must await w3. w3 awaits w2 awaits b2. So neither can finish until the other at least starts. If W arrives first, it waits for B. When B arrives, it wakes W and might proceed immediately to its wait

```
protected object PC
  l = int[];          //buffer itself as integer circular array
  int Cap = 10;       //buffer capacity
  int pi= 0, ci= 0; //producer, consumer indices into buffer

  operation void prod (x) when pi-ci<Cap    //produce
    {pi=pi+1; l[pi%Cap ] = x };
  operation int cons () when ci<pi          //consume
    {ci=ci+1; int y= l[ci%Cap ]; return y};
```

	producer		consumer
	int x;		**int** x;
	while (true) {		**while** (true) {
p_1:	x= //*make it*;	c_1:	x=PC.cons ();
p_2:	PC.prod (x) ;	c_2:	//*eat x*;
	}		}

Figure 1: Producer-consumer example to show protected object notation.

in which case it blocks, allowing W to reach its release, after which both threads can proceed. Show all paths achieve randezvous.

(Part b) The solution in **(Part a)** might switch between `B` and `W` one time more than necessary. Is there a more efficient solution? *(2p)*

Answer: Both release their own, then acquire the other. Then b3 awaits w2, and w3 awaits b2. So neither can finish until the other at least starts. But the first to arrive can leave as soon as the other arrives.

(Part c) What happens if both threads acquire before release? *(1p)*

Answer: Deadlock.

(Part d) Program an n-way rendezvous using binary semaphores. When the first $n-1$ threads arrive, they block until the nth thread arrives; then all the threads proceed. *Hint:* Let `count` remember how many threads have arrived, let semaphore `mutex` provide exclusive access to `count`, and let semaphore `barrier` be locked until all threads arrive, and unlocked after. *(4p)*

Answer:

```
mutex. wait ();  count++;  mutex. signal ();
if count == n then barrier. signal ();  //otherwise wait at barrier
barrier. wait (); barrier. signal ();  //cascade

all leave   //works for single rendezvous, not repeated
```

(Part e) Repeat (Part a) using a monitor or a protected object. You can answer using any notation you like. You will not be penalised for getting the syntax wrong. *(4p)*

Answer:

```
prot obj barr

    int N = 10  //how many will rendezcvous?
    int n = 0   //how many have arrived?
    bool done = false  //round still ongoing

    operation void arrive when not done
        n++;
        if n == N then done = true

    operation void leave when done
        n--;
        if n==0 then done = false


participant
    loop
        barr. arrive; barr. leave
```

The **producer-consumer** problem: A *producer* (a cook) and a *consumer* (a diner) share a buffer (rotating table) on which the cook places fresh pancakes *when there is a space*, and from which the diner takes a pancake *when one is available*. The cook needs time to make a pancake, and the diner time to eat one, but we skip these details.

Refresher on protected objects. Figure 1 shows a **protected object** `PC` to solve the producer-consumer problem. Like a monitor, `PC` provides encapsulates access to a data structure, in this case the buffer in `PC`. Only one **operation** of a protected object can execute at a time.

Unlike a monitor, a protected object does not provide condition variables. Instead, an operation can have a guard (a boolean condition following the keyword **when**), which must hold for the operation to execute; otherwise th e operation is blocked. Thus `PC.prod` can execute only when there is a slot free on the table, and `PC.cons` can only run if there is a pancake on the table.

Calling processes are blocked on a FIFO queue, a separate one for each operation. After any operation, all the guards are re-evaluated automatically. So it is not the producer's responsibility to explicitly

signal the consumer, or vice-versa. That is all done by `PC`, which can also accommodate multiple cooks and diners, who all use the same `prod` and `cons` operations. *End of refresher.*

Code below for Q3 and Q4

Let the command `iaf` (`increment-and-fetch`) be defined as below.

```
int iaf(int C) {
  atomic{
    C++;
    return C;
  }
}
```

	int C= 0; T= 1; D=0 //Counter, Turn, Dummy	
	p	q
	int P=-1; //p's turn	**int** Q=-2; //q's turn
	while (true) {	**while** (true) {
	//NCS (non-critical section)	*//NCS (non-critical section)*
p_2:	P=iaf(C) ;	q_2: Q=iaf(C);
p_3:	**while** (P != T) **do** {};	q_3: **while** (Q != T) **do** {};
	//CS (critical section)	*//CS (critical section)*
p_5:	D=iaf(T);	q_5: D=iaf(T);
	}	}

Figure 2: Code for **Q3** and **Q4**.

Notes: The pseudo-code above tries to solve the critical section (CS) problem with two *interleaving* threads, `p` and `q`. A command in one thread cannot overlap with a command in the other thread.

Remember that **(1):** an **atomic** sequence of commands happen as one step, without the other thread acting partway through the atomic sequence, **(2):** CS executes a finite number of commands, but NCS may loop, **and (3):** We use the notation p_i to mean both the label of a command, and to mean the proposition that thread `p` has reached p_i, so *the next command* `p` will execute is p_i.

Bridge example: A two-lane road between the towns of Persimmon and Quince has a single lane bridge on the way, to cross which you need a ticket (nummerlapp) from dispensers at either end of the bridge. Persimmon-bound cars `p` take a ticket by the command p_2 `iaf(C)` and wait, move on to the bridge by p_3, and leave the bridge by p_5 `iaf(T)`, incrementing the ticket number to be served. Similarly for Quince-bound cars `q`.

Q3 (22p). There are many parts to this question, and you may use the result of earlier parts even if you don't answer them.

First, reason some preliminaries from the program text. For a refresher on logical notation, see the Appendix.

(Part a) Show that every update to any of the variables `C`, `T`, `P`, `Q` sets it to an integer greater than or equal to 1. *(1p)*

Answer: The first execution of p_2 or q_2 sets `C` to 1. All further assignments to these variables are increments in p_2, q_2, p_5 or q_5.

Value range of `C-T`. As the processes `p` and `q` run, the combination of their program counters takes values from the set PQ, which, listed in dictionary order, is $PQ = \{p_2q_2, p_2q_3, p_2q_5, p_3q_2, p_3q_3, p_3q_5, p_5q_2, p_5q_3, p_5q_5\}$. We partition PQ into three subsets:

`CT-` $= \{p_2q_2\}$, (here, the bridge is idle, and `C`=`T`-1)

`CT0` $= \{p_2q_3, p_2q_5, p_3q_2, p_5q_2\}$, (here, one car has a ticket and may be crossing the bridge; `C`=`T`)

`CT+` $= \{p_3q_3, p_3q_5, p_5q_3, p_5q_5\}$ (here, two cars have tickets and may be crossing the bridge; `C`=`T`+1)

(Part b).

1 Show that any p_2 or q_2 move takes the system from any pq in `CT-` to a pq in `CT0`, and from any pq in `CT0` to a pq in `CT+`.

2 Show that any p_5 or q_5 move takes the system from any pq in `CT0` to a pq in `CT-`, and from any pq in `CT1` to a pq in `CT0`.

3 Show that a p_3 or q_3 move takes the system from a pq in `CT0` to another pq in `CT0`, and from a pq in `CT+` to another pq in `CT+`.

4 `P` is an *active ticket* if `P` $\geq$ `T`; similary `Q`. There can be at most two active tickets, and these are consecutive. Why?

(4p)

Answer: 1) p_2 or q_2 increases `C`, but not `T`. 2) p_5 or q_5 increases `T`, but not `C`. 3) p_3 or q_3 change neither `C` nor `T`. 4) `p` has to do a p_2 (get a new `P`) and p_5 (discard `P`) in alteration, except if it gets stuck at p_3. So there can be at nost two new tickets, when both `p` and `q` have gotten a new ticket but not yet discarded it. Why do we say "discard"? Because once a `P` is accepted, `T` will increase past it and never return.

We have considered all moves in every pq, so {-1, 0, 1} are all the possible values of `C-T`. But pq's are not states of the system, which need to keep track of `P` and `Q` too. We wouldn't expect to find a program state with two cars on the bridge at one time, like p_5q_5.

Aid for graders: Here is a table of the positions of `p` and `q`, showing the value of `C-T`. The "transitions" ignore the values of `P` and `Q`, simply assuming every move is possible. The purpose of the table is to examine the change in value of `C-T`. We write "-" for `C-T`=-1, "0" for `C-T`=0, and "+" for `C-T`=1. These are not real states of the program.

position combination = $p_i q_j$, C-T		*new state if p moves*	*new state if q moves*
pc1	$p_2 q_2 -$	$p_3 q_2 0$ = *pc4*	$p_2 q_3 0$ = *pc2*
pc2	$p_2 q_3 0$	$p_3 q_3 +$ = *pc5*	$p_2 q_5 0$ = *pc3*
pc3	$p_2 q_5 0$	$p_3 q_5 +$ = *pc6*	$p_2 q_2 -$ = *pc1*
pc4	$p_3 q_2 0$	$p_5 q_2 0$ = *pc7*	$p_3 q_3 +$ = *pc5*
pc5	$p_3 q_3 +$	$p_5 q_3 +$ = *pc8*	$p_3 q_5 +$ = *pc6*
pc6	$p_3 q_5 +$	$p_5 q_5 +$ = *pc9*	$p_3 q_2 0$ = *pc4*
pc7	$p_5 q_2 0$	$p_2 q_2 -$ = *pc1*	$p_5 q_3 +$ = *pc8*
pc8	$p_5 q_3 +$	$p_2 q_3 0$ = *pc2*	$p_5 q_5 +$ = *pc9*
pc9	$p_5 q_5 +$	$p_2 q_5 0$ = *pc2*	$p_5 q_2 0$ = *pc7*

end of teacher aid

(Part c) How can q influence the value assigned to P? *(1p).*

Answer: P is set only in p_2, but can be influenced by q via C.

(Part d) Prove $\mathtt{P} \neq \mathtt{Q}$ is an invariant. *(1p)*

Answer: C is monotonically increasing. P and Q are set to C as it is incremented, so they can never be set to the same value.

Matter subsumed by newer formulation:

We define a **p-round** *(resp.* **q-round***) as a return by* p *(resp.* q*) to its NCS. So a* p*-round is an execution of p_2, p_3, and p_5, possibly with interleaved actions by* q*.*

(Part d) Let R be $(p_2 \wedge q_2) \rightarrow (\mathtt{C} = \mathtt{T} - 1)$. Show that R is invariant. *Hint:* By induction on the number of returns to $p_2 \wedge q_2$. *(3p)*

Answer: R holds at the start, with C=0 and T=1. Now suppose at some point, we have $p_2 \wedge q_2$ and C=T-1. Then by the next time we have $p_2 \wedge q_2$ again, a whole number of p*-rounds and a whole number of* q*-rounds would have happened. So there would have been as many calls of iaf(C) as of iaf(T), and C and T would hve increased equally. So C=T-1, still.*

Answer: Consider $(p_2 \wedge q_3)$. We can get there from $(p_2 \wedge q_2)$, when C increases from T-1 to T. Or from $(p_5 \wedge q_3)$ where

Now suppose at some point, we have $p_2 \wedge q_2$ and C=T-1. Then by the next time we have $p_2 \wedge q_2$ again, a whole number of p*-rounds and a whole number of* q*-rounds would have happened. So there would have been as many calls of iaf(C) as of iaf(T), and C and T would hve increased equally. So C=T-1, still.*

End of Matter subsumed by newer formulation:

To make a state-transition table (S-T table), we need a full state description $(p_i, q_j, \mathtt{P}, \mathtt{Q}, \mathtt{C}, \mathtt{T})$, saying where p and q are, and what values

the variables have. To make the table manageable, we drop the parentheses and the commas. As with the sets above, we represent `C-T` by adding a -, 0, or + after the $p_i q_j$.

Note that `C` and `T` increase forever, yet the table is finite, because `C` and `T` stay close, so it's a small sliding window.

(Part e) Why do we need only `C-T` and not `C` and `T` individually? *(1p)*

Answer: Because p_3 indirectly compares `C` and `T`, and that is the only use made of these variables.

Q3 continues on next page.

Code below for Q3 and Q4

Let the command `iaf` (`increment-and-fetch`) be defined as below.

```
int iaf(int C) {
  atomic{
    C++;
    return C;
  }
}
```

The pseudo-code below tries to solve the critical section (CS) problem with two *interleaving* threads, `p` and `q`. A command in one thread cannot overlap with a command in the other thread.

	int C= 0; T= 1; D=0 //Counter, Turn, Dummy		
	p		q
	int P=-1; //p's turn		**int** Q=-2; //q's turn
	while (true) {		**while** (true) {
	//NCS (non-critical section)		*//NCS (non-critical section)*
p_2:	P=iaf(C) ;	q_2:	Q=iaf(C);
p_3:	**while** (P != T) **do** {};	q_3:	**while** (Q != T) **do** {};
	//CS (critical section)		*//CS (critical section)*
p_5:	D=iaf(T);	q_5:	D=iaf(T);
	}		}

Figure 3: Code for **Q3** and **Q4**, repeated for convenience.

Note that variable `P` (resp. `Q`) is written in p_2 (resp. q_2), but read only in p_3 (resp. q_3). So we drop `P`, `Q` except in p_3 and q_3. We use the notation p_3 for "`p` at label p_3 and `P`$\neq$`T`", and "p'_3 for `p` at label p_3 and `P`=`T`". Simlarly for q_3 and q'_3. *Beware: If `T` changes, the primes change.* In this abbreviated notation, here is a partial S-T table. A "spin" entry means next state = current state.

state = p_iq_j (un)primed, `C-T`		new state if p moves	new state if q moves
s1	p_2q_2-		$p_2q'_30$ = s2
s2	$p_2q'_30$		p_2q_50 = s3
s3	p_2q_50	p_3q_5+ = s7	p_2q_2- = s1
s4			
s5	p'_3q_3+	p_5q_3+ = s9	spin = s5
s6			
s7	p_3q_5+	spin = s7	
s8			
s9	p_5q_3+	$p_2q'_30$ = s2	spin = s9

(Part f) Complete the table above. (In the answer sheet, which has space for corrections. Write down the missing entries and rows). *(6p)*

Here is the full table with no blanks.

state = p_iq_j (un)primed, C-T		new state if p moves	new state if q moves
s1	p_2q_2-	p'_3q_20 = s4	$p_2q'_30$ = s2
s2	$p_2q'_30$	$p_3q'_3+$ = s6	p_2q_50 = s3
s3	p_2q_50	p_3q_5+ = s7	p_2q_2- = s1
s4	p'_3q_20	p_5q_20 = s8	p'_3q_3+ = s5
s5	p'_3q_3+	p_5q_3+ = s9	spin = s5
s6	$p_3q'_3+$	spin = s6	p_3q_5+ = s7
s7	p_3q_5+	spin = s7	p'_3q_20 = s4
s8	p'_5q_20	p_2q_2- = s1	p_5q_3+ = s9
s9	p_5q_3+	$p_2q'_30$ = s2	spin = s9

Here is the monster table.

state = p_iq_j (un)primed, C-T		new state if p moves	new state if q moves
s1	$p_2q_2-, P_{-1}Q_{-2}C_0T_1$	$p'_3q_20, P_1Q_{-2}C_1T_1$=s4	$p_2q'_30, P_{-1}Q_1C_1T_1$=s2
s2	$p_2q'_30, P_{-1}Q_1C_1T_1$	$p_3q'_3+, P_2Q_1C_2T_1 = s6$	$p_2q_50, P_{-1}Q_1C_1T_1 = s3$
s3	$p_2q_50, P_{-1}Q_1C_1T_1$	$p_3q_5+, P_2Q_1C_2T_1 = s7$	$p_2q_2-, P_{-1}Q_1C_1T_2 = s1$
s4	$p'_3q_20, P_1Q_{-2}C_1T_1$	$p_5q_20, P_1Q_{-2}C_1T_1$=s8	$p'_3q_3+, P_1Q_2C_2T_1$=s5
s5	$p'_3q_3+, P_1Q_2C_2T_1$	$p_5q_3+, P_1Q_2C_2T_1$=s9	spin = s5
s6	$p_3q'_3+, P_2Q_1C_2T_1 = s6$	spin = s6	$p_3q_5+, P_2Q_1C_2T_1 = s7$
s7	$p_3q_5+, P_2Q_1C_2T_1 = s7$	spin = s7	$p'_3q_20, P_2Q_1C_2T_2 = s4$
s8	$p_5q_20, P_1Q_{-2}C_1T_1$=s8	$p_2q_2-, P_1Q_{-2}C_1T_2$=s1	$p_5q_3+, P_1Q_2C_2T_1$
s9	$p_5q_3+, P_1Q_2C_2T_1$=s9	$p_2q'_30, P_1Q_2C_2T_2$=s9	spin = s9

(Part g) Why is it sufficient to only note in the table whether P and Q are equal to T or not? *(1p)*

Answer: Because P is used only in p_3 to test if P=T.

(Part h) From your complete S-T table, show that p and q cannot both be in their respective CS at the same time. *(1p).*

Answer: No state with p_5q_5.

(Part i) From your table, show that the program can't livelock. *(1p).*

Answer: No state with both entries "spin".

(Part j) Prove that given fair scheduling, every $p2$-state (one where p is at $p2$) will lead at some future point to a $p5$-state. *Hint:* Iteratively build a set S of all states that must lead to a $p5$-state in zero, one or more moves. First, $S :=$ the set of all $p5$-states. Next, $S := S \cup \{s_5\}$, as s_5 must lead to a p_5-state. Proceed like this. The remaining states will then form a loop. Show that every state of this loop has a p action that leads to S. *(4p)*

Answer:States in S: s8, s9. Add s5 by fairness. Add s4 as both arms lead to sS. Add s7; must lead to 4. Add s6; must lead to s7. That leaves s1, s2, s3 loop, but by fairness, p must act some time and lead to S.

(Part k) The table has very few rows. How do we know the other combinations of variables are not relevant? *(1p).*

Answer: The table is closed and shows all reachable states.

Q4 (10p). In this question, you must argue from the program text, not from the S-T table.

(Part a) Mutex: Let M be $\neg(p_5 \wedge q_5)$. Prove by induction that M is an invariant. First show that M holds for the start state $s_1 = p_2 q_2 -$. Then show that if M holds for any state s, then it holds for all the successors of s. Think which states could have a successor where M could be false. *(4p)*

Answer: s_1 trivially satisfies M. The only way to falsify M is to start from a state with p_5 and move to a state with also q_5. The only such state is $p_5 q_3$. But p_5 means `P=T`*, and* `T` *hasn't changed. But* `Q≠P`*, so* `Q≠T`*, so q_3 spins, and cannot proceed to q_5.*

Removed quetion: **(Part f)** Show that $(p_2 \wedge q_5) \rightarrow$ (`C` = `T`). *Hint:* Suppose not, and `q` acts. *(1p)*

Answer: A `q` *action leads to Part e. So (*`C≠T`*) leads to a contradiction.*

(Part b) Show that starting from any $p_2 q_2$ state (i.e., a state where `p` is at p_2, and `q` is at q_2), any sequence that leads to a $p_3 q_3$ state will result in exactly one of `p` or `q` spinning. *(2p)*

Answer: If `p` *is first to act, then* `P=T`*, so* `p` *won't spin, and* `q` *will.*

(Part c) Show that starting from any $p_2 q_5$ state (i.e., a state where `p` is at p_2, and `q` is at q_5), any sequence that leads to a $p_3 q_3$ state will result in `q` spinning, and `p` moving on to p_5. *(2p)*

Answer: If `p` *acts first, then* `P>T`*, so* `p` *spins, till* `q` *acts, and will then spin. If* `q` *acts first, we have (Part g).*

(Together, (Parts b and c) help show absence of livelock).

(Part d) Suppose you were to replace command p_5 : `D` = `iaf`(`T`) by p_5 : `T` = `T` + 1. Similary for q_5. What would happen? *(2p)*

Answer: Nothing. The old p_5 uses `iaf` *to atomically increment* `T`*, and discards the value. But mutex has been shown, so p_5 can increment* `T` *without* `q` *interfering.*

Q5 (8p). Write an insertion sort in Erlang. The idea is that given the list [7,3,8,5] as input, the program should go through the following development. `feed`(list of integers), `cell`(integer), and `done`() are processes with parameters of the types shown. The input list is fed left-to-right through a chain of processes. An arrow between processes I and J means that I knows J's Pid, and sends its output to J. A process receives its input from the left.

- fed in: `feed`([7,3,8,5]) → `done`

7 fed in: `feed`([3,8,5]) → `cell`(7) → `done`

3 fed in: `feed`([8,5]) → `cell`(3) → `cell`(7) →

8 fed in: `feed`([5]) → `cell`(3) → `cell`(7) → `cell`(8) → `done`

5 fed in: `feed`([]) → `cell`(3) → `cell`(5) → `cell`(7) → `cell`(8) → `done`

(Part a) Write the code for `cell` and `done`. The exact syntax doesn't matter, but the intent should be clear. *(4p)*

(Part b) What parts of the computation to feed in 8 can overlap computation to feed in 5? *(3p)*

(Part c) Sketch how to implement a command to print out the sorted list. *(1p)*

Removed part:

(Part d) Sketch a recursive modification of your program to accept rational numbers r such that $0 \leq \mathrm{r} < 10$, with arbitrary numbers of decimal places. If fed in [7, 3.1, 5, 3.14], the program should in effect store [[3.1, [3.14]], 5, 7], i.e., there should be nested chains of processes representing the nested lists. *(4p)*

```
public class PC
  l = int[];                  //buffer itself as an integer array
  int Cap = 10;               //buffer capacity
  int pi= 0, ci= 0;           //producer, consumer indices into buffer

  public void prod (x)                    //produce
    {  //pre-protocol
       l[pi%Cap ]=x; pi=pi+1;          //% is the modulo operator
       //post-protocol
    }
  public int cons ()                      //consume
    {  //pre-protocol
       ci=ci+1; int y= l[ci%Cap ];
       //post-protocol
       return y;
    }
```

	producer		consumer
	int x;		int x;
	while (true) {		while (true) {
p_1:	x= //*make it*;	c_1:	x=PC.cons ();
p_2:	PC.prod (x) ;	c_2:	//*eat x*;
	}		}

Figure 4: Producer-consumer template.

Q6 (12p). A description of the **producer-consumer** problem can be found on page 3. Figure 4 shows a object-based template to solve the problem.

Below, assume one `prod` and one `cons` each running on its own CPU.

(Part a) Declare any binary semaphores you need, and write code for the pre- and post-protocols of the methods `prod` and `cons`, so that a `prod` waits if the buffer is full, and a `cons` waits if the buffer is empty. Under what conditions can `prod` and `cons` work in parallel? *(3p)*

(Part b) Using no locks, semaphores, or other locking constructs, write code for the pre- and post-protocols of the methods `prod` and `cons`, so that a `prod` does not over-write a full buffer, and a `cons` does not read from an empty buffer. Under what conditions can `prod` and `cons` work in parallel? *(2p)*

(Part c) Make the code in Figure 4 nicely linearizable. Where are the linearization points? How do we use linearization points in reasoning about the program? *(4p)*

(Part d) Are there linearization points in lock-based operations on a shared object? Where are the linearization points in Figure 1? *(3p)*

A Linear Temporal Logic (LTL) notation

1. An atomic proposition such as q_2 (process q is at label q_2) *holds for* a state s if and only if process q is at q_2 in s.

2. Let ϕ and ψ be formulas of LTL. Formulas are either atomic propositions, or are built up from other formulas using the following operators: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", $\rightarrow$ for "implies", $\Box$ for "always", and $\Diamond$ for "eventually". A convenient abbreviation is ϕ iff ψ (i.e., ϕ if and only if ψ) for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

   These operators have the obvious meanings, but two differ from what might be your interpretation of the names. First, $\phi \vee \psi$ ("ϕ or ψ") is false iff both ϕ and ψ are false. This is an "inclusive or", so $\phi \vee \psi$ is also true if both ϕ and ψ are true. Second, $\phi \rightarrow \psi$ ("ϕ implies ψ") is false iff ϕ is true and ψ is false. So, in particular, $\phi \rightarrow \psi$ is true if ϕ is false. The meanings of the operators $\Box$ and $\Diamond$ are defined below.

3. A *path* is a possible future of the system, a possibly infinite sequence of states, each reachable from the previous state in the path. A state s *satisfies* formula ϕ if every path from s satisfies ϕ.

   A path π satisfies $\Box\phi$ if ϕ holds for the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ if ϕ holds for some state in π.

   Note that $\Box$ and $\Diamond$ are duals:

$$\Box\phi \equiv \neg\Diamond\neg\phi \qquad \text{and} \qquad \Diamond\phi \equiv \neg\Box\neg\phi.$$