IN5170/IN9170 – Models of concurrency

Program Analysis II, lecture 6

Fall 2022

26.09.2022



Program Analysis: Part II Concurrency

Today

Analysis Part I: Reasoning about sequential programs.

Analysis Part II: How to reason about concurrent systems.

- Repetition of Rules
- Await statements
- Shared variables and interference
  - Concurrent execution
  - Interference freedom
  - Example: Producer / Consumer (await language)
- Monitors
  - invariants
  - Signal/wait
  - Example: reader/writer
  - Example: semaphores

Program Logic (PL)

- PL lets us *express* and *prove* properties about programs
- *Formulas* are on the form

"triple" (or "Hoare triple")

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

- S: program statement(s)
- P and Q: assertions over program states
- P: Pre-condition
- Q: Post-condition

If we can use PL to prove some property of a program, then this property will hold for *all* executions of the program

PL rules from part I: structural inference rules

$$\frac{P' \Rightarrow P \qquad \{\, P \,\}\ S\ \{\, Q \,\} \qquad Q \Rightarrow Q'}{\{\, P' \,\}\ S\ \{\, Q' \,\}}\ \text{Consequence}$$

$$\frac{\{\, P \,\}\ S\ \{\, Q \,\} \qquad \{\, P' \,\}\ S\ \{\, Q' \,\}}{\{\, P \wedge P' \,\}\ S\ \{\, Q \wedge Q' \,\}}\ \text{Conjunction}$$

$$\frac{\{\, P \,\}\ S_1\ \{\, R \,\} \qquad \{\, R \,\}\ S_2\ \{\, Q \,\}}{\{\, P \,\}\ S_1; S_2\ \{\, Q \,\}}\ \text{Seq}$$

- Red: logical proof obligations

PL rules from part I: if/while/assignment

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \} \qquad P \wedge \neg B \Rightarrow Q}{\{\ P\ \}\ \text{if } B \text{ then } S \text{ fi}\ \{\ Q\ \}}\ \text{Cond1}$$

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \} \qquad \{\ P \wedge \neg B\ \}\ S'\ \{\ Q\ \}}{\{\ P\ \}\ \text{if } B \text{ then } S \text{ else } S'\ \{\ Q\ \}}\ \text{Cond2}$$

$$\{\ P[e/x]\ \}\ x := e\ \{\ P\ \}\quad \text{Assign}$$

$$\frac{\{\ I \wedge B\ \}\ S\ \{\ I\ \}}{\{\ I\ \}\ \text{while } B \text{ do } S\ \{\ I \wedge \neg B\ \}}\ \text{While}$$

Restrictions

- No side-effects (in expressions)
- No remote access by pointers
- No pointer arithmetic
- So far, we did not consider arrays

And so far no concurrency!

How to actually *use* the while rule?

- Cannot control execution in the same way as for **if**-statements
  - Cannot tell *from the code* how many times the loop body will be executed (not a "syntax-directed" rule)

$$\{\ y \geq 0\ \}\ \text{while}\ (y > 0)\ \ \text{do}\ y := y - 1$$

  - Cannot speak about the state after the first, second, third . . . iteration

How to actually *use* the while rule?

- Cannot control execution in the same way as for **if**-statements
  - Cannot tell *from the code* how many times the loop body will be executed (not a "syntax-directed" rule)

$$\{\ y \geq 0\ \} \text{ while } (y > 0) \text{ do } y := y - 1$$

  - Cannot speak about the state after the first, second, third . . . iteration

- Solution: Find assertion I that is maintained by the loop body
  - Loop invariant: express a property preserved by the loop

While rule

$$\frac{\{\ I \wedge B\ \}\ S\ \{\ I\ \}}{\{\ I\ \}\ \text{while } B \text{ do } S\ \{\ I \wedge \neg B\ \}}\ \text{While}$$

We can use this rule to reason about the general situation:

$$\{\ P\ \}\ \text{while } B \text{ do } S\ \{\ Q\ \}$$

where

- P need not be the loop invariant
- Q need not match $(I \wedge \neg B)$ syntactically

While rule

$$\frac{\{\, I \wedge B \,\}\ S\ \{\, I \,\}}{\{\, I \,\}\ \text{while } B \text{ do } S\ \{\, I \wedge \neg B \,\}}\ \text{While}$$

We can use this rule to reason about the general situation:

$$\{\, P \,\}\ \text{while } B \text{ do } S\ \{\, Q \,\}$$

where

- P need not be the loop invariant
- Q need not match $(I \wedge \neg B)$ syntactically

Combine While-rule with Consequence-rule to prove:

- **Entry:** $P \Rightarrow I$
- **Loop:** $\{\, I \wedge B \,\}\ S\ \{\, I \,\}$
- **Exit:** $I \wedge \neg B \Rightarrow Q$

While rule

$$\frac{\{\, I \wedge B \,\}\ S\ \{\, I \,\}}{\{\, I \,\}\ \text{while } B \text{ do } S\ \{\, I \wedge \neg B \,\}}\ \text{While}$$

We can use this rule to reason about the general situation:

$$\{\, P \,\}\ \text{while } B \text{ do } S\ \{\, Q \,\}$$

where

- P need not be the loop invariant
- Q need not match $(I \wedge \neg B)$ syntactically

Combine While-rule with Consequence-rule to prove:

- **Entry:** $P \Rightarrow I$
- **Loop:** $\{\, I \wedge B \,\}\ S\ \{\, I \,\}$
- **Exit:** $I \wedge \neg B \Rightarrow Q$

$$\frac{P \Rightarrow I \qquad \{\, I \wedge B \,\}\ S\ \{\, I \,\} \qquad I \wedge \neg B \Rightarrow Q}{\{\, P \,\}\ \text{while } B \text{ do } S\ \{\, Q \,\}}$$

While rule: example

$\{\ 0 \leq n\ \}\ k := 0;\ \{\qquad\qquad\}\ \text{while}\ \ (k < n)\ k := k + 1;\ \{\ k = n\ \}$

While rule: example

$$\{\ 0 \leq n\ \}\ k := 0;\ \{\ k \leq n\ \}\ \text{while}\ (k < n)\ k := k + 1;\ \{\ k = n\ \}$$

Composition rule splits a proof in two: assignment and loop.

While rule: example

$$\{\, 0 \leq n \,\}\ k := 0;\ \{\, k \leq n \,\}\ \text{while}\ (k < n)\ k := k + 1;\ \{\, k = n \,\}$$

Composition rule splits a proof in two: assignment and loop.

Let $k \leq n$ be the loop invariant

While rule: example

$$\{\ 0 \leq n\ \}\ k := 0;\ \{\ k \leq n\ \}\ \text{while}\ (k < n)\ k := k + 1;\ \{\ k = n\ \}$$

Composition rule splits a proof in two: assignment and loop.

Let $k \leq n$ be the loop invariant

- **Entry:** $k \leq n$ follows from itself
- **Loop:**

$$\frac{k < n \Rightarrow k + 1 \leq n}{\{\ k \leq n \wedge k < n\ \}\ k := k + 1\ \{\ k \leq n\ \}}$$

- **Exit:** $(k \leq n \wedge \neg(k < n)) \Rightarrow k = n$

Rule for await

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \}}{\{\ P\ \}\ \langle \text{await}\,(B)\ S \rangle\ \{\ Q\ \}}\ \text{Await}$$

Remember:
We are reasoning about safety properties/partial correctness

- termination is assumed/ignored
- the rule does not ensure waiting nor progress

Assume two statements S_1 and S_2 such that:

$$\{ P_1 \} \langle S_1 \rangle \{ Q_1 \} \quad \text{and} \quad \{ P_2 \} \langle S_2 \rangle \{ Q_2 \}$$

To simplify, we let S_1 and S_2 be atomic regions.

Assume two statements S_1 and S_2 such that:

$$\{\ P_1\ \}\ \langle S_1 \rangle\ \{\ Q_1\ \} \quad \text{and} \quad \{\ P_2\ \}\ \langle S_2 \rangle\ \{\ Q_2\ \}$$

To simplify, we let S_1 and S_2 be atomic regions.
First attempt for a co . . . oc rule in PL:

$$\frac{\{\ P_1\ \}\ \langle S_1 \rangle\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ \langle S_2 \rangle\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ \text{co}\ \langle S_1 \rangle \parallel \langle S_2 \rangle\ \text{oc}\ \{\ Q_1 \wedge Q_2\ \}}\ \text{Par}$$

Assume two statements S_1 and S_2 such that:

$$\{\ P_1\ \}\ \langle S_1 \rangle\ \{\ Q_1\ \} \quad \text{and} \quad \{\ P_2\ \}\ \langle S_2 \rangle\ \{\ Q_2\ \}$$

To simplify, we let S_1 and S_2 be atomic regions.
First attempt for a co . . . oc rule in PL:

$$\frac{\{\ P_1\ \}\ \langle S_1 \rangle\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ \langle S_2 \rangle\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ \text{co}\ \langle S_1 \rangle \parallel \langle S_2 \rangle\ \text{oc}\ \{\ Q_1 \wedge Q_2\ \}}\ \text{Par}$$

Example (Problem with this rule)

$$\frac{\{\ x = 0\ \}\ \langle x := x + 1 \rangle\ \{\ x = 1\ \} \qquad \{\ x = 0\ \}\ \langle x := x + 2 \rangle\ \{\ x = 2\ \}}{\{\ x = 0\ \}\ \text{co}\ \langle x := x + 1 \rangle \parallel \langle x = x + 2 \rangle\ \text{oc}\ \{\ x = 1 \wedge x = 2\ \}}$$

but this conclusion is not true: the postcondition should be $x = 3$!

Interference problem

$$S_1 \quad \{\, x = 0 \,\}\ \langle x := x + 1 \rangle\ \{\, x = 1 \,\}$$

$$S_2 \quad \{\, x = 0 \,\}\ \langle x := x + 2 \rangle\ \{\, x = 2 \,\}$$

- Execution of S_2 interferes with pre- and postconditions of S_1
  - The assertion $x = 0$ need not hold when S_1 starts execution
- Execution of S_1 interferes with pre- and postconditions of S_2
  - The assertion $x = 0$ need not hold when S_2 starts execution

$$S_1 \quad \{\, x = 0 \,\}\ \langle x := x + 1 \rangle\ \{\, x = 1 \,\}$$

$$S_2 \quad \{\, x = 0 \,\}\ \langle x := x + 2 \rangle\ \{\, x = 2 \,\}$$

- Execution of S_2 interferes with pre- and postconditions of S_1
  - The assertion $x = 0$ need not hold when S_1 starts execution
- Execution of S_1 interferes with pre- and postconditions of S_2
  - The assertion $x = 0$ need not hold when S_2 starts execution

Solution: weaken the assertions to account for the other process:

$$S_1 \quad \{\, x = 0 \vee x = 2 \,\}\ \langle x := x + 1 \rangle\ \{\, x = 1 \vee x = 3 \,\}$$

$$S_2 \quad \{\, x = 0 \vee x = 1 \,\}\ \langle x := x + 2 \rangle\ \{\, x = 2 \vee x = 3 \,\}$$

Interference problem

Apply the previous "parallel-composition-is-conjunction" rule again:

$$\frac{\{\, x = 0 \vee x = 2 \,\}\ \langle x := x + 1 \rangle\ \{\, x = 1 \vee x = 3 \,\} \qquad \{\, x = 0 \vee x = 1 \,\}\ \langle x := x + 2 \rangle\ \{\, x = 2 \vee x = 3 \,\}}{\{\, \mathit{PRE} \,\}\ \mathsf{co}\langle x := x + 1 \rangle \parallel \langle x := x + 2 \rangle\, \mathsf{oc}\ \{\, \mathit{POST} \,\}}$$

where:

PRE :

POST :

Apply the previous "parallel-composition-is-conjunction" rule again:

$$\frac{\{\ x = 0 \vee x = 2\ \}\ \langle x := x + 1\rangle\ \{\ x = 1 \vee x = 3\ \} \quad \{\ x = 0 \vee x = 1\ \}\ \langle x := x + 2\rangle\ \{\ x = 2 \vee x = 3\ \}}{\{\ \mathit{PRE}\ \}\ \mathsf{co}\langle x := x + 1\rangle \parallel \langle x := x + 2\rangle\,\mathsf{oc}\ \{\ \mathit{POST}\ \}}$$

where:

$$\begin{aligned} \mathit{PRE} &: (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1) \\ \mathit{POST} &: (x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3) \end{aligned}$$

Apply the previous "parallel-composition-is-conjunction" rule again:

$$\frac{\{\ x = 0 \vee x = 2\ \}\ \langle x := x + 1\rangle\ \{\ x = 1 \vee x = 3\ \} \quad \{\ x = 0 \vee x = 1\ \}\ \langle x := x + 2\rangle\ \{\ x = 2 \vee x = 3\ \}}{\{\ \mathit{PRE}\ \}\ \mathsf{co}\langle x := x + 1\rangle \parallel \langle x := x + 2\rangle\,\mathsf{oc}\ \{\ \mathit{POST}\ \}}$$

where:

$$\begin{aligned} \mathit{PRE} &: (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1) \\ \mathit{POST} &: (x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3) \end{aligned}$$

which gives:

$$\{\ x = 0\ \}\ \mathsf{co}\ \langle x := x + 1\rangle \parallel \langle x := x + 2\rangle\ \mathsf{oc}\ \{\ x = 3\ \}$$

Assume $\{ P_i \}\ S_i\ \{ Q_i \}$ for all $S_1, \ldots, S_n$

$$\frac{\{ P_i \}\ S_i\ \{ Q_i \} \quad \text{are } \textit{interference free}}{\{ P_1 \wedge \ldots \wedge P_n \}\ \text{co}\ S_1 \parallel \ldots \parallel S_n\ \text{oc}\ \{ Q_1 \wedge \ldots \wedge Q_n \}}\ \text{Co}$$

Assume $\{ P_i \}\ S_i\ \{ Q_i \}$ for all $S_1, \ldots, S_n$

$$\frac{\{ P_i \}\ S_i\ \{ Q_i \} \quad \text{are } \textit{interference free}}{\{ P_1 \wedge \ldots \wedge P_n \}\ \text{co}\ S_1 \parallel \ldots \parallel S_n\ \text{oc}\ \{ Q_1 \wedge \ldots \wedge Q_n \}}\ \text{Co}$$

Interference freedom

A process interferes with (the specification of) another process, if its execution changes the assertions[a] of the other process.

[a] Only "critical assertions" considered, i.e. outside mutex-protected sections

- Assertions inside awaits: not endangered
- Critical assertions or critical conditions: assertions outside await statement bodies and mutex-protected sections.

Interference freedom

- S: statement in some process, with pre-condition $pre(S)$
- C: critical assertion in another process
- S does not interfere with C, if

$$\vdash \{\ C \wedge pre(S)\ \}\ S\ \{\ C\ \}$$

is derivable in PL (= theorem).

"C is *invariant* under the execution of the other process"

Interference freedom

- S: statement in some process, with pre-condition $pre(S)$
- C: critical assertion in another process
- S does not interfere with C, if

$$\vdash \{\ C \wedge pre(S)\ \}\ S\ \{\ C\ \}$$

is derivable in PL (= theorem).

"C is *invariant* under the execution of the other process"

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ \mathrm{co}\ S_1 \parallel S_2\ \mathrm{oc}\ \{\ Q_1 \wedge Q_2\ \}}$$

Four interference freedom requirements (for 2 statements in co-oc):

Interference freedom

- S: statement in some process, with pre-condition $pre(S)$
- C: critical assertion in another process
- S does not interfere with C, if

$$\vdash \{\ C \wedge pre(S)\ \}\ S\ \{\ C\ \}$$

is derivable in PL (= theorem).

"C is *invariant* under the execution of the other process"

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ \mathrm{co}\ S_1 \parallel S_2\ \mathrm{oc}\ \{\ Q_1 \wedge Q_2\ \}}$$

Four interference freedom requirements (for 2 statements in co-oc):

$$\{P_2 \wedge P_1\}\ S_1\ \{P_2\} \qquad \{P_1 \wedge P_2\}\ S_2\ \{P_1\}$$
$$\{Q_2 \wedge P_1\}\ S_1\ \{Q_2\} \qquad \{Q_1 \wedge P_2\}\ S_2\ \{Q_1\}$$

"Avoiding" interference: Weakening assertions

$$
\begin{array}{ll}
S_1: & \{\, x = 0 \,\} < x := x + 1; > \{\, x = 1 \,\} \\
S_2: & \{\, x = 0 \,\} < x := x + 2; > \{\, x = 2 \,\}
\end{array}
$$

Here we have interference, for instance the precondition of S_1 is not maintained by the execution of S_2:

$$\{\, (x = 0) \wedge (x = 0) \,\}\ x := x + 2\ \{\, x = 0 \,\}$$

is not true

However, after weakening:

$$
\begin{array}{ll}
S_1: & \{\, x = 0 \vee x = 2 \,\}\ \langle x := x + 1 \rangle\ \{\, x = 1 \vee x = 3 \,\} \\
S_2: & \{\, x = 0 \vee x = 1 \,\}\ \langle x := x + 2 \rangle\ \{\, x = 2 \vee x = 3 \,\}
\end{array}
$$

$$S_1: \ \{ x = 0 \} < x := x + 1; > \{ x = 1 \}$$
$$S_2: \ \{ x = 0 \} < x := x + 2; > \{ x = 2 \}$$

Here we have interference, for instance the precondition of S_1 is not maintained by the execution of S_2:

$$\{ (x = 0) \wedge (x = 0) \} \, x := x + 2 \, \{ x = 0 \}$$

is not true

However, after weakening:

$$S_1: \ \{ x = 0 \vee x = 2 \} \langle x := x + 1 \rangle \{ x = 1 \vee x = 3 \}$$
$$S_2: \ \{ x = 0 \vee x = 1 \} \langle x := x + 2 \rangle \{ x = 2 \vee x = 3 \}$$

$$\{ (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1) \} \, x := x + 2 \, \{ x = 0 \vee x = 2 \}$$

(Correspondingly for the other three critical conditions)

Avoiding interference: Disjoint variables

- *V set:* global variables referred to (i.e. read or written) by a process
- *W set:* global variables written to by a process
- *Reference set:* global variables in critical (Hoare) assertions/conditions of one process

Avoiding interference: Disjoint variables

- *V set:* global variables referred to (i.e. read or written) by a process
- *W set:* global variables written to by a process
- *Reference set:* global variables in critical (Hoare) assertions/conditions of one process

S_1 and S_2: in 2 different processes. No interference, if:

- W set of S_1 is disjoint from reference set of S_2
- W set of S_2 is disjoint from reference set of S_1

Unfortunately: variables in a critical condition of one process will often be among the written variables of another

Global inductive invariants

- An assertion that only refers to global (shared) variables
- Holds initially.
- Preserved by all assignments/transitions (“inductive”)

Avoiding interference: Global invariants

Global inductive invariants

- An assertion that only refers to global (shared) variables
- Holds initially.
- Preserved by all assignments/transitions ("inductive")

"Separation of concerns": We avoid interference if critical conditions are on the form $\{ I \wedge L \}$ where:

- I is a global invariant
- L only refers to local variables of the considered process

Avoiding interference: Synchronization

- Hide critical conditions
- MUTEX to critical sections

$$\text{co} \ldots ; S; \ldots \parallel \ldots ; S_1; \{\ C\ \} S_2; \ldots \text{oc}$$

S might interfere with C
Hide the critical condition by a critical region:

$$\text{co} \ldots ; S; \ldots \parallel \ldots ; \langle S_1; \{\ C\ \} S_2 \rangle; \ldots \text{oc}$$

Example: Producer/ consumer synchronization

Let process `Producer` deliver data to a `Consumer` process

```
int buf, p := 0; c := 0;

process Producer {
  int a[N];...
  while (p < N) {
    < await (p = c) ; >
    buf := a[p];
      p := p+1;
 }
}
```

```
process Consumer {
int b[N];...
while (c < N) {
 < await (p > c) ; >
  b[c] := buf;
     c := c+1;
  }
}
```

Example: Producer/ consumer synchronization

Let process `Producer` deliver data to a `Consumer` process

$PC : c \leq p \leq c + 1 \wedge (p = c + 1 \Rightarrow buf = a[p - 1])$

PC a *global inductive invariant* of the producer/consumer?

```
                int buf, p := 0; c := 0;

process Producer {                    process Consumer {
  int a[N];...                        int b[N];...
  while (p < N) {                     while (c < N) {
    < await (p = c) ; >                < await (p > c) ; >
     buf := a[p];                        b[c] := buf;
       p := p+1;                           c := c+1;
  }                                     }
}                                     }
```

Example: Producer

Loop invariant of Producer:

$I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
                                  //  entering loop
  while (p < n) {
    < await (p = c); >

    buf := a[p];
    p := p + 1;
  }

}
```

Example: Producer

Loop invariant of Producer:

$I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
  { I_P }                          //  entering loop
  while (p < n) {
    < await (p = c); >

    buf := a[p];
    p := p + 1;
  }

}
```

Example: Producer

Loop invariant of Producer:
$I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
  { I_P }                          //  entering loop
  while (p < n) {                  { I_P ∧ p < n }
    < await (p = c); >

    buf := a[p];
    p := p + 1;
  }

}
```

Example: Producer

Loop invariant of Producer:
$I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
  { I_P }                          //  entering loop
  while (p < n) {                  { I_P ∧ p < n }
    < await (p = c); >             { I_P ∧ p < n ∧ p = c }

    buf := a[p];
    p := p + 1;
  }

}
```

Example: Producer

Loop invariant of Producer:
$I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
  { I_P }                          //  entering loop
  while (p < n) {                  { I_P ∧ p < n }
    < await (p = c); >             { I_P ∧ p < n ∧ p = c }

    buf := a[p];
    p := p + 1;                    {I_P}
  } {I_P ∧ ¬(p < n)}               //  exit loop
     (⇔ {PC ∧ p = n})
}
```

Example: Producer

Loop invariant of Producer:
$I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
  { I_P }                          //  entering loop
  while (p < n) {                  { I_P ∧ p < n }
    < await (p = c); >             { I_P ∧ p < n ∧ p = c }

    buf := a[p];                   { I_P[p + 1/p] }
    p := p + 1;                    {I_P}
  } {I_P ∧ ¬(p < n)}               //  exit loop
     (⇔ {PC ∧ p = n})
}
```

Example: Producer

Loop invariant of Producer:
$I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
  { I_P }                          //  entering loop
  while (p < n) {                  { I_P ∧ p < n }
    < await (p = c); >             { I_P ∧ p < n ∧ p = c }
                                   { I_P[p + 1/p][a[p]/buf] }
    buf := a[p];                   { I_P[p + 1/p] }
    p := p + 1;                    {I_P}
  } {I_P ∧ ¬(p < n)}               //  exit loop
     (⇔ {PC ∧ p = n})
}
```

Example: Producer

Loop invariant of Producer:
$I_P : PC \wedge p \leq n$

```
process Producer {
  int a[n];
  { I_P }                          //  entering loop
  while (p < n) {                  { I_P ∧ p < n }
    < await (p = c); >             { I_P ∧ p < n ∧ p = c }
                                   { I_P[p + 1/p][a[p]/buf] }
    buf := a[p];                   { I_P[p + 1/p] }
    p := p + 1;                    {I_P}
  } {I_P ∧ ¬(p < n)}               //  exit loop
     (⇔ {PC ∧ p = n})
}
```

Proof obligation:
$I_P \wedge p < n \wedge p = c \Rightarrow I_P[p + 1/p][a[p]/buf]$

Example: Consumer

Loop invariant of Consumer:

$I_C : PC \wedge c \leq n \wedge b[0 : c-1] = a[0 : c-1]$

```
process Consumer {
  int b[n];
                                  // entering loop
  while (c < n) {
    < await (p > c) ; >

    b[c] := buf;
    c := c + 1;
  }

}
```

Example: Consumer

Loop invariant of Consumer:

$I_C : PC \wedge c \leq n \wedge b[0 : c-1] = a[0 : c-1]$

```
process Consumer {
  int b[n];
  {I_C}                          // entering loop
  while (c < n) {
    < await (p > c) ; >

    b[c] := buf;
    c := c + 1;
  }

}
```

Example: Consumer

Loop invariant of Consumer:

$I_C : PC \wedge c \leq n \wedge b[0 : c-1] = a[0 : c-1]$

```
process Consumer {
  int b[n];
  {I_C}                          // entering loop
  while (c < n) {                {I_C ∧ c < n}
    < await (p > c) ; >

    b[c] := buf;
    c := c + 1;
  }

}
```

Example: Consumer

Loop invariant of Consumer:

$I_C : PC \wedge c \leq n \wedge b[0 : c-1] = a[0 : c-1]$

```
process Consumer {
  int b[n];
  {I_C}                          // entering loop
  while (c < n) {                {I_C ∧ c < n}
    < await (p > c) ; >          {I_C ∧ c < n ∧ p > c}

    b[c] := buf;
    c := c + 1;
  }

}
```

Example: Consumer

Loop invariant of Consumer:
$I_C : PC \wedge c \leq n \wedge b[0 : c-1] = a[0 : c-1]$

```
process Consumer {
  int b[n];
  {I_C}                          // entering loop
  while (c < n) {                {I_C ∧ c < n}
    < await (p > c) ; >          {I_C ∧ c < n ∧ p > c}

    b[c] := buf;
    c := c + 1;                  {I_C}
  } {I_C ∧ ¬(c < n)}             // exit loop
     (⇔ {PC ∧ c = n ∧ b[0 : c-1] = a[0 : c-1]})
}
```

Example: Consumer

Loop invariant of Consumer:
$I_C : PC \wedge c \leq n \wedge b[0 : c-1] = a[0 : c-1]$

```
process Consumer {
  int b[n];
  {I_C}                          // entering loop
  while (c < n) {                {I_C ∧ c < n}
    < await (p > c) ; >          {I_C ∧ c < n ∧ p > c}

    b[c] := buf;                 {I_C}[c + 1/c]
    c := c + 1;                  {I_C}
  } {I_C ∧ ¬(c < n)}             // exit loop
     (⇔ {PC ∧ c = n ∧ b[0 : c − 1] = a[0 : c − 1]})
}
```

Example: Consumer

Loop invariant of Consumer:
$I_C : PC \wedge c \leq n \wedge b[0 : c-1] = a[0 : c-1]$

```
process Consumer {
  int b[n];
  {I_C}                                  // entering loop
  while (c < n) {                        {I_C ∧ c < n}
    < await (p > c) ; >                  {I_C ∧ c < n ∧ p > c}
                                         {I_C[c + 1/c][buf/b[c]]}
    b[c] := buf;                         {I_C}[c + 1/c]
    c := c + 1;                          {I_C}
  } {I_C ∧ ¬(c < n)}                     // exit loop
     (⇔ {PC ∧ c = n ∧ b[0 : c − 1] = a[0 : c − 1]})
}
```

Example: Consumer

Loop invariant of Consumer:
$I_C : PC \wedge c \leq n \wedge b[0 : c-1] = a[0 : c-1]$

```
process Consumer {
  int b[n];
  {I_C}                                  // entering loop
  while (c < n) {                        {I_C ∧ c < n}
    < await (p > c) ; >                  {I_C ∧ c < n ∧ p > c}
                                         {I_C[c+1/c][buf/b[c]]}
    b[c] := buf;                         {I_C}[c+1/c]
    c := c + 1;                          {I_C}
  } {I_C ∧ ¬(c < n)}                     // exit loop
     (⇔ {PC ∧ c = n ∧ b[0 : c-1] = a[0 : c-1]})
}
```

Proof Obligation:

$I_C \wedge c < n \wedge p > c \;\Rightarrow\; I_C[c+1/c][buf/b[c]]$

Example: Producer/Consumer

Assume that the processes are interference free (argumentation on next slides).

The final state of the program satisfies:

$$PC \wedge p = n \wedge c = n \wedge b[0 : c - 1] = a[0 : c - 1]$$

which ensures that all elements in a are received and occur in the same order in b