Andrews Chapter 02

Semantics of concurrent programs

- **State:** the values of all the variables at a point in time.
- **Atomic action:** Action which indivisibly examine or change state.
- **History or interleaving or trace:** A particular sequence of the atomic actions of a program. Results in a particular sequence of states: $s_0 \rightarrow s_1 \rightarrow s_3 ... \rightarrow s_n$
- **Critical section:** A section that cannot be interleaved with other actions in other processes that reference the same variables.

Properties of concurrent programs

- **Property:** Something which is true of every possible history of a program.
  - **Safety properties:** program never enters a bad state. Example: Mutual exclusion.
  - **Liveness properties:** program eventually enters a good state. Example: Eventual entry into a critical section (no deadlock).
- **Partial correctness:** final state is correct, assuming the program terminates. A safety property.
- **Termination:** every loop and procedure call terminates. All histories are finite. A liveness property
- **Total correctness:** partially correct and terminates.

Demonstrating properties of concurrent programs

- Testing and debugging.
- Operational reasoning.
- m threads executing n atomic actions each:

$$\frac{(mn)!}{(m!)^n}$$

m	n	
3	1	6
3	2	90
3	3	1680
3	4	34650
3	5	756756
3	6	17153136
3	7	399072960
3	8	9465511770

m	n	
4	1	24
4	2	2520
4	3	369600
4	4	63063000
4	5	11732745024
4	6	2308743493056
4	7	472518347558400
4	8	99561092450391000

Assertional reasoning

- Axiomatic semantics
- Use *assertions* to characterize sets of states:
  - All states in which the assertion is true.
- Atomic actions are *predicate transformers*.
- Prove that bad states can't happen.
- Can statically analyze programs.
- Very difficult to use correctly.
- Best to combine with testing and operational reasoning.

```
string line;
read a line of input from stdin into line;
while (!EOF) {    # EOF is end of file
   look for pattern in line;
   if (pattern is in line)
      write line;
   read next line of input;
}
```

Finding Patterns: Sequential Program

Independence of parallel processes

- **Read set:** variables read by a part of a program.
- **Write set:** variables written to by a part of a program (and maybe read).
- Two parts are **independent** if the write set of each part is disjoint from both the read and write sets of the other part.
- Two parts are independent if both only read shared variables, or each part reads different variables than the ones written into by the other part.
- Which parts of the pattern finding program are independent?

```
string line;
read a line of input from stdin into line;
while (!EOF) {
  co look for pattern in line;
     if (pattern is in line)
        write line;
  // read next line of input into line;
  oc;
}
```

Finding Patterns: Concurrent Program with Interference

```
string line1, line2;
read a line of input from stdin into line1;
while (!EOF) {
  co look for pattern in line1;
     if (pattern is in line1)
        write line1;
  // read next line of input into line2;
  oc;
}
```

Finding Patterns: Disjoint Processes

```
string line1, line2;
read a line of input from stdin into line1;
while (! EOF) {
  co look for pattern in line1;
     if (pattern is in line1)
        write line1;
  // read next line of input into line2;
  oc;
  line1 = line2;
}
```

Finding Patterns: Co Inside While

```
string buffer;  # contains one line of input
bool done = false;  # used to signal termination
co  # process 1:  find patterns
  string line1;
  while (true) {
    wait for buffer to be full or done to be true;
    if (done) break;
    line1 = buffer;
    signal that buffer is empty;
    look for pattern in line1;
    if (pattern is in line1)
      write line1;
  }
//  # process 2:  read new lines
  string line2;
  while (true) {
    read next line of input into line2;
    if (EOF) {done = true; break; }
    wait for buffer to be empty;
    buffer = line2;
    signal that buffer is full;
  }
oc;
```

Figure 2.1 Finding patterns in a file.

Maximum of an array

Sequential:

```
int m = 0;
for [i = 0 to n-1]
  if (a[i] > m)
    m = a[i];
```

Incorrect but fast:

```
int m = 0;
co [i = 0 to n-1] {
  if (a[i] > m)
    m = a[i];
}
```

Correct but slow:

```
int m = 0;
co [i = 0 to n-1] {
  ⟨ if (a[i] > m)
    m = a[i];⟩
}
```

Pointless:

```
int m = 0;
co [i = 0 to n-1] {
  if (a[i] > m)
    ⟨ m = a[i];⟩
}
```

Correct and fast:

```
int m = 0;
co [i = 0 to n-1] {
  if (a[i] > m)
    ⟨ if (a[i] > m)
      m = a[i];⟩
}
```

Atomic actions and await statements

```
int y = 0, z = 0;
co x = y+z; // y = 1; z = 2; oc;
```

- Final value of `x` could be 0,1,2, or 3
- There is *never* a state in which `y+z` is 2

Assumptions for our machines

- Values of basic types are stored in memory elements that are read and written as atomic actions.
- Values are manipulated by loading into registers, operating on them, and storing results back into memory.
- Each process has its own set of registers. (Either separate cores, dedicated registers, or a *context switch*.
- Intermediate results when a complex expression is evaluated are stored in registers or private memory (e.g. a private stack).

Appearance of atomicity

- If evaluating expression `e`, one process does not reference a variable altered by another process, expression evaluation will appear to be atomic.
  - None of the values on which `e` depends could change during the evaluation
  - No other process can see any temporary values that might be created while the expression is evaluated
- If an assignment `x = e` does not reference any variable altered by another process then the assignment will appear atomic.

At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by another process.
- Assume a critical reference is to a simple variable.
- An assignment `x = e` is **at-most-once** if either
  - `e` contains at most one critical reference and `x` is not read by another process
  - `e` contains no critical references
- Such an assignment statement will appear to be atomic.

Examples

No critical references:

```
int x = 0, y = 0;
co x = x+1; // y = y+1; oc;
```

`x` and `y` are both 1

One critical reference and one lhs read by another:

```
int x = 0, y = 0;
co x = y+1; // y = y+1; oc;
```

`x` is 1 or 2, `y` is 1

Neither satisfies at-most-once:

```
int x = 0, y = 0;
co x = y+1; // y = x+1; oc;
```

Values could be 1 and 2, 2 and 1, or even 1 and 1

The Await Statement

- `<await (B) S;>`
- `B` specifies a delay condition.
- `S` is a sequence of statements guaranteed to terminate.
- Atomic actions are specified by angle brakets, `<` and `>`
- Therefore `B` is guaranteed to be true when execution of `S` begins.
- No internal state in `S` is visible to other processes.
- What is this?

```
<await (s > 0) s = s - 1;>
```

Await is very powerful

- Await is convenient to use.
- Await is expensive to implement.
- Mutual exclusion:

```
<S;>
```

- For example: `< x = x+1; y = y+1; >`
- If `S` is a single assignment that is at-most-once, then `<S;>` is the same as `S;`
- Condition synchronization:

```
< await (B); >
```

- Example, delay until `count > 0`:

```
<await (count > 0);>
```

- If `B` is at-most-once, then `< await (B); >` can be implemented as:

```
while (not B);
```

Atomic actions

- An **unconditional** atomic action is one that does not contain a delay condition `B`.
- A **conditional** atomic action is an `await` statement with a guard `B`.
- If `B` is false, it can only become true as the result of actions by other processes.

```
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;
  }
}
```

Figure 2.2 Copying an array from a producer to a consumer.

Formal Logic

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$
$true$	$true$	$true$	$true$	$true$
$true$	$false$	$false$	$true$	$false$
$false$	$true$	$false$	$true$	$true$
$false$	$false$	$false$	$false$	$true$

Logic: equivalences

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$
$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

$$\neg\neg A \equiv A$$
$$A \vee A \equiv A$$
$$A \vee \neg A \equiv true$$
$$A \vee true \equiv true$$
$$A \vee false \equiv A$$
$$A \wedge true \equiv A$$
$$A \wedge false \equiv false$$
$$A \wedge A \equiv A$$
$$A \wedge \neg A \equiv false$$
$$A \Rightarrow true \equiv true$$

$$A \Rightarrow false \equiv \neg A$$
$$true \Rightarrow A \equiv A$$
$$false \Rightarrow A \equiv true$$
$$A \Rightarrow A \equiv true$$
$$A \Rightarrow B \equiv \neg A \vee B$$
$$A \Rightarrow B \equiv \neg B \Rightarrow \neg A$$
$$\neg(A \Rightarrow B) \equiv A \wedge \neg B$$
$$A \wedge (A \vee B) \equiv A$$
$$A \vee (A \wedge B) \equiv A$$
$$A \wedge (\neg A \vee B) \equiv A \wedge B$$
$$A \vee (\neg A \wedge B) \equiv A \vee B$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

Logic: inference rules

- Modus Ponens

$$\frac{A \Rightarrow B, A}{B}$$

- Modus Tollens

$$\frac{A \Rightarrow B, \neg B}{\neg A}$$

- Conjunction

$$\frac{A, B}{A \wedge B}$$

- Simplification

$$\frac{A \wedge B}{A}$$

- Addition

$$\frac{A}{A \vee B}$$

- Disjunctive syllogism

$$\frac{A \vee B, \neg A}{B}$$

- Hypothetical syllogism

$$\frac{A \Rightarrow B, B \Rightarrow C}{A \Rightarrow C}$$

- Constructive dilemma

$$\frac{A \vee B, A \Rightarrow C, B \Rightarrow D}{C \vee D}$$

- Destructive dilemma

$$\frac{\neg C \vee \neg D, A \Rightarrow C, B \Rightarrow D}{\neg A \vee \neg B}$$

Proof strategies

- **Conditional proof:** To prove something like $X \Rightarrow Y$:
  - Assume X is true on one line of a proof.
  - Show that Y follows on some later line.
- **Indirect proof:** To prove something like X:
  - Assume $\neg X$ on one line of a proof.
  - Show that a contradiction (Z and $\neg Z$) shows up on later lines.

Example conditional proof

Prove: $P \Rightarrow (Q \Rightarrow (P \wedge Q))$

1.	P	assumption for conditional proof
2.	...	
3.	...	
4.	$Q \Rightarrow (P \wedge Q)$	somehow...
5.	$P \Rightarrow (Q \Rightarrow (P \wedge Q))$	from lines 1 and 4

Now we need to find a proof of $Q \Rightarrow (P \wedge Q)$

Example conditional proof

Prove: $P \Rightarrow (Q \Rightarrow (P \wedge Q))$

We use a *nested proof*:

1.	P	assumption for conditional proof
2.	$\quad Q$	assumption for conditional proof
3.	$\quad$ ...	
4.	$\quad$ ...	
5.	$\quad P \wedge Q$	somehow...
6.	$Q \Rightarrow (P \wedge Q)$	from lines 2 and 5
7.	$P \Rightarrow (Q \Rightarrow (P \wedge Q))$	from lines 1 and 6

But line 5 follows immediately from lines 1 and 2 by conjunction! We can eliminate lines 3 and 4.

Example conditional proof

Prove: $P \Rightarrow (Q \Rightarrow (P \wedge Q))$

1.	P	assumption for conditional proof
2.	$\quad Q$	assumption for conditional proof
3.	$\quad P \wedge Q$	from lines 1 and 2 by conjunction
4.	$Q \Rightarrow (P \wedge Q)$	from lines 2 and 3
5.	$P \Rightarrow Q \Rightarrow (P \wedge Q)$	from lines 1 and 4

Example indirect proof

Prove: $A \Rightarrow (B \Rightarrow A)$

1.	$\neg(A \Rightarrow (B \Rightarrow A))$	assumption for indirect proof
2.	...	
3.	Z	somehow
4.	...	
5.	$\neg Z$	somehow
6.	...	
7.	$A \Rightarrow (B \Rightarrow A)$	from 1, 3, and 5

Z can be anything at all, so we have more freedom with this method.

Example indirect proof

Prove: $A \Rightarrow (B \Rightarrow A)$

1.	$\neg(A \Rightarrow (B \Rightarrow A))$	assumption for indirect proof
2.	$A \wedge \neg(B \Rightarrow A)$	equiv to line 1
3.	A	simplification from line 2
4.	$\neg(B \Rightarrow A)$	simplification from line 2
5.	$B \wedge \neg A$	equiv to line 4
6.	B	simplification from line 5
7.	$\neg A$	simplification from line 5
8.	...	
9.	Z	somehow
10.	...	
11.	$\neg Z$	somehow
12.	...	
13.	$A \Rightarrow (B \Rightarrow A)$	from 1, 9, and 11

Now what?

Example indirect proof

Prove: $A \Rightarrow (B \Rightarrow A)$

1.	$\neg(A \Rightarrow (B \Rightarrow A))$	assumption for indirect proof
2.	$A \wedge \neg(B \Rightarrow A)$	equiv to line 1
3.	A	simplification from line 2
4.	$\neg(B \Rightarrow A)$	simplification from line 2
5.	$B \wedge \neg A$	equiv to line 4
6.	B	simplification from line 5
7.	$\neg A$	simplification from line 5
8.	$A \Rightarrow (B \Rightarrow A)$	from 1, 3, and 7

Example proof

Prove: $(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$

Ideas?

Example proof

Prove: $(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$

1.	$A \Rightarrow C$	assumption for conditional proof
2.	...	
3.	$A \Rightarrow (B \vee C)$	
4.	$(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$	from 1 and 3

Ideas?

Example proof using both conditional and indirect

Prove: $(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$

1.	$A \Rightarrow C$	assumption for conditional proof
2.	$\quad A$	assumption for conditional proof
3.	$\quad ...$	
4.	$\quad B \vee C$	
5.	$A \Rightarrow (B \vee C)$	from 2 and 4
6.	$(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$	from 1 and 5

Ideas?

Example proof using both conditional and indirect

Prove: $(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$

1.	$A \Rightarrow C$	assumption for conditional proof
2.	$\quad A$	assumption for conditional proof
3.	$\quad\quad \neg(B \vee C)$	assumption for indirect
4.	$\quad\quad ...$	
5.	$\quad B \vee C$	from 3 and ? and ?
6.	$A \Rightarrow (B \vee C)$	from 2 and 4
7.	$(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$	from 1 and 5

Ideas?

Example proof using both conditional and indirect

Prove: $(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$

1.	$A \Rightarrow C$	assumption for conditional proof
2.	$\quad A$	assumption for conditional proof
3.	$\quad\quad \neg(B \vee C)$	assumption for indirect
4.	$\quad\quad \neg B \wedge \neg C$	equiv of 3
5.	$\quad\quad \neg B$	simplification of 4
6.	$\quad\quad \neg C$	simplification of 4
7.	$\quad\quad ...$	
8.	$\quad B \vee C$	from 3 and ? and ?
9.	$A \Rightarrow (B \vee C)$	from 2 and 8
10.	$(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$	from 1 and 9

Ideas?

Example proof using both conditional and indirect

Prove: $(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$

1.	$A \Rightarrow C$	assumption for conditional proof
2.	$\quad A$	assumption for conditional proof
3.	$\quad\quad \neg(B \vee C)$	assumption for indirect
4.	$\quad\quad \neg B \wedge \neg C$	equiv of 3
5.	$\quad\quad \neg B$	simplification of 4
6.	$\quad\quad \neg C$	simplification of 4
7.	$\quad\quad C$	1 and 2 and modus ponens
8.	$\quad B \vee C$	from 3 and 6 and 7
9.	$A \Rightarrow (B \vee C)$	from 2 and 8
10.	$(A \Rightarrow C) \Rightarrow (A \Rightarrow (B \vee C))$	from 1 and 9

Composition Rule:

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

If Statement Rule:

$$\frac{\{P \wedge B\}\ S\ \{Q\},\ (P \wedge \neg B) \Rightarrow Q}{\{P\}\ \text{if (B) } S;\ \{Q\}}$$

While Statement Rule:

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{while(B) } S;\ \{I \wedge \neg B\}}$$

Rule of Consequence:

$$\frac{P' \Rightarrow P,\ \{P\}\ S\ \{Q\},\ Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

Figure 2.3 Inference rules in programming logic *PL*.

Axiomatic Semantics

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

- If P is true before S is executed, and S terminates, then Q is true after the execution of S.

Axiomatic Semantics

- Assignment Axiom

$$\{Q(x/t)\}\ \texttt{x := t}\ \{Q\}$$

- Composition

$$\frac{\{P\}\ \texttt{S1}\ \{R\}, \{R\}\ \texttt{S2}\ \{Q\}}{\{P\}\ \texttt{S1;S2}\ \{Q\}}$$

- Consequence

$$\frac{P \Rightarrow R, \{R\}\ \texttt{S}\ \{Q\}}{\{P\}\ \texttt{S}\ \{Q\}}$$

$$\frac{\{P\}\ \texttt{S}\ \{T\}, T \Rightarrow Q}{\{P\}\ \texttt{S}\ \{Q\}}$$

- If-then

$$\frac{\{P \wedge C\}\ \texttt{S}\ \{Q\}, P \wedge \neg C \Rightarrow Q}{\{P\}\ \texttt{if C then S}\ \{Q\}}$$

- If-then-else

$$\frac{\{P \wedge C\}\ \texttt{S1}\ \{Q\}, \{P \wedge \neg C\}\ \texttt{S2}\ \{Q\}}{\{P\}\ \texttt{if C then S1 else S2}\ \{Q\}}$$

- While

$$\frac{\{P \wedge C\}\ \texttt{S}\ \{P\}}{\{P\}\ \texttt{while C do S}\ \{P \wedge \neg C\}}$$

Proofs in Axiomatic Semantics

Prove: $\{x < 5\}$ `x := x + 1` $\{x < 7\}$

1. $\{x + 1 < 7\}$ `x := x + 1` $\{x < 7\}$	Axiom
2. $x < 5$	Assumption
3. $x + 1 < 6$	2, arithmetic
4. $6 < 7$	arithmetic
5. $x + 1 < 7$	3,4, transitive
6. $(x < 5) \Rightarrow (x + 1 < 7)$	2,5, conditional proof
7. $\{x < 5\}$ `x := x + 1` $\{x < 7\}$	1,6, consequence

$$\{Q(x/t)\}\ \texttt{x := t}\ \{Q\}$$

Proofs in Axiomatic Semantics

Prove: $\{true\}$ `if x < 0 then x := -x` $\{x \geq 0\}$

1.	$\{-x \geq 0\}$ `x := -x` $\{x \geq 0\}$	Axiom
2.	$true \wedge (x < 0)$	assumption
3.	$x < 0$	2, simplification
4.	$-x > 0$	3, arithmetic
5.	$-x \geq 0$	4, arithmetic
6.	$true \wedge (x < 0) \Rightarrow (-x \geq 0)$	2,5, conditional proof
7.	$\{true \wedge (x < 0)\}$ `x := -x` $\{x \geq 0\}$	1,6, consequence
8.	$true \wedge \neg(x < 0)$	assumption
9.	$\neg(x < 0)$	8, simplification
10.	$x \geq 0$	9, arithmetic
11.	$true \wedge \neg(x < 0) \Rightarrow (x \geq 0)$	8,10, conditional proof
12.	$\{true\}$ `if x < 0 then x := -x` $\{x \geq 0\}$	7,11,if-then

$$\frac{\{P \wedge C\}\ \texttt{S}\ \{Q\}, P \wedge \neg C \Rightarrow Q}{\{P\}\ \texttt{if C then S}\ \{Q\}}$$

Proofs in Axiomatic Semantics

$\{true\}$
```
m := x;
```
$\{m = x\}$
```
if (y > m)
  m := y;
```
$\{(m = x) \wedge m \geq y) \vee (m = y \wedge m > x)\}$

$$\frac{\{P \wedge C\}\ \texttt{S}\ \{Q\}, P \wedge \neg C \Rightarrow Q}{\{P\}\ \texttt{if C then S}\ \{Q\}}$$

Proofs in Axiomatic Semantics

$\{true\}$
```
i := 1;
```
$\{i = 1 \land \forall\, j : 1 \leq j < i : a[j] \neq x)\}$
```
while (a[i] != x)
```
  $\{(\forall\, j : 1 <= j < i : a[j] \neq x)\}$
```
  i := i+1;
```
$\{(\forall\, j : 1 \leq j < i : a[j] \neq x) \land a[i] = x\}$

$$\frac{\{P \land C\}\ \texttt{S}\ \{P\}}{\{P\}\ \texttt{while C do S}\ \{P \land \neg C\}}$$

Proofs in Axiomatic Semantics

$\{(a > 0) \wedge (b \geq\ 0)\}$
```
i := 0;
p := 1;
```
$\{P\} = \{(p = a\hat{}i) \wedge (i \leq\ b)\}$
```
while i < b do
  p := p*a;
  i := i+1;
```
$\{(P \wedge \neg\ C)\} = \{(p = a\hat{}i) \wedge (i \leq\ b) \wedge \neg(i < b)\}$
$\{p = a\hat{}b\}$

$$\frac{\{P \wedge C\}\ \texttt{S}\ \{P\}}{\{P\}\ \texttt{while C do S}\ \{P \wedge \neg C\}}$$

Await Statement Rule:

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \langle \texttt{await (B) S;} \rangle\ \{Q\}}$$

Co Statement Rule:

$$\frac{\{P_i\}\ S_i\ \{Q_i\} \text{ are interference free}}{\begin{array}{l}\{P_1 \wedge \ldots \wedge P_n\} \\ \texttt{co } S_1\texttt{; // ... // } S_n\texttt{; oc} \\ \{Q_1 \wedge \ldots \wedge Q_n\}\end{array}}$$

Inference Rules for Await and Co Statements

Sematics of Concurrent Execution

- Await rule

$$\frac{\{P \wedge B\}\ \texttt{S}\ \{Q\}}{\{P\}\ \langle\texttt{await (B) S;}\rangle\ \{Q\}}$$

- Co rule

$$\frac{\{P_i\}\ \texttt{Si}\ \{Q_i\} \text{ are interference free}}{\{P_1 \wedge ... \wedge P_n\}\ \texttt{co S1; // ... // Sn; oc}\ \{Q_1 \wedge ... \wedge Q_n\}}$$

- One process **interferes** with another if it executes an assignment that invalidates an assertion in the other process.

Co example

- $\{x = 0\}$
  `co` $\langle$ `x := x+1;` $\rangle$ `//` $\langle$ `x := x+2;` $\rangle$ `oc`
  $\{x = 3\}$

- Neither process can assume that $x = 0$ when they start.
- But the following can be assumed:

- $\{x = 0\}$
  `co`
  $\{x = 0 \vee x = 2\}$
  $\langle$ `x := x + 1;` $\rangle$
  $\{x = 1 \vee x = 3\}$
  `//`
  $\{x = 0 \vee x = 1\}$
  $\langle$ `x := x + 2;` $\rangle$
  $\{x = 2 \vee x = 3\}$
  `oc`
  $\{ x = 3 \}$

Definition of Noninterference

- An **assignment action** is an assignment statement or an `await` statement that contains one or more assignments.
- A **critical assertion** is a precondition or postcondition that is not within an `await` statement.
- Rename all local variables in all processes so that no two processes have the same local variable names.
- Let `a` be an assignment action in one process and let $pre(a)$ be its precondition. Let C be a critical assertion in another process. Then `a` **does not interfere** with C if the following is a theorem:

$$\{C \wedge pre(a)\}\ \texttt{a}\ \{C\}$$

Example of noninterference

- $\{x = 0\}$
  co
  $\{x = 0 \lor x = 2\}$
  $\langle$ x := x + 1; $\rangle$
  $\{x = 1 \lor x = 3\}$
  //
  $\{x = 0 \lor x = 1\}$
  $\langle$ x := x + 2; $\rangle$
  $\{x = 2 \lor x = 3\}$
  oc
  $\{ x = 3 \}$

- The precondition of the first process is a critical assertion.
- It is not interfered with by the second process because:

- $\{(x = 0 \lor x = 2) \land (x = 0 \lor x = 1)\}$
  x := x+2;
  $\{x = 0 \lor x = 2\}$

Techniques for avoiding interference: disjoint variables

- **Read set:** set of variables read by a process.
- **Write set:** set of variables written to by a process.
- **Reference set:** set of variables that appear in the assertions of a proof of that process.
- If the write set of one process is disjoint from the reference set of a second, and vice versa, the two processes cannot interfere.
- Example:
- `co x := x+1; // y := y+1; oc`
- Both of the following are theorems:
- $\{x = 0\}$ `x := x + 1;` $\{x = 1\}$
- $\{y = 0\}$ `y := y + 1;` $\{y = 1\}$
- Each statement contains one assignment and two assertions, so there are four noninterference theorems to prove, but they are all trivial.

Techniques for avoiding interference: weakened assertions

- A **weakened assertion** is one that admits more program states than another assertion that might be true of a process in isolation.
- Example:
- $\{x = 0\}$
  co
  $\{x = 0 \lor x = 2\}$
  $\langle$ `x := x + 1;` $\rangle$
  $\{x = 1 \lor x = 3\}$
  //
  $\{x = 0 \lor x = 1\}$
  $\langle$ `x := x + 2;` $\rangle$
  $\{x = 2 \lor x = 3\}$
  oc
  $\{ x = 3 \}$

Techniques for avoiding interference: global invariants

- If I is a predicate that references global variables, then I is a **global invariant** with respect to a set of processes if:
  1. I is true when the processes begin, and
  2. I is preserved by every assignment action.

- Suppose I is a global invariant, and every critical assertion has the form $I \wedge L$, where L is only about local variables, then all processes will be interference free.
- Checking this is a *linear* process, as opposed to the *exponential* number of program histories.

Techniques for avoiding interference: synchronization

- Statements within `await` statements appear indivisible to other processes.
- Hence we can ignore them when considering interference, and consider only *entire* sequences.
- For example, in the statement

  $\langle$`x := x+1; y := y+1;` $\rangle$

  we don't have to consider each statement by itself.

```
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;
  }
}
```

Figure 2.2 Copying an array from a producer to a consumer.

```
int buf, p = 0, c = 0;
{PC: c <= p <= c+1 ∧ a[0:n-1] == A[0:n-1] ∧
          (p == c+1) ⇒ (buf == A[p-1])}

process Producer {
  int a[n];    # assume a[i] is initialized to A[i]
  {IP: PC ∧ p <= n}
  while (p < n) {
    {PC ∧ p < n}
    ⟨await (p == c);⟩   # delay until buffer empty
    {PC ∧ p < n ∧ p == c}
    buf = a[p];
    {PC ∧ p < n ∧ p == c ∧ buf == A[p]}
    p = p+1;
    {IP}
  }
  {PC ∧ p == n}
}

process Consumer {
  int b[n];
  {IC: PC ∧ c <= n ∧ b[0:c-1] == A[0:c-1]}
  while (c < n) {
    {IC ∧ c < n}
    ⟨await (p > c);⟩    # delay until buffer full
    {IC ∧ c < n ∧ p > c}
    b[c] = buf;
    {IC ∧ c < n ∧ p > c ∧ b[c] == A[c]}
    c = c+1;
    {IC}
  }
  {IC ∧ c == n}
}
```

Figure 2.4 Proof outline for the array copy program.

Example illustrates all four techniques

- Many statements and parts are disjoint.
- Weakened assertions, e.g. `buf == A[p-1]` but only when `p == c+1`
- Global invariant PC
- Synchronization using `await`

Safety and Liveness Properties

- **Safety:** nothing bad ever happens.
  - Final state is correct.
  - Mutual exclusion.
  - No deadlock.
- **Liveness:** something good eventually happens.
  - Program terminates
  - Process eventually enters critical section.
  - A request for service will eventually be honored.
  - A message will reach its destination.

Proving Safety Properties

- Show that BAD is false in every state.
- Show that $GOOD$ is true in every state.

Proving Safety Properties: exclusion of configurations

```
co # process 1
  ...;  { pre(S1) } S1; ...
// # process 2
  ...;  { pre(S2) } S2; ...
oc
```

- Suppose the preconditions do not interfere.
- Suppose `pre(S1)` $\wedge$ `pre(S2) == false`.
- Then the two processes cannot be at these statements at the same time!
- `false` means that the program is *never* in this state.
- In the array copy example, both processes cannot simultaneously be delayed in their `await` statements.

```
int buf, p = 0, c = 0;
{PC: c <= p <= c+1 ∧ a[0:n-1] == A[0:n-1] ∧
          (p == c+1) ⇒ (buf == A[p-1])}

process Producer {
  int a[n];    # assume a[i] is initialized to A[i]
  {IP: PC ∧ p <= n}
  while (p < n) {
    {PC ∧ p < n}
    ⟨await (p == c);⟩   # delay until buffer empty
    {PC ∧ p < n ∧ p == c}
    buf = a[p];
    {PC ∧ p < n ∧ p == c ∧ buf == A[p]}
    p = p+1;
    {IP}
  }
  {PC ∧ p == n}
}

process Consumer {
  int b[n];
  {IC: PC ∧ c <= n ∧ b[0:c-1] == A[0:c-1]}
  while (c < n) {
    {IC ∧ c < n}
    ⟨await (p > c);⟩    # delay until buffer full
    {IC ∧ c < n ∧ p > c}
    b[c] = buf;
    {IC ∧ c < n ∧ p > c ∧ b[c] == A[c]}
    c = c+1;
    {IC}
  }
  {IC ∧ c == n}
}
```

Figure 2.4 Proof outline for the array copy program.

Scheduling and Fairness

- **Fairness:** each process gets a chance to proceed.
- An atomic action is **eligible** if it is the next one in the process that could be executed.
- A **scheduling policy** determines which eligible action will be executed next.

Scheduling and Fairness

- ```
  bool continue = true;
  co while (continue);
  // continue = false;
  oc
  ```

- If the first process always goes first, this program will never terminate.
  ```

Fairness

- **Unconditional fairness:** if every unconditional atomic action that is eligible is executed eventually.
- **Weak fairness:**
  1. it is unconditionally fair, and
  2. every conditional atomic action that is eligible is executed eventually, assuming that its condition becomes true and then remains true until it is seen by the process executing the conditional atomic action.
- In other words, assuming that if ⟨`await (B) S;`⟩ is eligible and `B` becomes true, then `B` remains true until after `S`.
- Round-robin and time slicing are weakly fair.
- Not sufficient to guarantee that any eligible `await` statement eventually executes.

Fairness

- **Strong fairness:**
  1. it is unconditionally fair, and
  2. every conditional atomic action that is eligible is executed eventually, assuming that its condition is infinitely often true.
- Such a scheduling policy cannot happen to select an action only when the condition is false.

Difference between strong and weak fairness

```
bool continue = true, try = false;

co while (continue) {try = true; try = false;}
// ⟨ await (try) continue = false; ⟩
oc
```

- With strong fairness, this program will eventually terminate.
- With weak fairness, this program might not terminate, even though `try` becomes true infinitely often.
- It is impossible to devise a scheduler that is both practical and strongly fair.

Fairness in the array copy example

- The example is deadlock free, even with weak fairness:
  - When one process makes the delay condition of the other true, it stays true until after the other process continues.
  - Each `await` statement has the form $\langle$`await(B);`$\rangle$, and `B` refers to only one variable altered by the other process.
  - Both `await` statements could be implemented with busy loops:

$$\langle \texttt{await (p == c)} \rangle \equiv \texttt{while (p != c)}$$

- This program will terminate only on unconditional fairness, since there are no more conditional atomic actions.

```
int buf, p = 0, c = 0;
{PC: c <= p <= c+1 ∧ a[0:n-1] == A[0:n-1] ∧
          (p == c+1) ⇒ (buf == A[p-1])}

process Producer {
  int a[n];    # assume a[i] is initialized to A[i]
  {IP: PC ∧ p <= n}
  while (p < n) {
    {PC ∧ p < n}
    ⟨await (p == c);⟩   # delay until buffer empty
    {PC ∧ p < n ∧ p == c}
    buf = a[p];
    {PC ∧ p < n ∧ p == c ∧ buf == A[p]}
    p = p+1;
    {IP}
  }
  {PC ∧ p == n}
}

process Consumer {
  int b[n];
  {IC: PC ∧ c <= n ∧ b[0:c-1] == A[0:c-1]}
  while (c < n) {
    {IC ∧ c < n}
    ⟨await (p > c);⟩    # delay until buffer full
    {IC ∧ c < n ∧ p > c}
    b[c] = buf;
    {IC ∧ c < n ∧ p > c ∧ b[c] == A[c]}
    c = c+1;
    {IC}
  }
  {IC ∧ c == n}
}
```

Figure 2.4 Proof outline for the array copy program.

Livelock

- If all busy waiting loops spin forever, the program is said to suffer **livelock**.
- The program is alive, but the processes are not going anywhere.
- Like two people passing each other in a hallway, each constantly moving to the other side.
- Absence of livelock is a safety property.