Q: Which of these programs (pseudocode-ish) creates and launches immediately `someNum` of threads, and also times the concurrent execution of those threads?
Atomic action: Makes an *indivisible* state transformation. Any intermediate state that might exist in the implementation of the action must NOT be visible to other processes.
Atomic action: Makes an indivisible state transformation. Any intermediate state that might exist in the implementation of the action must NOT be visible to other processes.

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

When executed concurrently, the assignment statements might be implemented by a sequence of fine-grained machine instructions, and depending on the execution history one of several (potentially many) different results may result.

Q: What are the possible final values of x if the code is run concurrently?
Atomic action: Makes an **indivisible** state transformation. Any intermediate state that might exist in the implementation of the action must **NOT** be visible to other processes.

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

When executed concurrently, the assignment statements might be implemented by a sequence of fine-grained machine instructions, and depending on the execution history one of several (potentially many) different results may result.

Q: **What are the possible final values of x if the code is run concurrently?**

0, 1, 2 or 3
Review: Atomicity

**Atomic action:** Makes an *indivisible* state transformation. Any intermediate state that might exist in the implementation of the action must NOT be visible to other processes.

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

**Q:** How might $x=2$ be the final state?

**Q:** What are the possible final values of $x$ if the code is run concurrently? **0, 1, 2 or 3**
Atomic action: Makes an **indivisible** state transformation. Any intermediate state that might exist in the implementation of the action must **NOT** be visible to other processes.

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

Keep in mind that \( x=y+z \) is itself a series of instructions:

- \( i_1 \): fetch \( y \)
- \( i_2 \): fetch \( z \)
- \( i_3 \): compute \( x=y+z \)
- \( i_4 \): write back \( x \)

Q: Assuming the following:

- \( i_5 \): \( y=1 \)
- \( i_6 \): \( z=2 \)

What is an instruction history such that \( x \) has a final value of 2?

Q: How might \( x=2 \) be the final state?

0, 1, \( \boxed{2} \) or 3
From last time

Logical axioms

\[
\begin{align*}
A \land (B \lor C) & \equiv (A \land B) \lor (A \land C) \\
A \lor (B \land C) & \equiv (A \lor B) \land (A \lor C) \\
\neg A & \equiv A \\
A \lor A & \equiv A \\
A \lor \neg A & \equiv \text{true} \\
A \lor \text{true} & \equiv \text{true} \\
A \lor \text{false} & \equiv \text{false} \\
A \land \text{true} & \equiv A \\
A \land \text{false} & \equiv \text{false} \\
A \implies \text{true} & \equiv \text{true} \\
A \implies \text{false} & \equiv \neg A \\
A \implies A & \equiv \text{true} \\
A \implies B & \equiv \neg A \lor B \\
\neg (A \implies B) & \equiv A \land \neg B \\
A \land (A \lor B) & \equiv A \\
A \lor (A \land B) & \equiv A \\
A \land (\neg A \lor B) & \equiv A \land B \\
A \lor (\neg A \land B) & \equiv A \lor B \\
\neg (A \land B) & \equiv \neg A \lor \neg B \\
\neg (A \lor B) & \equiv \neg A \land \neg B
\end{align*}
\]

Be able to “prove” any of these using a truth table approach and using either proof by contradiction or a direct proof
From last time

Symbols: a set of “characters”
Formulas: Constructed from symbols; well-formed
Axioms: special formula that are a priori assumed to be True
Inference Rules: “rules” for how to derive additional formulas that are True

- Modus Ponens
  \[
  A \implies B, A \\
  \hline
  B
  \]

- Modus Tollens
  \[
  A \implies B, \neg B \\
  \hline
  \neg A
  \]

- Conjunction
  \[
  A, B \\
  \hline
  A \land B
  \]

- Simplification
  \[
  A \land B \\
  \hline
  A
  \]

- Addition
  \[
  A \\
  \hline
  A \lor B
  \]

- Disjunctive syllogism
  \[
  A \lor B, \neg A \\
  \hline
  B
  \]

- Hypothetical syllogism
  \[
  A \implies B, B \implies C \\
  \hline
  A \implies C
  \]

- Constructive dilemma
  \[
  A \lor B, A \implies C, B \implies D \\
  \hline
  C \lor D
  \]

- Destructive dilemma
  \[
  \neg C \lor \neg D, A \implies C, B \implies D \\
  \hline
  \neg A \lor \neg B
  \]

Inference rules have a hypothesis, and a conclusion

If the hypothesis is true, then the conclusion is true

Q: Why use the “,” in the hypothesis, and not the word “and”?
Two predominant proof “methods”

Conditional Proof (prove that X is a logical implication to Y)

• Assume X is true
• Show that Y follows

Indirect Proof (proof by contradiction; to prove X)

• Assume NOT X
• Show a contradiction (Z AND NOT Z for example)
• Therefore X is True
Today

Proof by Contradiction
Programming Logic
Proof Strategies

Statements can be proven via either of the 2 methods. It’s up to you. Using one may be “easier” than the other.

Hint: use the axioms to write equivalence statements. Use ...

\[-(A \Rightarrow B) \equiv A \land \neg B\]
Proof Strategies

Q: Do these have the same form?

\[ \neg (A \Rightarrow B) \equiv A \land \neg B \]

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

1. \( \neg (A \Rightarrow (B \Rightarrow A)) \)
2. ...
3. \( Z \)
4. ...
5. \( \neg Z \)
6. \( A \Rightarrow (B \Rightarrow A) \)
Proof Strategies

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

1. $\neg A \Rightarrow (B \Rightarrow A)$
2. ...
3. $Z$
4. ...
5. $\neg Z$
6. $A \Rightarrow (B \Rightarrow A)$

Q: How does that help us?
Proof Strategies

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

1. \( \neg A \Rightarrow (B \Rightarrow A) \)
2. \( \ldots \)
3. \( Z \)
4. \( \ldots \)
5. \( \neg Z \)
6. \( A \Rightarrow (B \Rightarrow A) \)

Q: How does that help us?

The axiom tells us that when we see THAT specific pattern, it IS biconditionally equivalent to what you see on the RHS of the biconditional sign.

Hence NOT (A implication B) can be rewritten as ...
Proof Strategies

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

1. \( \neg A \Rightarrow (B \Rightarrow A) \)
2. \( \ldots \)
3. \( Z \)
4. \( \ldots \)
5. \( \neg Z \)
6. \( A \Rightarrow (B \Rightarrow A) \)

Use the axioms to “rewrite” statements into equivalent forms
Proof Strategies

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

1. $\neg A \Rightarrow (B \Rightarrow A)$
2. $\ldots$
3. $Z$
4. $\ldots$
5. $\neg Z$
6. $A \Rightarrow (B \Rightarrow A)$

Q: How does that “help” us?
Proof Strategies

Q: How does that “help” us?

Conjunction (A AND B) implies that both A and B are True ... thus are we “allowed” to write A? Are we “allowed” to write B?
Proof Strategies

Prove: \( A \implies (B \implies A) \)

1. \( \neg(A \implies (B \implies A)) \)
2. \( \ldots \)
3. \( Z \)
4. \( \ldots \)
5. \( \neg Z \)
6. \( A \implies (B \implies A) \)

Because we’ve made the statement that BOTH A and B (in this case the B is NOT \( B \implies A \)) are true, then we can use those entities as “True” facts.
Proof Strategies

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

1. \( \neg(A \Rightarrow (B \Rightarrow A)) \)
2. ... 
3. \( Z \)
4. ... 
5. \( \neg Z \)

6. \( A \Rightarrow (B \Rightarrow A) \)

Q: How does that help us?

Q: Does NOT \((B \Rightarrow A)\) look familiar?
Proof Strategies

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

1. $\neg(A \Rightarrow (B \Rightarrow A))$
2. ...
3. $Z$
4. ...
5. $\neg Z$
6. $A \Rightarrow (B \Rightarrow A)$
Proof Strategies

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

1. \( \neg (A \Rightarrow (B \Rightarrow A)) \)
2. ...
3. \( Z \)
4. ...
5. \( \neg Z \)
6. \( A \Rightarrow (B \Rightarrow A) \)

Q: What can we do next?
Proof Strategies

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

1. $\neg(A \Rightarrow (B \Rightarrow A))$
2. ...
3. $Z$
4. ...
5. $\neg Z$
6. $A \Rightarrow (B \Rightarrow A)$

Q: Now what?

$A \land \neg(B \Rightarrow A)$
$A$
$\neg(B \Rightarrow A)$
$B \land \neg A$
$B$
$\neg A$
Proof Strategies

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

1. $\neg(A \Rightarrow (B \Rightarrow A))$
2. ...
3. Z
4. ...
5. $\neg Z$
6. $A \Rightarrow (B \Rightarrow A)$

We are done! Look at the lines of the proof. We are making the statement that both $A$ and $\neg A$ are True. But that CANNOT be, therefore, the premise (hypothesis, or our starting assumption), must be FALSE...
Proof Strategies

Because our premise is FALSE, the negation of it must be True, which is what we are trying to prove!

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

1. $\neg(A \Rightarrow (B \Rightarrow A))$
2. ...
3. $Z$
4. ...
5. $\neg Z$
6. $A \Rightarrow (B \Rightarrow A)$

$A \land \neg(B \Rightarrow A)$

$A$

$\neg(B \Rightarrow A)$

$B \land \neg A$

$B$

$\neg A$
A proof’s each line is usually substantiated

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

1. \( \neg(A \Rightarrow (B \Rightarrow A)) \)  
   assumption for indirect proof
2. \( A \land \neg(B \Rightarrow A) \)  
   equiv to line 1
3. \( A \)  
   simplification from line 2
4. \( \neg(B \Rightarrow A) \)  
   simplification from line 2
5. \( B \land \neg A \)  
   equiv to line 4
6. \( B \)  
   simplification from line 5
7. \( \neg A \)  
   simplification from line 5
8. \( A \Rightarrow (B \Rightarrow A) \)  
   3 7 indirect proof
Proof Strategies

Another, more challenging proof ...

\[ \text{Prove: } (A \Rightarrow C) \Rightarrow (A \Rightarrow (B \lor C)) \]

(take home exercise)

A proof has NOT been included in homework 3, but there will be a simple proof on the midterm, and a more complex proof request on the final exam
Proving Correctness of Concurrent Programs

```c
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;
    co;
    line1 = line2;
}

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\}
}

(array copy problem)
```
All of this logic. What does it have to do with Concurrent programming?

We want to be able to reason logically about our programming languages, PL.
Programming Logic

All of this logic. What does it have to do with Concurrent programming?

We want to be able to reason logically about our programming languages, PL

Axiomatic Semantics

Define axiom
Define semantics
Programming Logic

All of this logic. What does it have to do with Concurrent programming?

We want to be able to reason logically about our programming languages, PL

Axiomatic Semantics

\{ P \} \ S \ \{ Q \}
All of this logic. What does it have to do with Concurrent programming?

We want to be able to reason logically about our programming languages, PL.

Axiomatic Semantics

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

• Called a **triple**
• If \(P\) is true before \(S\) is executed, and \(S\) terminates, then \(Q\) is true after \(S\) executes
• \(P\) and \(Q\) are assertions
• \(P\) is the precondition
• \(Q\) is the post-condition
Using such axiomatic semantics ... We could write ...

\[
\begin{align*}
{x=0} & \quad x = x + 1; \quad {x=1} \\
{x=0} & \quad x = x + 1; \quad {y=1}
\end{align*}
\]

Q: Which of these are sound?
Using such axiomatic semantics ... We could write ...

\{ x=0 \} \ x = x + 1; \ \{ x=1 \}

\{ x=0 \} \ x = x + 1; \ \{ y=1 \}

Task: Be able to explain in your own words what these “mean”, and which of these are \textbf{sound}.

Whether something is \textbf{sound} is based on whether it (a fact or theorem) is derived from facts (true statements).
Using such axiomatic semantics ... We could write ...

sound: \{ x=0 \} \ x = x + 1; \ { x=1 \}  
This should be a theorem

Not sound: \{ x=0 \} \ x = x + 1; \ { y=1 \}  
This should not be a theorem

Task: Be able to explain in your own words what these “mean”, and which of these are sound

Whether something is sound is based on whether it (a fact or theorem) is derived from facts (true statements)
Using axiomatic semantics ... the **assignment axiom**

\[
\{ P_{x \leftarrow e} \} \ x = e \ {P}
\]

\[
\{1 = 1\} \ x = 1 \ {x = 1}
\]

\[
\{true \} \ x = 1 \ {x = 1}
\]

\[
\{x + 1 = 1\} \ x = x + 1 \ {x = 1}
\]

\[
\{x = 0\} \ x = x + 1 \ {x = 1}
\]

\[
\{x + 1 = n\} \ x = x + 1 \ {x = n}
\]

\[
\{x = n - 1\} \ x = x + 1 \ {x = n}
\]
Using axiomatic semantics ... the assignment axiom

\[ \{P \}_{x \leftarrow e} \quad x = e \quad \{P\} \]

Replace all free occurrences of variable \(x\) in predicate \(P\) by expression \(e\)

\[
\{1 = 1\} \quad x = 1 \quad \{x = 1\}
\]

\[
\{true\} \quad x = 1 \quad \{x = 1\}
\]

\[
\{x + 1 = 1\} \quad x = x + 1 \quad \{x = 1\}
\]

\[
\{x = 0\} \quad x = x + 1 \quad \{x = 1\}
\]

\[
\{x + 1 = n\} \quad x = x + 1 \quad \{x = n\}
\]

\[
\{x = n - 1\} \quad x = x + 1 \quad \{x = n\}
\]
Using axiomatic semantics ... the **assignment axiom**

\[
\begin{array}{c}
\text{pre} \quad \{P_{x=e}\} \quad x = e \quad \{P\} \\
\text{post}
\end{array}
\]

Statement that terminates

\[
\begin{align*}
\{1 = 1\} & \quad x = 1 \quad \{x = 1\} \\
\{true\} & \quad x = 1 \quad \{x = 1\}
\end{align*}
\]

\[
\begin{align*}
\{x + 1 = 1\} & \quad x = x + 1 \quad \{x = 1\} \\
\{x = 0\} & \quad x = x + 1 \quad \{x = 1\}
\end{align*}
\]

\[
\begin{align*}
\{x + 1 = n\} & \quad x = x + 1 \quad \{x = n\} \\
\{x = n - 1\} & \quad x = x + 1 \quad \{x = n\}
\end{align*}
\]
Using axiomatic semantics ... the assignment axiom

\[ \{P_{x=e}\} x = e \{P\} \]

All examples of the assignment axiom

\[ \{1 = 1\} x = 1 \{x = 1\} \]
\[ \{true\} x = 1 \{x = 1\} \]
\[ \{x + 1 = 1\} x = x + 1 \{x = 1\} \]
\[ \{x = 0\} x = x + 1 \{x = 1\} \]
\[ \{x + 1 = n\} x = x + 1 \{x = n\} \]
\[ \{x = n - 1\} x = x + 1 \{x = n\} \]
Programming Logic

Using axiomatic semantics ... the **assignment axiom**

\[
\{P_{x=e}\} \ x = e \ \{P\}
\]

\[
\begin{align*}
\{1 = 1\} \ x = 1 \ \{x = 1\} \\
\{true\} \ x = 1 \ \{x = 1\}
\end{align*}
\]

\[
\begin{align*}
\{x + 1 = 1\} \ x = x + 1 \ \{x = 1\} \\
\{x = 0\} \ x = x + 1 \ \{x = 1\} \\
\{x + 1 = n\} \ x = x + 1 \ \{x = n\} \\
\{x = n - 1\} \ x = x + 1 \ \{x = n\}
\end{align*}
\]
Programming Logic

Using axiomatic semantics ... the **assignment axiom**

\[
\{ P_{x \leftarrow e} \} \ x = e \ \{ P \} \\
\{ 1 = 1 \} \ x = 1 \ \{ x = 1 \} \\
\{ \text{true} \} \ x = 1 \ \{ x = 1 \} \\
\{ x + 1 = 1 \} \ x = x + 1 \ \{ x = 1 \} \\
\{ x = 0 \} \ x = x + 1 \ \{ x = 1 \} \\
\{ x + 1 = n \} \ x = x + 1 \ \{ x = n \} \\
\{ x = n - 1 \} \ x = x + 1 \ \{ x = n \} 
\]

If the pre-condition (starting state) is \( x=0 \), and we execute \( x = x + 1 \), then the post state (\( x=1 \)) is guaranteed to be the result.
Just as we had inference rules in “logic”, we have inference rules for our PL

**Composition Rule:**

\[
\frac{\{\mathcal{P}\} S_1 \{\mathcal{Q}\}, \{\mathcal{Q}\} S_2 \{\mathcal{R}\}}{\{\mathcal{P}\} S_1; S_2 \{\mathcal{R}\}}
\]

**If Statement Rule:**

\[
\frac{\{\mathcal{P} \land B\} S \{\mathcal{Q}\}, (\mathcal{P} \land \lnot B) \Rightarrow \mathcal{Q}}{\{\mathcal{P}\} \text{ if } (B) S; \{\mathcal{Q}\}}
\]

**While Statement Rule:**

\[
\frac{\{\mathcal{I} \land B\} S \{\mathcal{I}\}}{\{\mathcal{I}\} \text{ while}(B) S; \{\mathcal{I} \land \lnot B\}}
\]

**Rule of Consequence:**

\[
\frac{\mathcal{P}' \Rightarrow \mathcal{P}, \{\mathcal{P}\} S \{\mathcal{Q}\}, \mathcal{Q} \Rightarrow \mathcal{Q}'}{\{\mathcal{P}'\} S \{\mathcal{Q}'\}}
\]
Concurrency

Where/how is axiomatic semantics used in concurrency?
Concurrency

\[
\begin{aligned}
\{x = 0\} \\
\text{co} \langle x = x+1; \rangle &// \langle x = x+2; \rangle \text{ co} \\
\{x = 3\}
\end{aligned}
\]
Related to homework #3: What are the possible final states of x?
Related to homework #3: What are the possible final states of $x$?

The atomic signs $< >$ specify that each of the two arms has atomic access to the variables in that statement. Hence, the final possible value of $x$ is ONLY 3. Note that this is a consequence of the definition of the triple ... “if $P$ is true, and if $S$ completes, then $Q$ is true”
Concurrent Programming

What are the possible “intermediate” states of $x$?
What are the possible “intermediate” states of x?

This depends on which arm goes first. If the Left arm “goes first”, then x as it sees it is 0, and then the Right arm will see x as 1. If the Right arm “goes first”, then x as it sees it is 1, and then the Left arm will see x as 2.

Q: How can this be noted?
Concurrent Programming

\[
\begin{align*}
\{x &= 0\} \\
\text{co} & \\
\{x &= 0 \lor x = 2\} \\
\langle \ x = x + 1; \ \rangle & \\
\{x &= 1 \lor x = 3\} \\
\text{//} & \\
\{x &= 0 \lor x = 1\} \\
\langle \ x = x + 2; \ \rangle & \\
\{x &= 2 \lor x = 3\} \\
\text{oc} & \\
\{x &= 3\}
\end{align*}
\]
Concurrency

\{ x = 0 \}

con

\{ x = 0 \lor x = 2 \}
\langle x = x + 1; \rangle
\{ x = 1 \lor x = 3 \}

con

\{ x = 0 \lor x = 1 \}
\langle x = x + 2; \rangle
\{ x = 2 \lor x = 3 \}

ocl

\{ x = 3 \}
Concurrency

\[
\begin{align*}
\{x = 0\} \\
\begin{align*}
\{x = 0 \lor x = 2\} \\
\langle x = x + 1; \rangle \\
\{x = 1 \lor x = 3\}
\end{align*}
\end{align*}
\]

//

\[
\begin{align*}
\{x = 0 \lor x = 1\} \\
\langle x = x + 2; \rangle \\
\{x = 2 \lor x = 3\}
\end{align*}
\]

\[
\{x = 3\}
\]
Concurrent Programming

Notice that the first arm and the second arm may interfere with each other, because of their sharing of the variable \( x \). Although both S statements are atomic, what arm 1 does will influence what arm 2 computes, and vice versa.

Task: We need some way to reason about the extent that one “arm” interferes with another.
• An assignment action is an assignment statement or an await statement that contains one or more assignments
• A critical assertion is a precondition or post-condition that is not within an await statement

Task: Identify the assignment actions
Task: Identify the critical assertions
### Concurrency

- An **assignment action** is an assignment statement or an await statement that contains one or more assignments.
- A **critical assertion** is a precondition or post-condition that is not within an await statement.
- Let \( a \) be an assignment action in one process and let \( \text{pre}(a) \) be its precondition. Let \( C \) be a critical assertion in another process. Then \( a \) does not interfere with \( C \) if the following is true:

\[
\{C \land \text{pre}(a)\} \ a \ {C}
\]

- We say that \( C \) is **invariant** with respect to \( a \).
An **assignment action** is an assignment statement or an await statement that contains one or more assignments.

A **critical assertion** is a precondition or post-condition that is not within an await statement.

Let \( a \) be an assignment action in one process and let \( \text{pre}(a) \) be its precondition. Let \( C \) be a critical assertion in another process. Then \( a \) does not interfere with \( C \) if the following is true:

\[
\{ C \text{ AND } \text{pre}(a) \} \ a \ \{ C \}
\]

We say that \( C \) is **invariant** with respect to \( a \).

**Q:** Is \( C \) invariant with respect to \( a \)?
Concurrent Programming

• An **assignment action** is an assignment statement or an await statement that contains one or more assignments.

• A **critical assertion** is a precondition or post-condition that is not within an await statement.

• Let \( a \) be an assignment action in one process and let \( \text{pre}(a) \) be its precondition. Let \( C \) be a critical assertion in another process. Then \( a \) does not interfere with \( C \) if the following is true:

\[
\{ C \text{ AND } \text{pre}(a) \} \ a \ \{ C \}
\]

• We say that \( C \) is **invariant** with respect to \( a \).

Yes: because if \((x=0 \text{ or } x=2) \text{ AND } (x=0 \text{ or } x=1)\), followed by \( a \), then \( x=0 \text{ or } x=1 \).
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
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

We want to write concurrent programs such that BAD is false in every state, and GOOD is true in every state.
Safety and Liveness Properties

- Preconditions do not interfere
- Suppose $\text{pre}(S_1) \text{ AND } \text{pre}(S_2) = \text{false}$
Safety and Liveness Properties

- Preconditions do not interfere
- Suppose $\text{pre}(S1) \text{ AND } \text{pre}(S2) = false$

The two processes cannot be at these states at the same time

$false$ means that the program is NEVER in that state

- In our array copy example from last week, both processes cannot simultaneously be delays in executing their await statements
Safety and Liveness Properties

**Global invariant**

The PC refers to a rule that applies to BOTH the Producer and Consumer

```c
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}
}
```

**Consumer invariant**

The IC refers to a rule that applies to the consumer

```c
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}
}
```
Safety and Liveness Properties

```plaintext
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}
}
```
Safety and Liveness Properties

```plaintext
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 ∧ p == c}
    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}  
}
```
These preconditions (defined as invariants) which prevent the producer and consumer processes from reading from and writing to the buffer at the same time.
Safety and Liveness Properties

In the remaining chapters of the book and weeks of the quarter, we’ll learn how to implement these “behaviors” via locks, barriers, etc. (you’ve already seen semaphores)

```c
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}
}
```