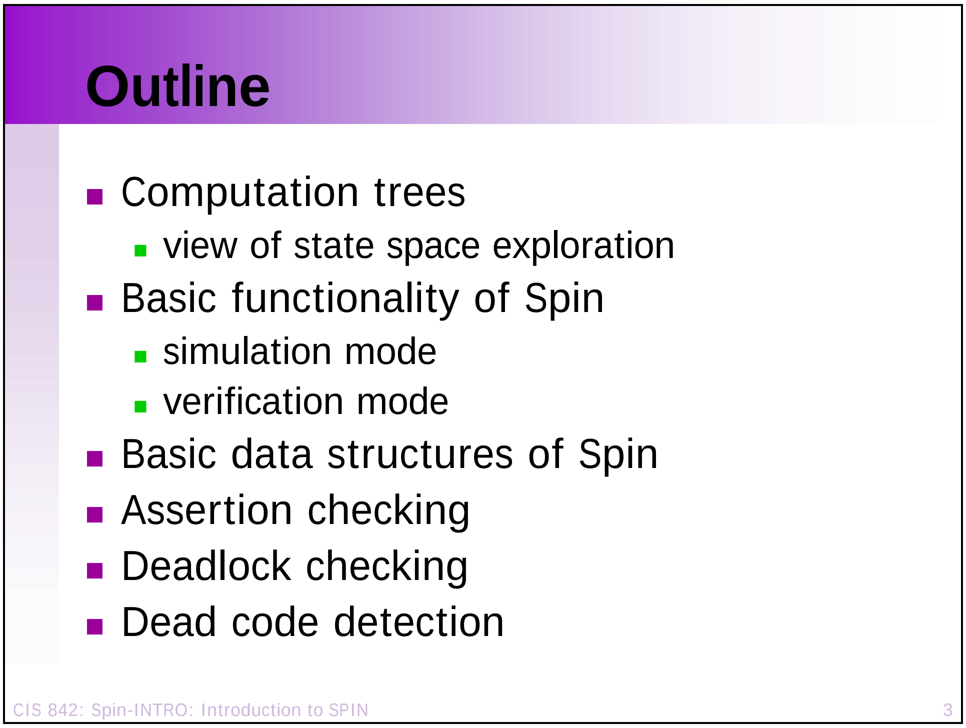
CIS 842: Specification and Verification of Reactive Systems

Lecture SPIN-INTRO: Introduction To SPIN

Objectives

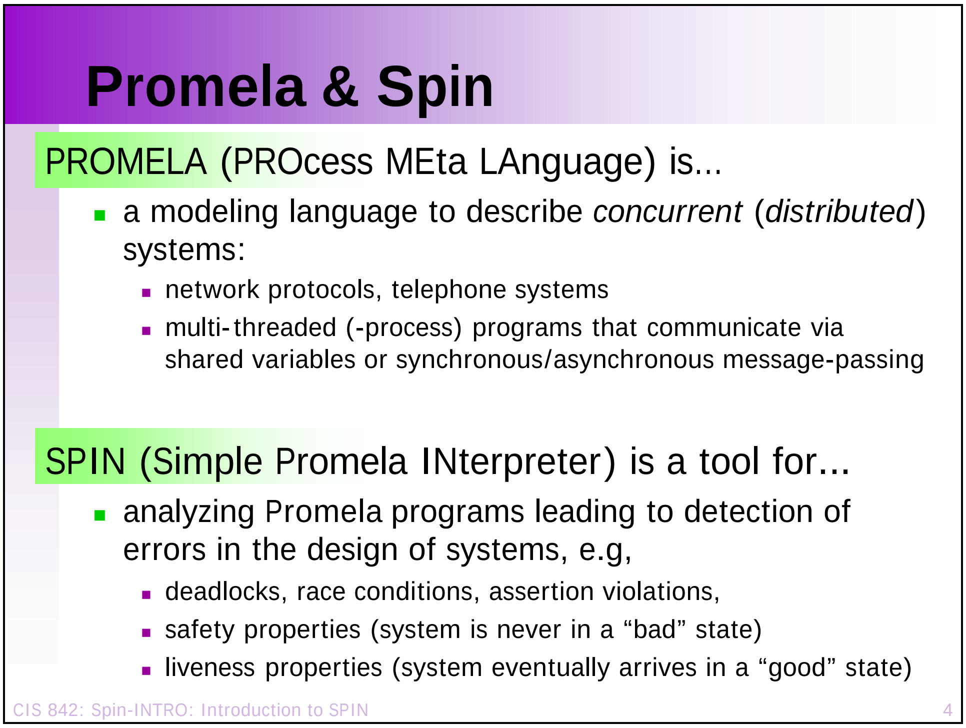
- Understand the purpose of
  - the Promela modeling language, and
  - the Spin analysis tool
- Understand the basic functionality of Spin and how to apply it to reason about and verify properties of simple concurrent systems
- Understand Spin's basic data structures

Be sure you have Spin installed before you start this lecture!



Outline

- Computation trees
  - view of state space exploration
- Basic functionality of Spin
  - simulation mode
  - verification mode
- Basic data structures of Spin
- Assertion checking
- Deadlock checking
- Dead code detection



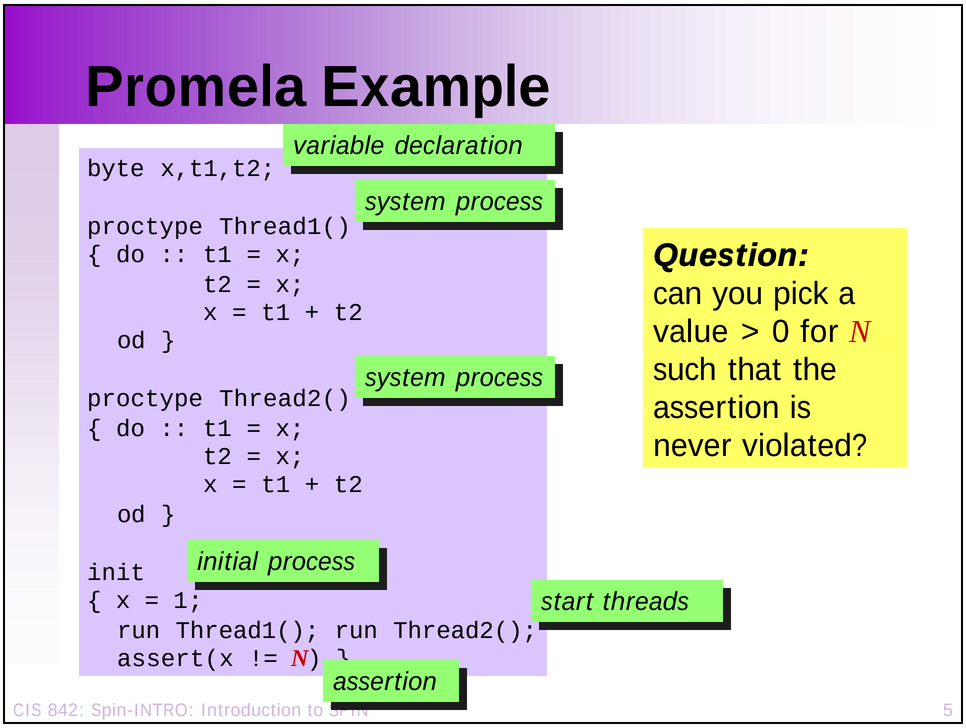
Promela & Spin

PROMELA (PROcess MEta LAnguage) is...

- a modeling language to describe *concurrent* (*distributed*) systems:
  - network protocols, telephone systems
  - multi-threaded (-process) programs that communicate via shared variables or synchronous/asynchronous message-passing

SPIN (Simple Promela INterpreter) is a tool for...

- analyzing Promela programs leading to detection of errors in the design of systems, e.g,
  - deadlocks, race conditions, assertion violations,
  - safety properties (system is never in a “bad” state)
  - liveness properties (system eventually arrives in a “good” state)



Promela Example

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;
        t2 = x;
        x = t1 + t2
  od }

proctype Thread2()
{ do :: t1 = x;
        t2 = x;
        x = t1 + t2
  od }

init
{ x = 1;
  run Thread1(); run Thread2();
  assert(x != N) }
```

variable declaration

system process

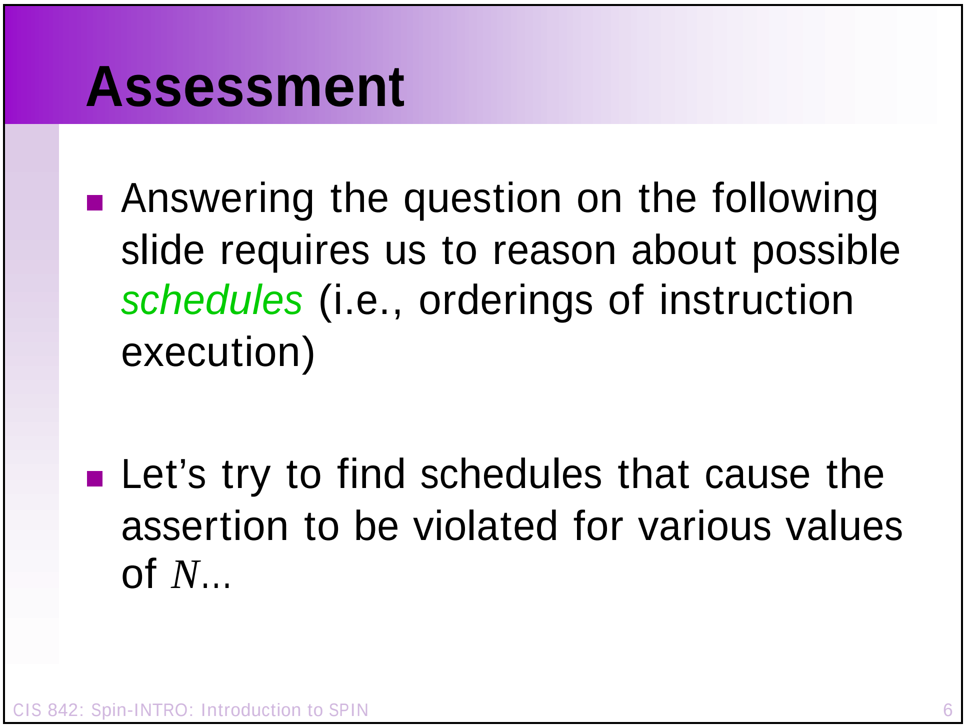
system process

initial process

start threads

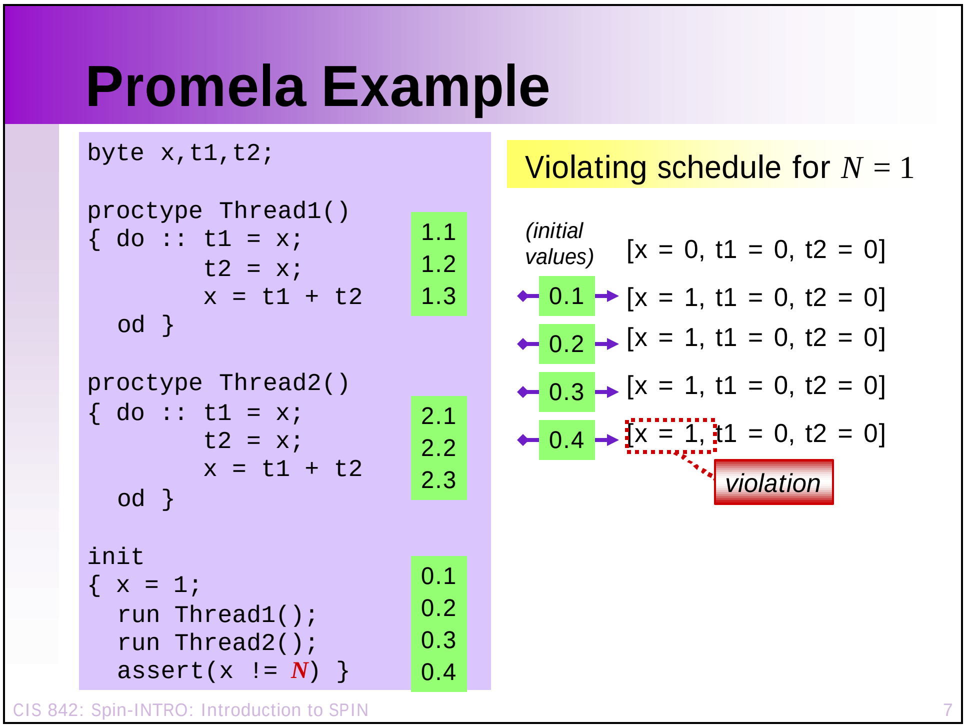
assertion

Question: can you pick a value > 0 for N such that the assertion is never violated?



Assessment

- Answering the question on the following slide requires us to reason about possible *schedules* (i.e., orderings of instruction execution)
- Let’s try to find schedules that cause the assertion to be violated for various values of N...



Promela Example

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;              1.1
        t2 = x;              1.2
        x = t1 + t2          1.3
  od }

proctype Thread2()
{ do :: t1 = x;              2.1
        t2 = x;              2.2
        x = t1 + t2          2.3
  od }

init
{ x = 1;                     0.1
  run Thread1();             0.2
  run Thread2();             0.3
  assert(x != N) }           0.4
```

Violating schedule for $N = 1$

(initial values) [x = 0, t1 = 0, t2 = 0]

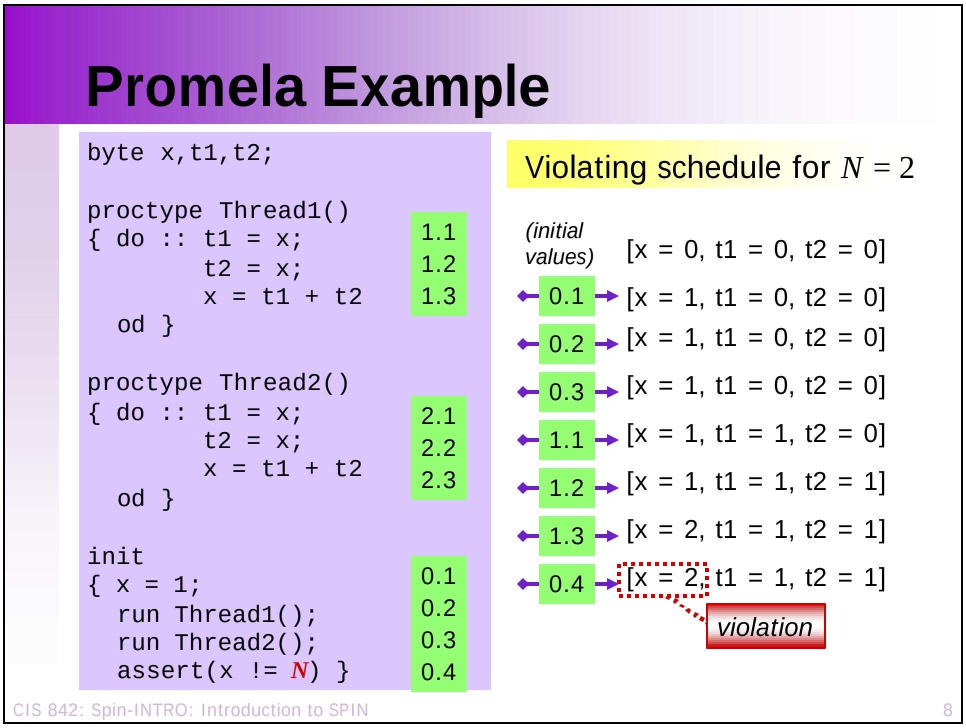
0.1 → [x = 1, t1 = 0, t2 = 0]

0.2 → [x = 1, t1 = 0, t2 = 0]

0.3 → [x = 1, t1 = 0, t2 = 0]

0.4 → [x = 1, t1 = 0, t2 = 0]

violation



Promela Example

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;              1.1
        t2 = x;              1.2
        x = t1 + t2          1.3
  od }

proctype Thread2()
{ do :: t1 = x;              2.1
        t2 = x;              2.2
        x = t1 + t2          2.3
  od }

init
{ x = 1;                     0.1
  run Thread1();             0.2
  run Thread2();             0.3
  assert(x != N) }           0.4
```

Violating schedule for $N = 2$

(initial values) [x = 0, t1 = 0, t2 = 0]

0.1 → [x = 1, t1 = 0, t2 = 0]

0.2 → [x = 1, t1 = 0, t2 = 0]

0.3 → [x = 1, t1 = 0, t2 = 0]

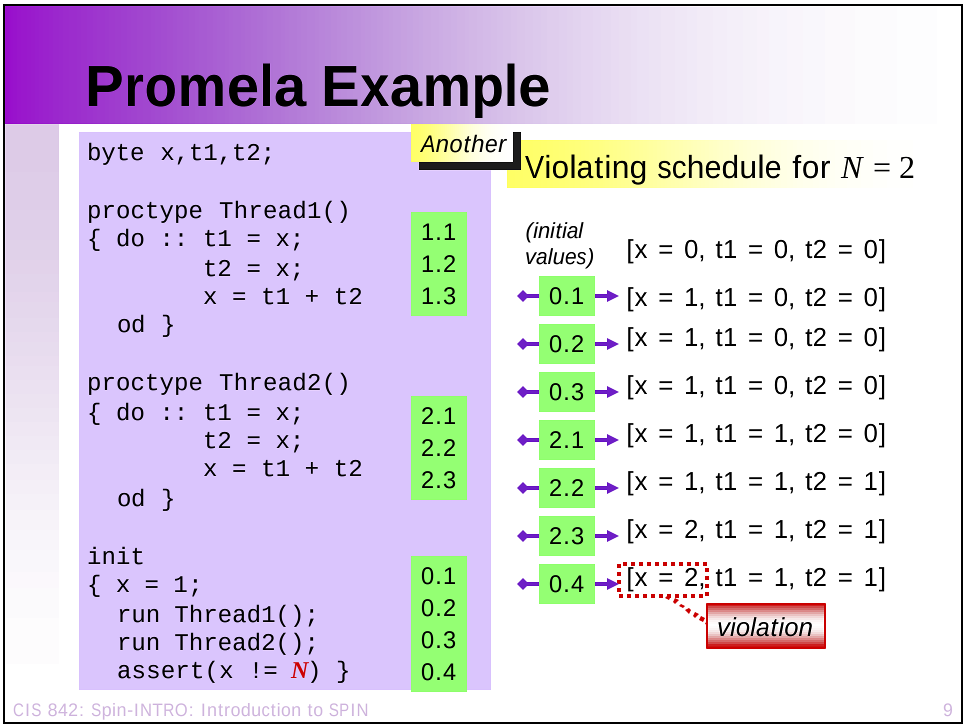
1.1 → [x = 1, t1 = 1, t2 = 0]

1.2 → [x = 1, t1 = 1, t2 = 1]

1.3 → [x = 2, t1 = 1, t2 = 1]

0.4 → [x = 2, t1 = 1, t2 = 1]

violation



Promela Example

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;              1.1
        t2 = x;              1.2
        x = t1 + t2          1.3
  od }

proctype Thread2()
{ do :: t1 = x;              2.1
        t2 = x;              2.2
        x = t1 + t2          2.3
  od }

init
{ x = 1;                     0.1
  run Thread1();             0.2
  run Thread2();             0.3
  assert(x != N) }           0.4
```

Another Violating schedule for $N = 2$

(initial values) [x = 0, t1 = 0, t2 = 0]

0.1 → [x = 1, t1 = 0, t2 = 0]

0.2 → [x = 1, t1 = 0, t2 = 0]

0.3 → [x = 1, t1 = 0, t2 = 0]

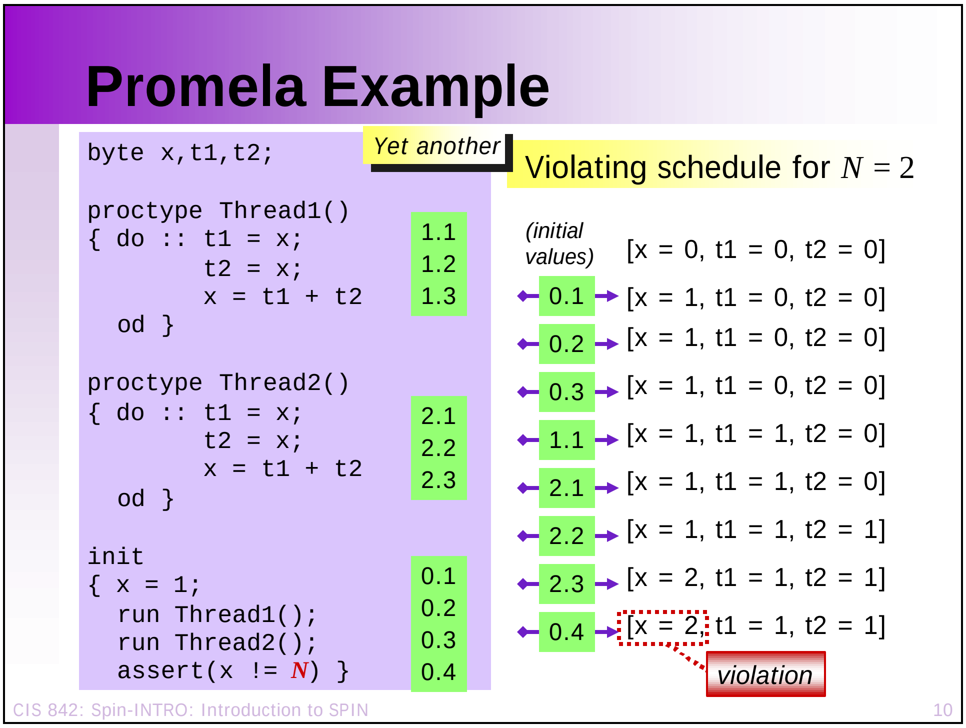
2.1 → [x = 1, t1 = 1, t2 = 0]

2.2 → [x = 1, t1 = 1, t2 = 1]

2.3 → [x = 2, t1 = 1, t2 = 1]

0.4 → [x = 2, t1 = 1, t2 = 1]

violation



Promela Example

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;              1.1
        t2 = x;              1.2
        x = t1 + t2          1.3
  od }

proctype Thread2()
{ do :: t1 = x;              2.1
        t2 = x;              2.2
        x = t1 + t2          2.3
  od }

init
{ x = 1;                     0.1
  run Thread1();             0.2
  run Thread2();             0.3
  assert(x != N) }           0.4
```

Yet another Violating schedule for $N = 2$

(initial values) [x = 0, t1 = 0, t2 = 0]

0.1 → [x = 1, t1 = 0, t2 = 0]

0.2 → [x = 1, t1 = 0, t2 = 0]

0.3 → [x = 1, t1 = 0, t2 = 0]

1.1 → [x = 1, t1 = 1, t2 = 0]

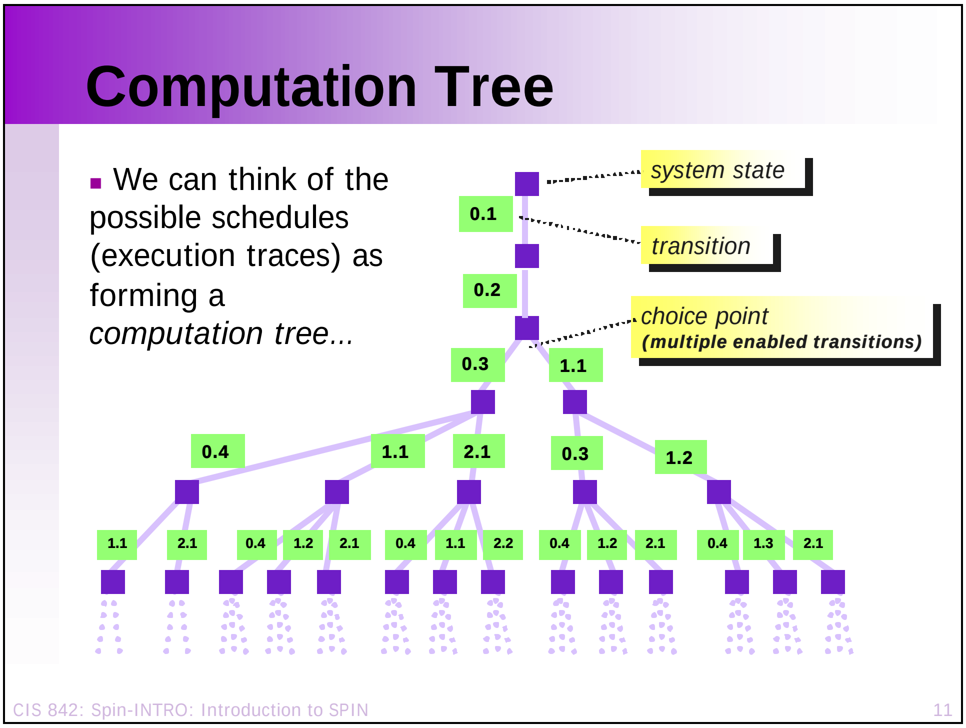
2.1 → [x = 1, t1 = 1, t2 = 0]

2.2 → [x = 1, t1 = 1, t2 = 1]

2.3 → [x = 2, t1 = 1, t2 = 1]

0.4 → [x = 2, t1 = 1, t2 = 1]

violation



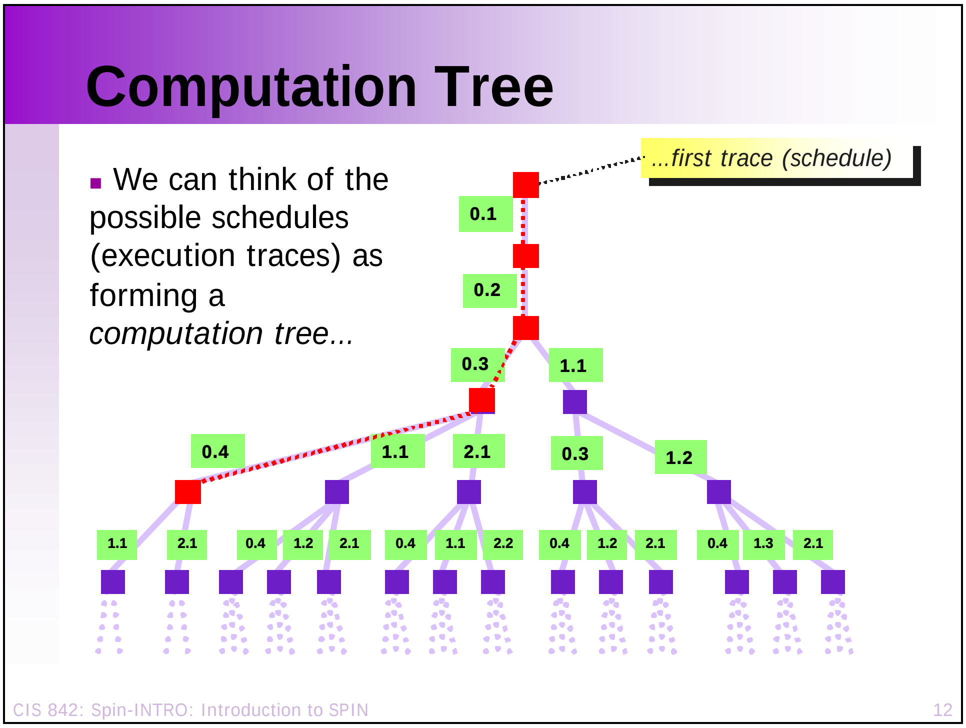
Computation Tree

- We can think of the possible schedules (execution traces) as forming a *computation tree...*

system state

transition

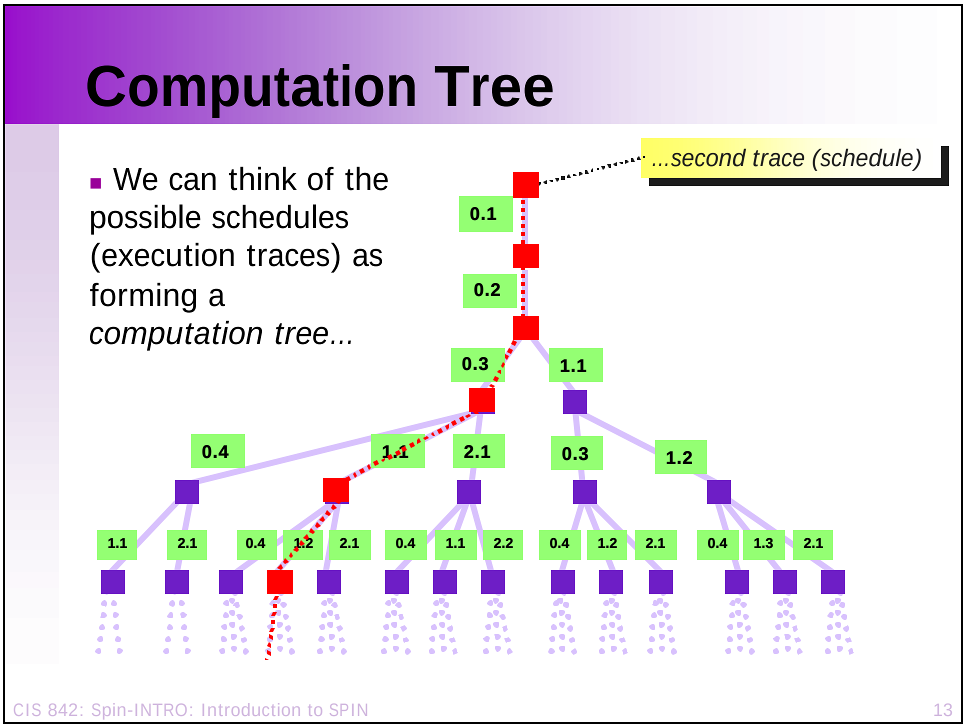
choice point
(multiple enabled transitions)



Computation Tree

- We can think of the possible schedules (execution traces) as forming a *computation tree...*

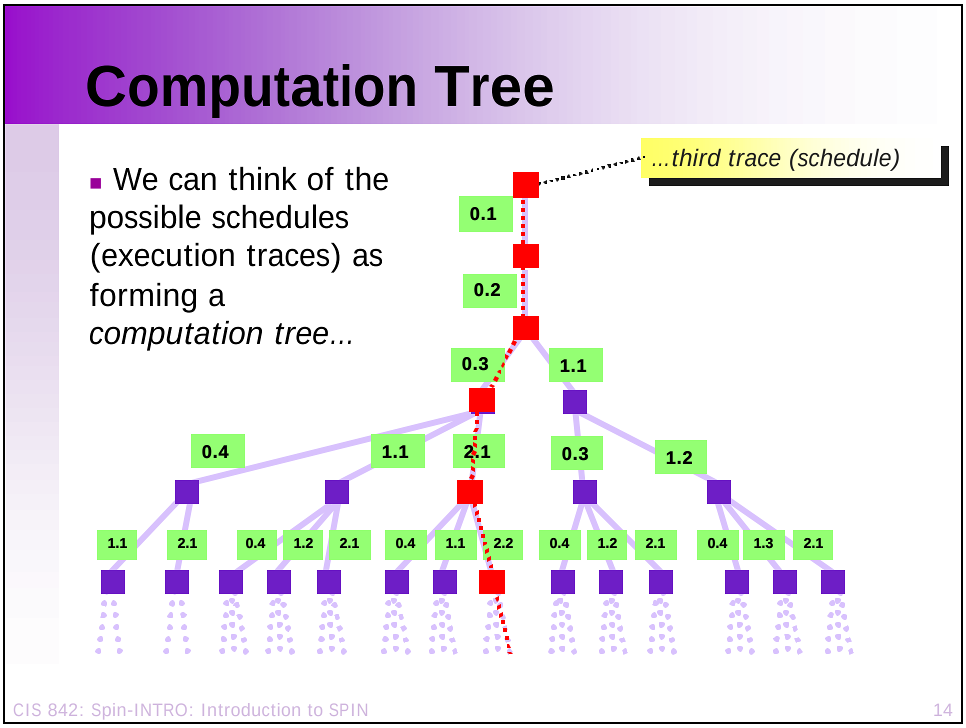
...first trace (schedule)



Computation Tree

- We can think of the possible schedules (execution traces) as forming a *computation tree...*

...second trace (schedule)



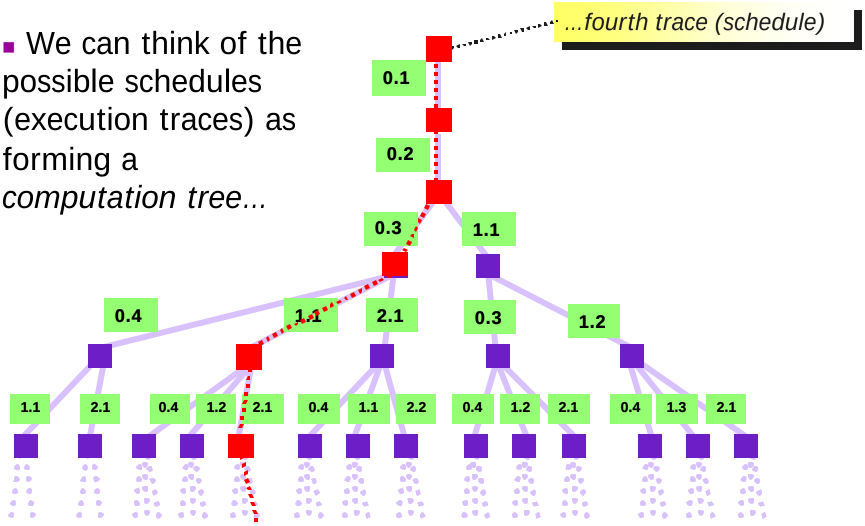
Computation Tree

- We can think of the possible schedules (execution traces) as forming a *computation tree...*

...third trace (schedule)

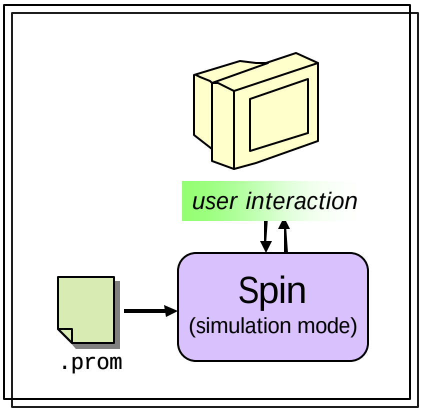
Computation Tree

- We can think of the possible schedules (execution traces) as forming a *computation tree...*



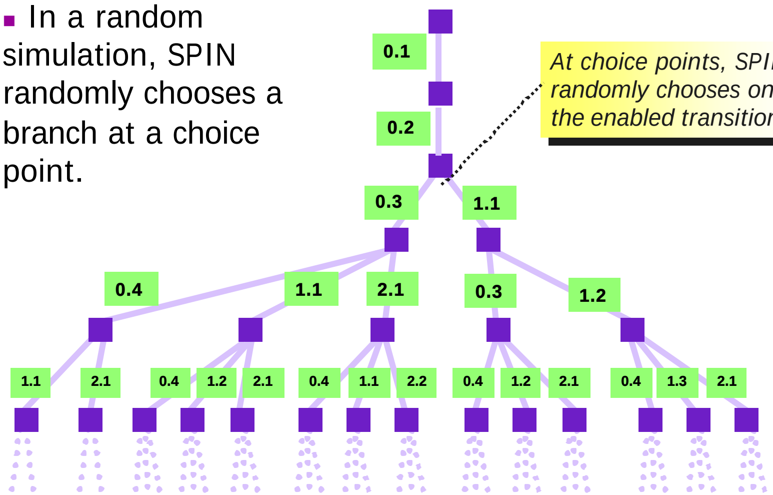
Assessment

- Even though this is a very small system, it is already tedious for us to try to find a violating trace.
- Spin can act as a *simulator* to help us find a violating trace.
- Spin simulates in two ways:
  - random simulation
  - user-guided simulation



Random Simulation

- In a random simulation, SPIN randomly chooses a branch at a choice point.

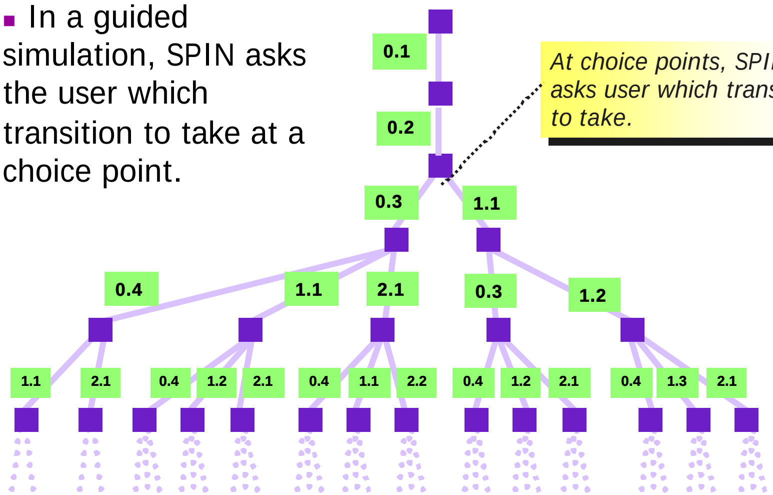


At choice points, SPIN randomly chooses one of the enabled transitions.

```
$ spin <promela file>
```

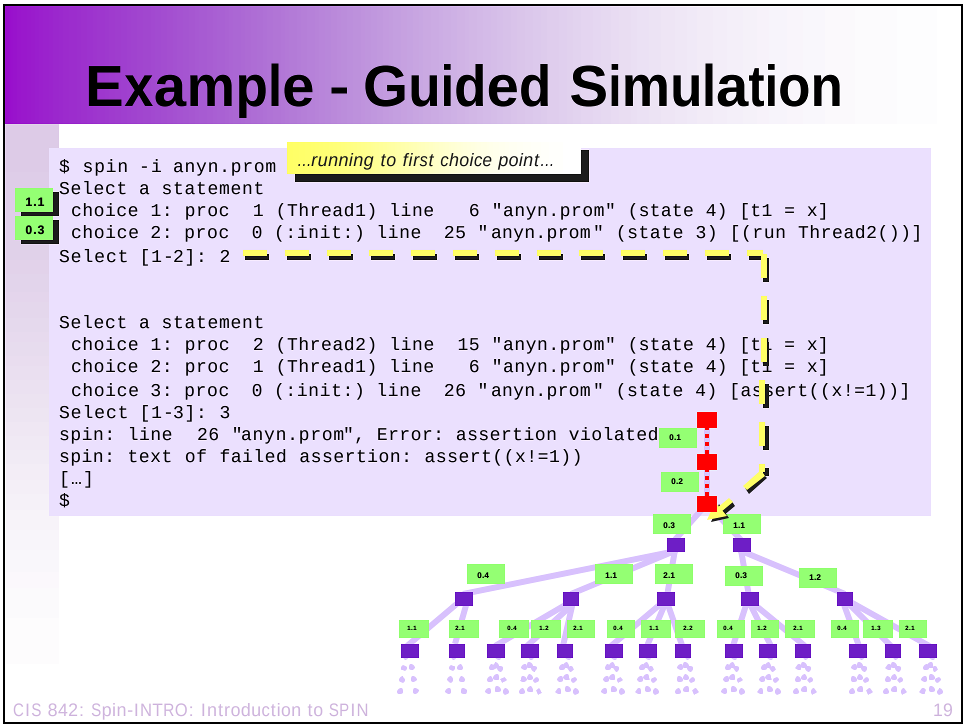
Guided Simulation

- In a guided simulation, SPIN asks the user which transition to take at a choice point.



At choice points, SPIN asks user which transition to take.

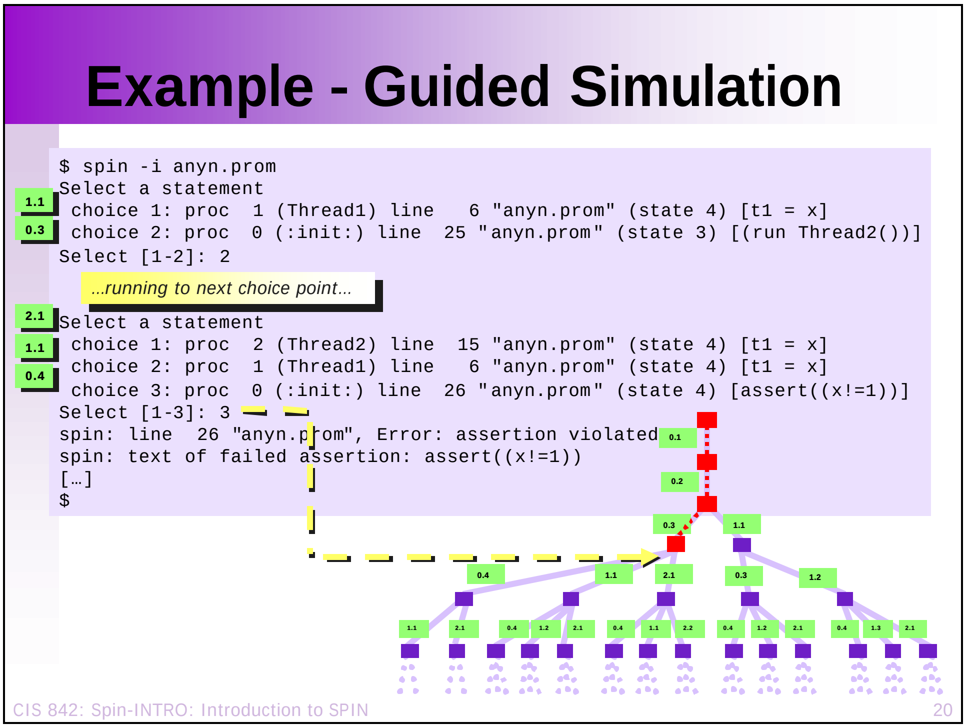
```
$ spin –i <promela file>
```



Example - Guided Simulation

...running to first choice point...

```
$ spin -i anyn.prom
Select a statement
 choice 1: proc  1 (Thread1) line   6 "anyn.prom" (state 4) [t1 = x]
 choice 2: proc  0 (:init:) line  25 "anyn.prom" (state 3) [(run Thread2())]
Select [1-2]: 2


Select a statement
 choice 1: proc  2 (Thread2) line  15 "anyn.prom" (state 4) [t1 = x]
 choice 2: proc  1 (Thread1) line   6 "anyn.prom" (state 4) [t1 = x]
 choice 3: proc  0 (:init:) line  26 "anyn.prom" (state 4) [assert((x!=1))]
Select [1-3]: 3
spin: line  26 "anyn.prom", Error: assertion violated
spin: text of failed assertion: assert((x!=1))
[…]
$
```

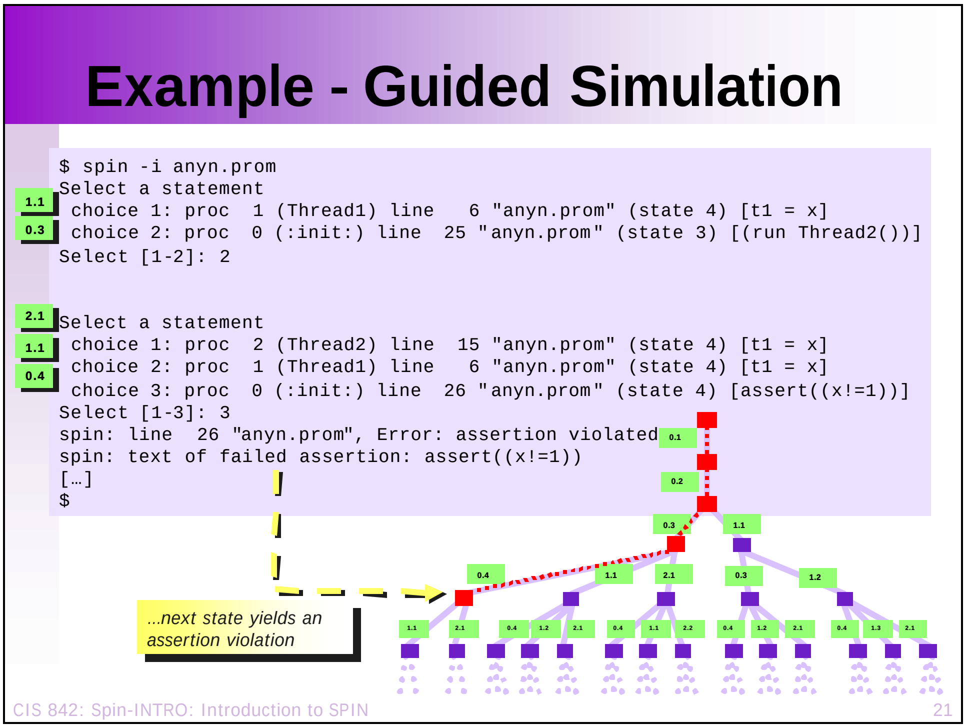


Example - Guided Simulation

```
$ spin -i anyn.prom
Select a statement
 choice 1: proc  1 (Thread1) line   6 "anyn.prom" (state 4) [t1 = x]
 choice 2: proc  0 (:init:) line  25 "anyn.prom" (state 3) [(run Thread2())]
Select [1-2]: 2
```

...running to next choice point...

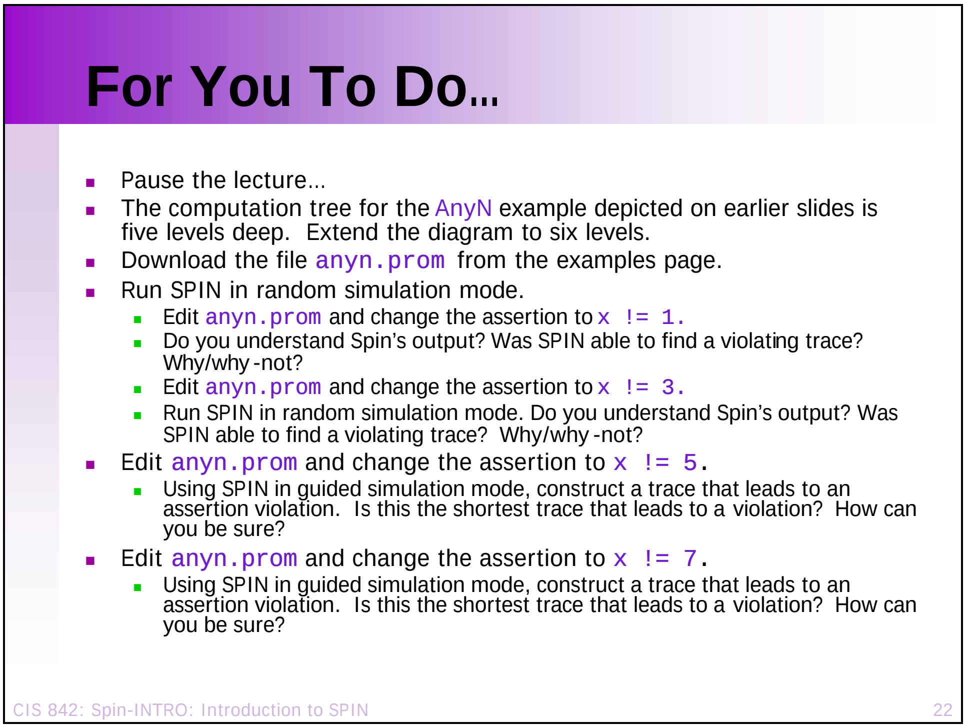
```
Select a statement
 choice 1: proc  2 (Thread2) line  15 "anyn.prom" (state 4) [t1 = x]
 choice 2: proc  1 (Thread1) line   6 "anyn.prom" (state 4) [t1 = x]
 choice 3: proc  0 (:init:) line  26 "anyn.prom" (state 4) [assert((x!=1))]
Select [1-3]: 3
spin: line  26 "anyn.prom", Error: assertion violated
spin: text of failed assertion: assert((x!=1))
[…]
$
```



Example - Guided Simulation

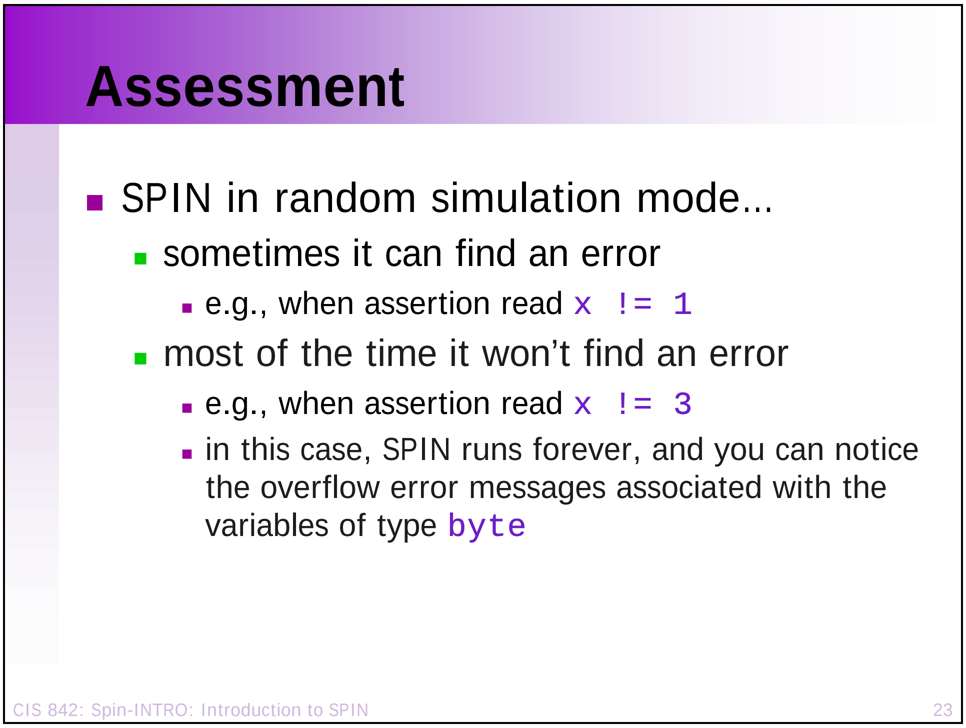
```
$ spin -i anyn.prom
Select a statement
 choice 1: proc  1 (Thread1) line   6 "anyn.prom" (state 4) [t1 = x]
 choice 2: proc  0 (:init:) line  25 "anyn.prom" (state 3) [(run Thread2())]
Select [1-2]: 2


Select a statement
 choice 1: proc  2 (Thread2) line  15 "anyn.prom" (state 4) [t1 = x]
 choice 2: proc  1 (Thread1) line   6 "anyn.prom" (state 4) [t1 = x]
 choice 3: proc  0 (:init:) line  26 "anyn.prom" (state 4) [assert((x!=1))]
Select [1-3]: 3
spin: line  26 "anyn.prom", Error: assertion violated
spin: text of failed assertion: assert((x!=1))
[…]
$
```

…next state yields an assertion violation



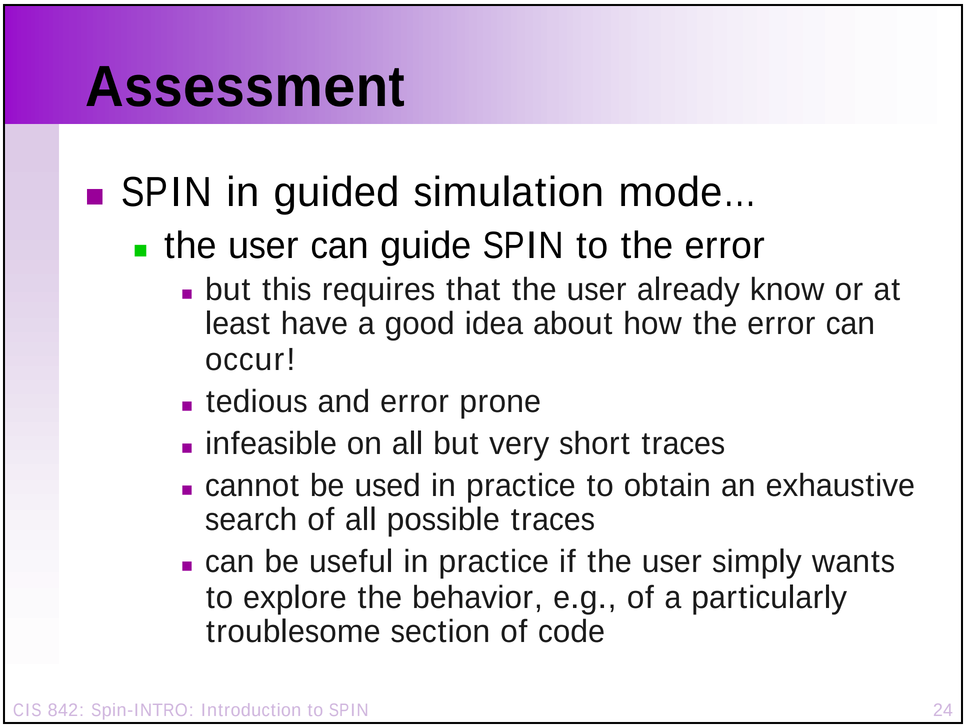
For You To Do…

- Pause the lecture…
- The computation tree for the AnyN example depicted on earlier slides is five levels deep. Extend the diagram to six levels.
- Download the file `anyn.prom` from the examples page.
- Run SPIN in random simulation mode.
  - Edit `anyn.prom` and change the assertion to `x != 1.`
  - Do you understand Spin’s output? Was SPIN able to find a violating trace? Why/why-not?
  - Edit `anyn.prom` and change the assertion to `x != 3.`
  - Run SPIN in random simulation mode. Do you understand Spin’s output? Was SPIN able to find a violating trace? Why/why-not?
- Edit `anyn.prom` and change the assertion to `x != 5`.
  - Using SPIN in guided simulation mode, construct a trace that leads to an assertion violation. Is this the shortest trace that leads to a violation? How can you be sure?
- Edit `anyn.prom` and change the assertion to `x != 7`.
  - Using SPIN in guided simulation mode, construct a trace that leads to an assertion violation. Is this the shortest trace that leads to a violation? How can you be sure?



Assessment

- SPIN in random simulation mode...
  - sometimes it can find an error
    - e.g., when assertion read `x != 1`
  - most of the time it won't find an error
    - e.g., when assertion read `x != 3`
    - in this case, SPIN runs forever, and you can notice the overflow error messages associated with the variables of type `byte`



Assessment

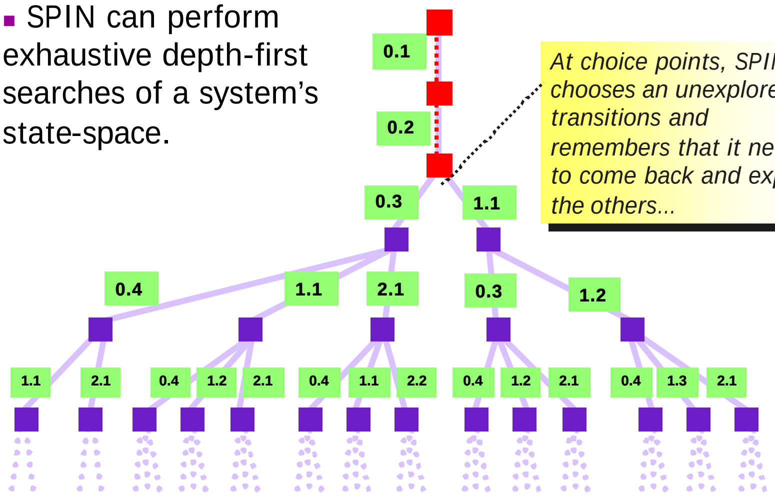
- SPIN in guided simulation mode...
  - the user can guide SPIN to the error
    - but this requires that the user already know or at least have a good idea about how the error can occur!
    - tedious and error prone
    - infeasible on all but very short traces
    - cannot be used in practice to obtain an exhaustive search of all possible traces
    - can be useful in practice if the user simply wants to explore the behavior, e.g., of a particularly troublesome section of code

On To Exhaustive Exploration…

- SPIN’s randon simulation
  - isn’t that useful for finding bugs
  - only explores one execution trace
- SPIN’s guided simulation
  - is only useful on short traces where the user already has a good idea of how a property violation might arise
  - only feasible to explore a few execution traces
- The main strength of SPIN is its automatic exhaustive search capabilities
  - this is why people use SPIN
  - this is what this course is all about

Exhaustive Depth-first Search

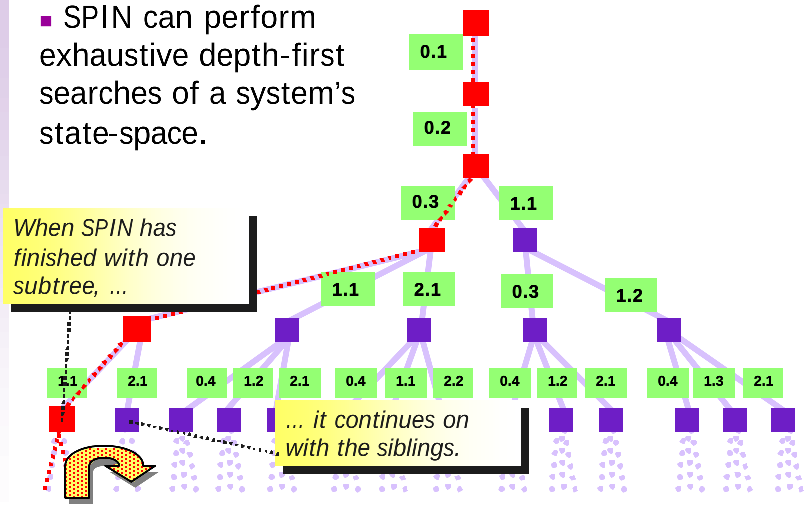
- SPIN can perform exhaustive depth-first searches of a system’s state-space.



At choice points, SPIN chooses an unexplored transitions and remembers that it needs to come back and explore the others…

Exhaustive Depth-first Search

- SPIN can perform exhaustive depth-first searches of a system's state-space.

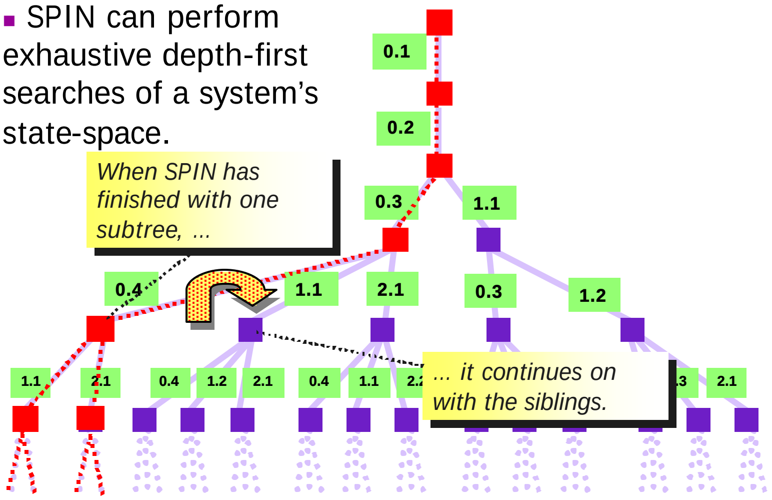


When SPIN has finished with one subtree, ...

... it continues on with the siblings.

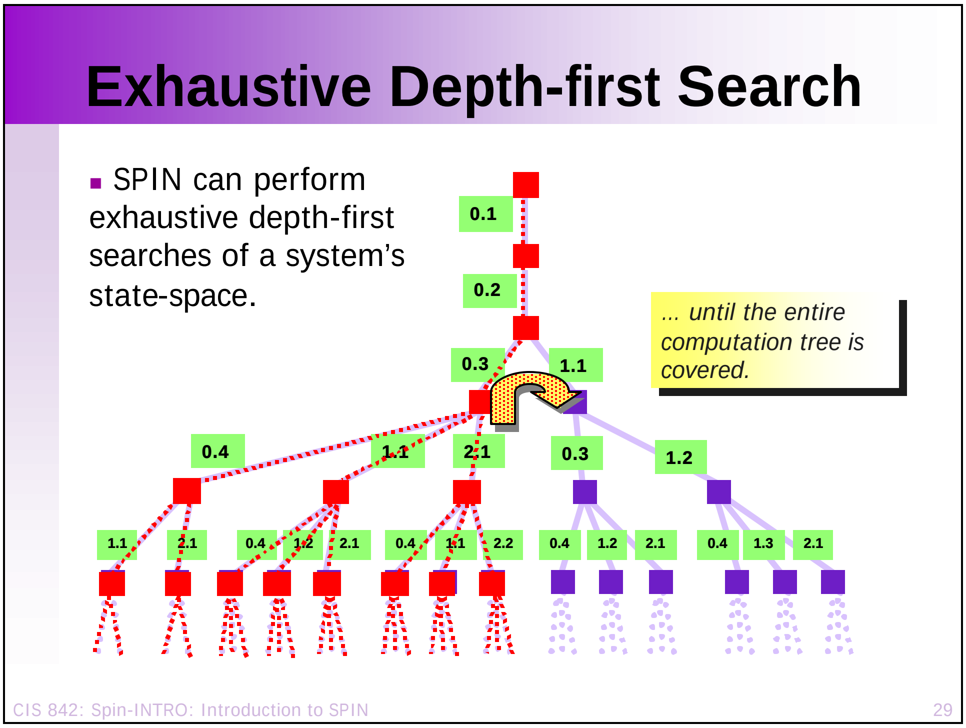
Exhaustive Depth-first Search

- SPIN can perform exhaustive depth-first searches of a system's state-space.



When SPIN has finished with one subtree, ...

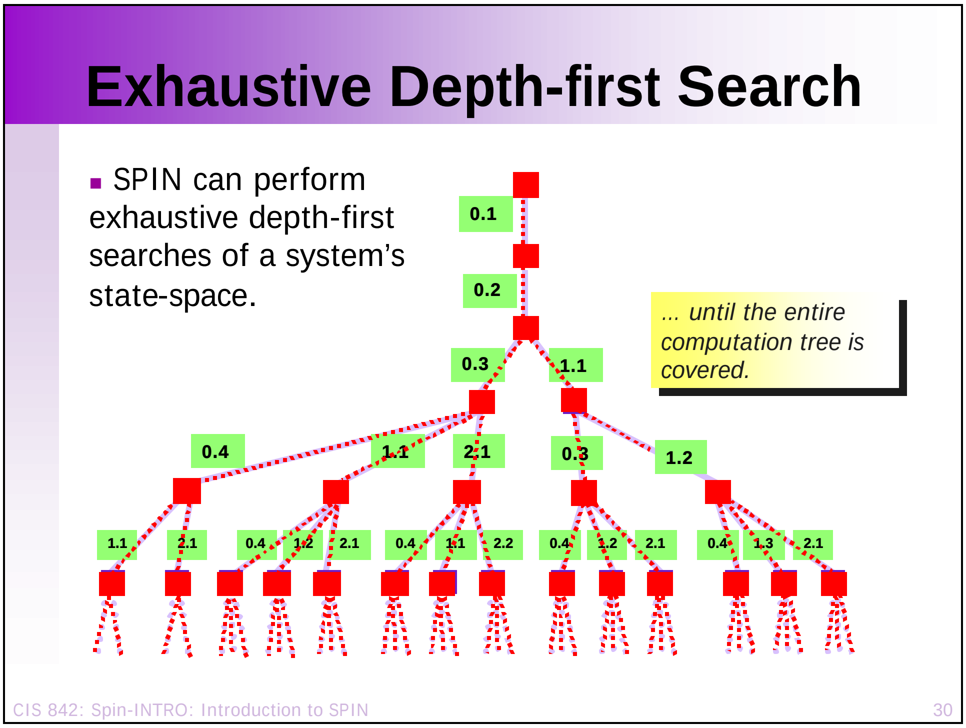
... it continues on with the siblings.



Exhaustive Depth-first Search

- SPIN can perform exhaustive depth-first searches of a system's state-space.

... until the entire computation tree is covered.

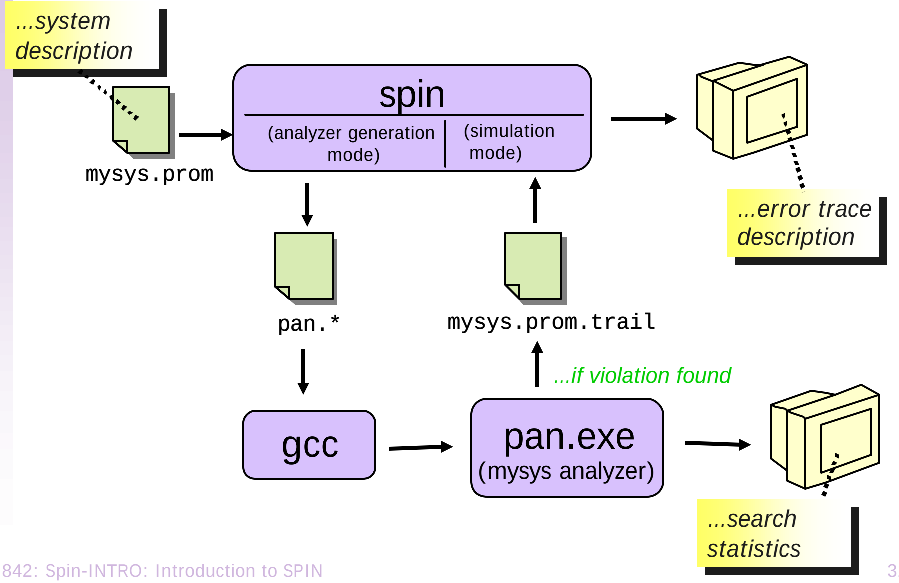


Exhaustive Depth-first Search

- SPIN can perform exhaustive depth-first searches of a system's state-space.

... until the entire computation tree is covered.

Process of Using SPIN



...system description

mysys.prom

spin

(analyzer generation mode)

(simulation mode)

...error trace description

pan.*

mysys.prom.trail

...if violation found

gcc

pan.exe

(mysys analyzer)

...search statistics

Process of Using SPIN

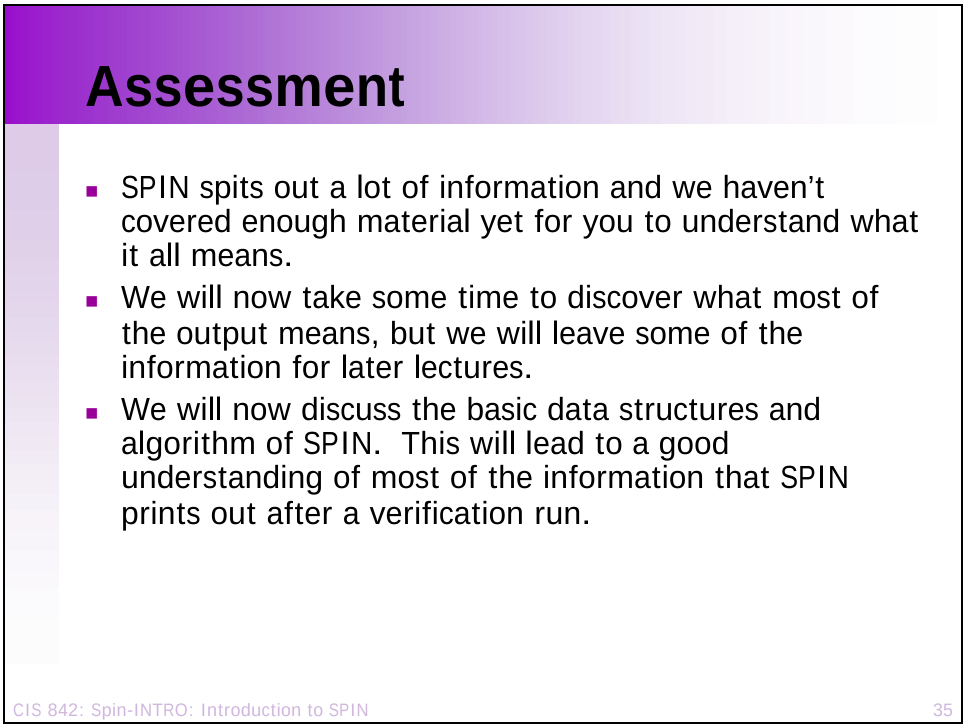
- `$ spin –a mysys.prom`
  - creates a dedicated PROMELA analyzer C program (`pan.*`) that implements an exhaustive search on the system described in `mysys.prom`
- `$ gcc pan.c –o pan.exe`
  - compiles the analyzer source (`pan.c`) to yield an executable (`pan.exe`)
  - user often supplies additional compiler flags (discussed later)
- `$ pan.exe`
  - runs the analyzer
  - user often supplies other command-line flags (discussed later)
  - produces `mysys.prom.trail` which indicates the particular execution trace that caused a property violation
- `$ spin –t mysys.prom`
  - runs SPIN in simulation mode along the trace indicated by `mysys.prom.trail`
  - also print out diagnostic information (variable values, execution steps) with various levels of verbosity (set by additional command-line args)
  - use this information to diagnose bug and fix the system defect

For You To Do…

- Pause the lecture…
- Edit `anyn.prom` and change the assertion to `x != 3`.
- Use SPIN as described on the previous slide to perform an exhaustive search for property violations on the anyn program.
- What happened? Try to figure out what SPIN's output is telling you.
  - Did SPIN find an execution path that causes the assertion to be violated?
  - Can you determine what the path is from SPIN's output?
  - Can you determine how long the path is (how many steps)?
  - What does the other information produced by SPIN tell you?

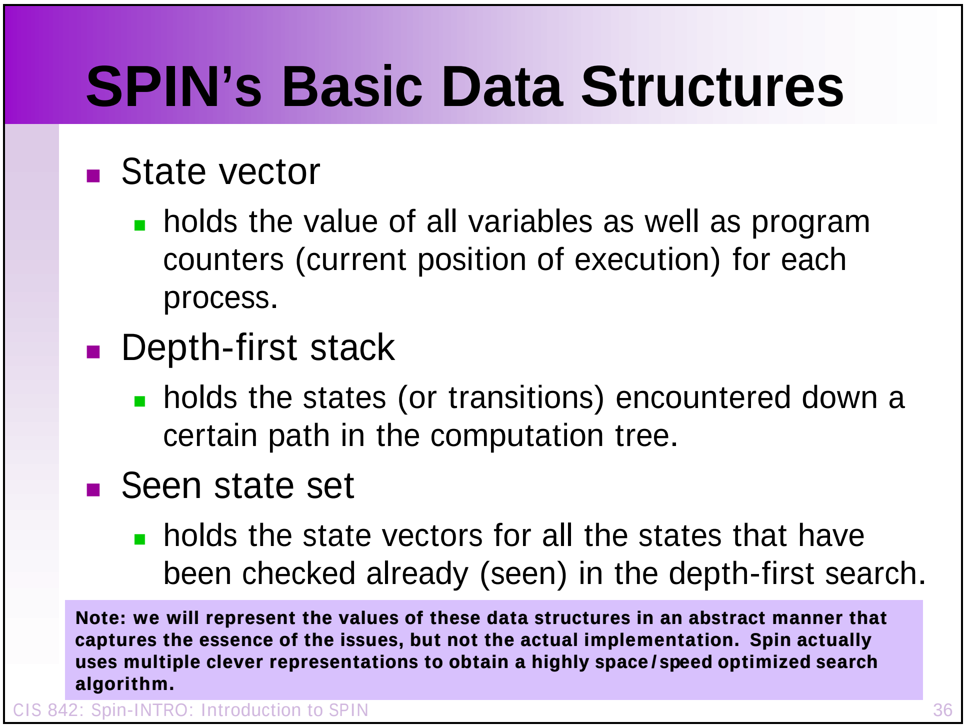
SPIN Output

```
pan: assertion violated (x!=3) (at depth 1358)
pan: wrote anyn.prom.trail
(Spin Version 3.4.16 -- 2 June 2002)
Warning: Search not completed
       + Partial Order Reduction

Full statespace search for:
       never-claim              - (none specified)
       assertion violations     +
       acceptance   cycles      - (not selected)
       invalid endstates   +

State-vector 24 byte, depth reached 3267, errors: 1
   14478 states, stored
   19167 states, matched
   33645 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 243 (resolved)
(max size 2^18 states)
```



Assessment

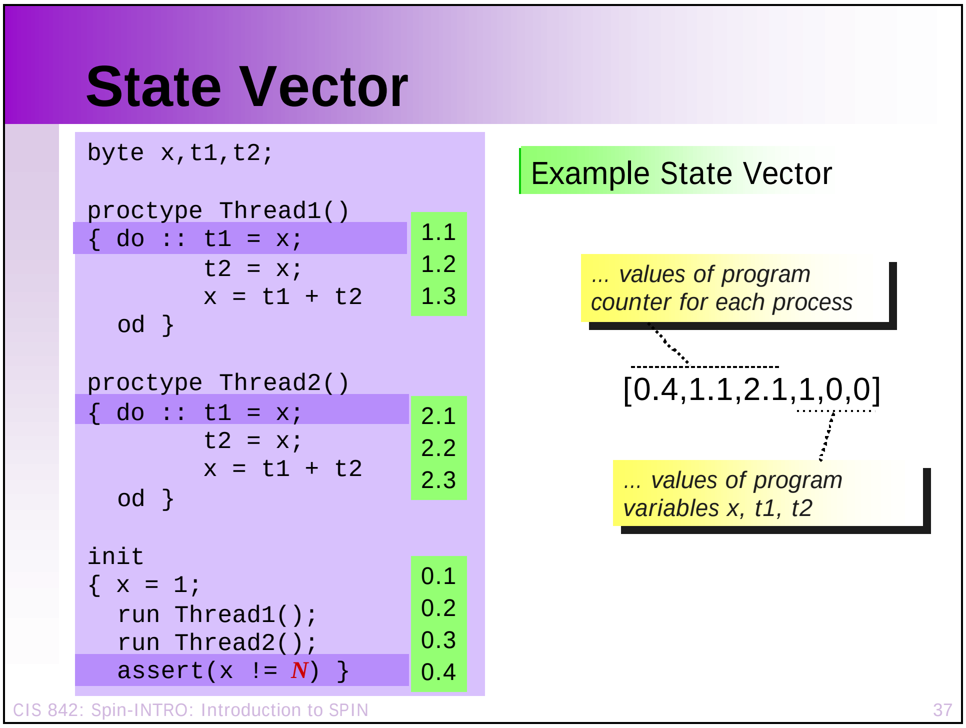
- SPIN spits out a lot of information and we haven’t covered enough material yet for you to understand what it all means.
- We will now take some time to discover what most of the output means, but we will leave some of the information for later lectures.
- We will now discuss the basic data structures and algorithm of SPIN. This will lead to a good understanding of most of the information that SPIN prints out after a verification run.



SPIN’s Basic Data Structures

- State vector
  - holds the value of all variables as well as program counters (current position of execution) for each process.
- Depth-first stack
  - holds the states (or transitions) encountered down a certain path in the computation tree.
- Seen state set
  - holds the state vectors for all the states that have been checked already (seen) in the depth-first search.

Note: we will represent the values of these data structures in an abstract manner that captures the essence of the issues, but not the actual implementation. Spin actually uses multiple clever representations to obtain a highly space/speed optimized search algorithm.



State Vector

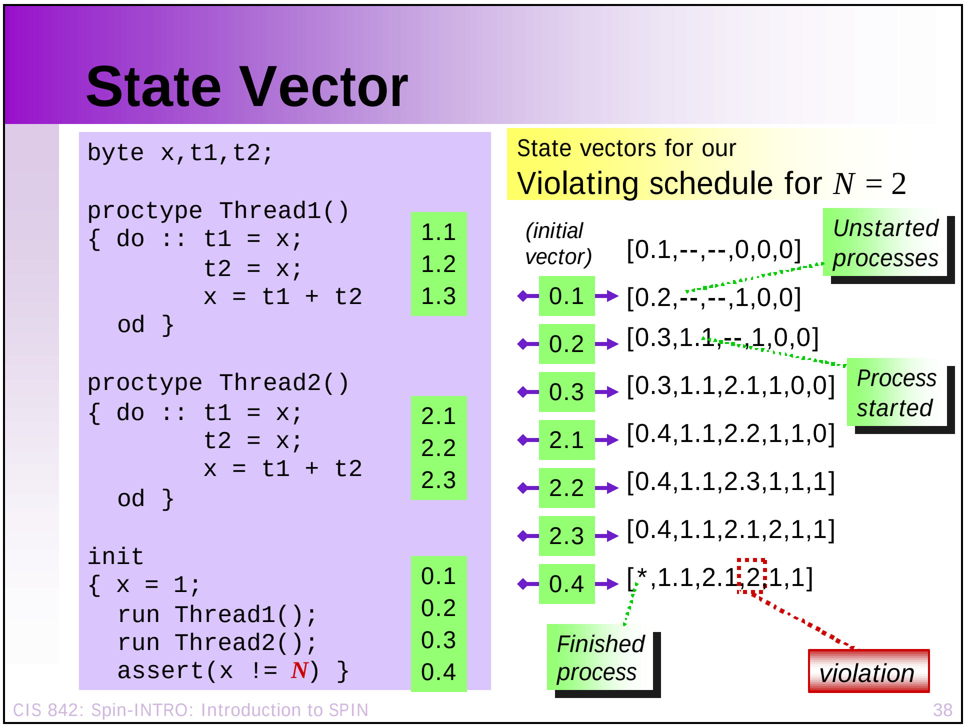
```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;           1.1
        t2 = x;           1.2
        x = t1 + t2       1.3
  od }

proctype Thread2()
{ do :: t1 = x;           2.1
        t2 = x;           2.2
        x = t1 + t2       2.3
  od }

init
{ x = 1;                  0.1
  run Thread1();          0.2
  run Thread2();          0.3
  assert(x != N) }        0.4
```

Example State Vector

... values of program counter for each process

[0.4,1.1,2.1,1,0,0]

... values of program variables x, t1, t2



State Vector

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;           1.1
        t2 = x;           1.2
        x = t1 + t2       1.3
  od }

proctype Thread2()
{ do :: t1 = x;           2.1
        t2 = x;           2.2
        x = t1 + t2       2.3
  od }

init
{ x = 1;                  0.1
  run Thread1();          0.2
  run Thread2();          0.3
  assert(x != N) }        0.4
```

State vectors for our

Violating schedule for $N = 2$

(initial vector) [0.1,--,--,0,0,0]

0.1 → [0.2,--,--,1,0,0]

0.2 → [0.3,1.1,--,1,0,0]

0.3 → [0.3,1.1,2.1,1,0,0]

2.1 → [0.4,1.1,2.2,1,1,0]

2.2 → [0.4,1.1,2.3,1,1,1]

2.3 → [0.4,1.1,2.1,2,1,1]

0.4 → [*,1.1,2.1,2,1,1]

Unstarted processes

Process started

Finished process

violation

SPIN Output

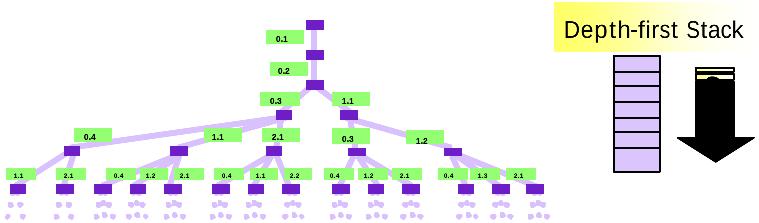
```
pan: assertion violated (x!=3) (at depth 1358)
pan: wrote anyn.prom.trail
(Spin Version 3.4.16 -- 2 June 2002)
Warning: Search not completed
       + Partial Order Reduction

Full statespace search for:
       never-claim                    - (none specified)
       assertion violations
       acceptance   cycles
       invalid endstates    +

State-vector 24 byte, depth reached 3267, errors: 1
   14478 states, stored
   19167 states, matched
   33645 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 243 (resolved)
(max size 2^18 states)
```

...indicates SPIN used 24 bytes to represent a state vector

For You To Do...

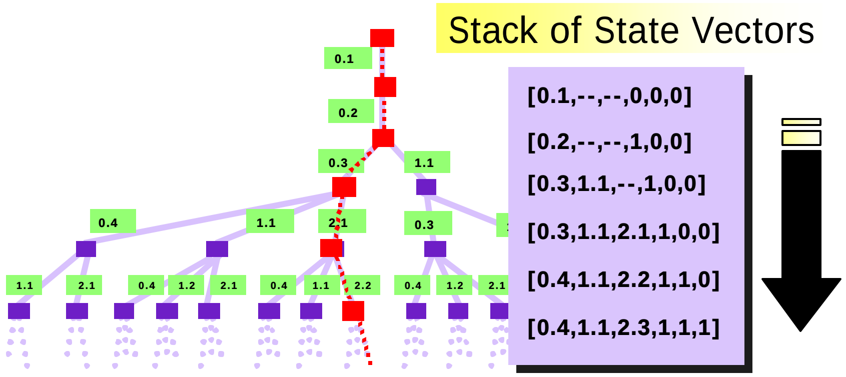
- Pause the lecture...
- Give the state vector sequence (as illustrated a few slides ago) for a schedule that leads to a violation of the assertion set to `assert(x != 3).`

Depth-first Stack



- The depth-first stack serves two purposes
  - When we come to the end of a path (or a state that we have seen before) and backtrack, the stack tells us where to backtrack to.
  - If an error is encountered, the current value of the stack gives the computation path that leads to the error.

Depth-first Stack



[0.1,--,--,0,0,0]

[0.2,--,--,1,0,0]

[0.3,1.1,--,1,0,0]

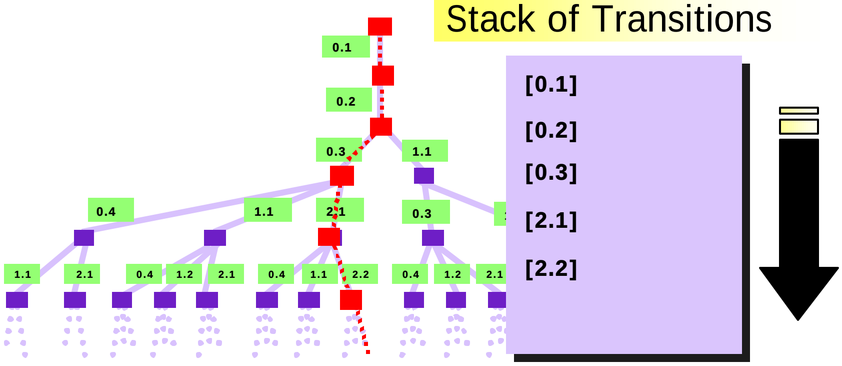
[0.3,1.1,2.1,1,0,0]

[0.4,1.1,2.2,1,1,0]

[0.4,1.1,2.3,1,1,1]

- The depth-first stack can be implemented to hold state vectors
  - straight-forward implementation

Depth-first Stack



Stack of Transitions

[0.1]

[0.2]

[0.3]

[2.1]

[2.2]

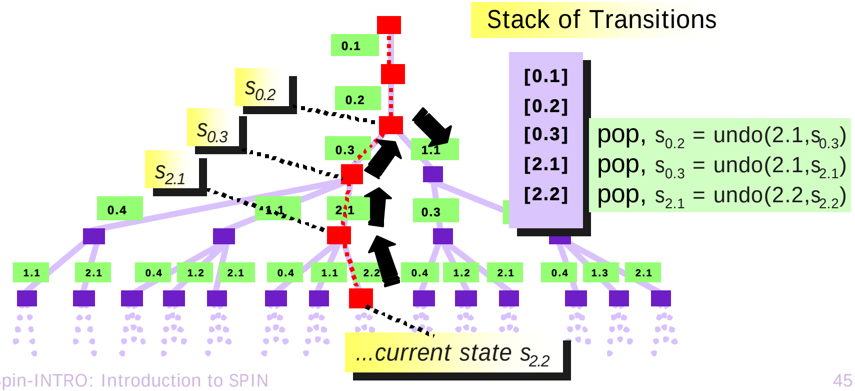
- The depth-first stack can be implemented to hold transitions
  - requires less space, but ...(see next slide)...

Depth-first Stack of Transitions

- Generating a new state requires that the analyzer run a transition on the current state.
- Since the analyzer is not holding states in the stack, if it needs to back-track and return to a previously encountered state, it needs an “undo” operation to run the transitions in the reverse direction.
- Since the analyzer is not holding states in the stack, when providing variable values as diagnostic information for an error path, the analyzer needs a simulation mode where choice points are decided by the stacked transitions.

Depth-first Stack of Transitions

- Since the analyzer is not holding states in the stack, if it needs to back-track and return to a previously encountered state, it needs an “undo” operation to run the transitions in the reverse direction.



Stack of Transitions

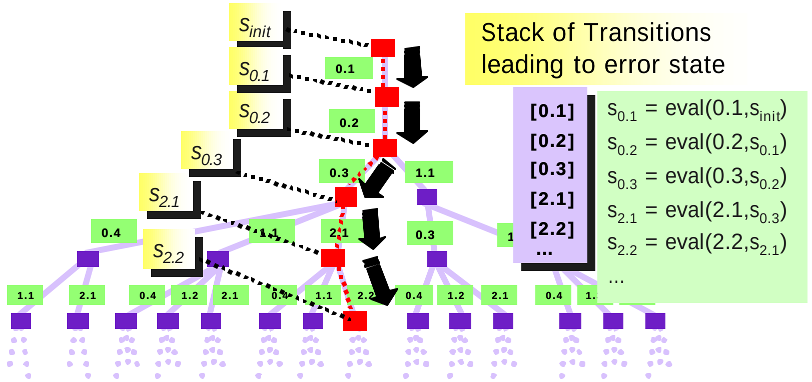
[0.1]
[0.2]
[0.3]
[2.1]
[2.2]

pop, $s_{0.2}$ = undo(2.1,$s_{0.3}$)
pop, $s_{0.3}$ = undo(2.1,$s_{2.1}$)
pop, $s_{2.1}$ = undo(2.2,$s_{2.2}$)

...current state $s_{2.2}$

Depth-first Stack of Transitions

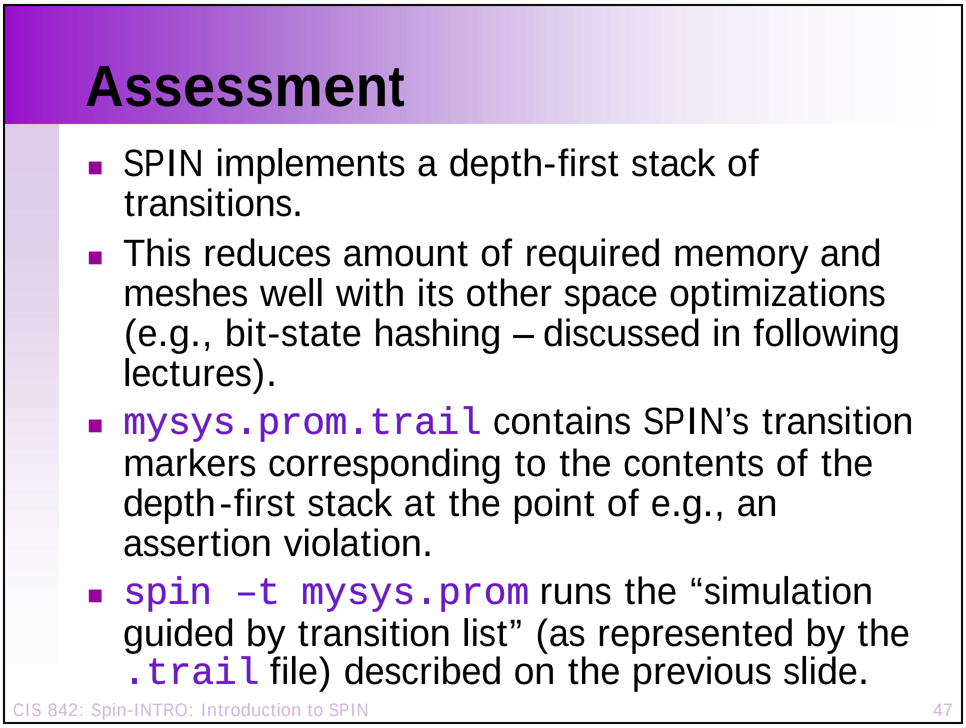
- Since the analyzer is not holding states in the stack, when providing variable values as diagnostic information for an error path, the analyzer needs a simulation mode where choice points are decided by the transitions



Stack of Transitions leading to error state

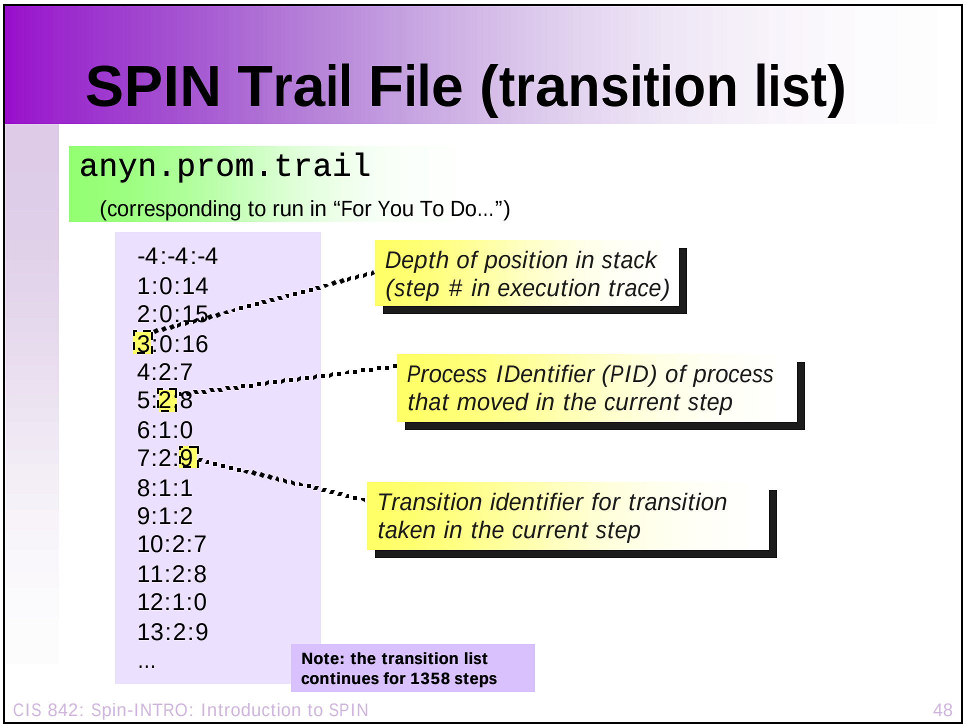
[0.1]
[0.2]
[0.3]
[2.1]
[2.2]
...

$s_{0.1}$ = eval(0.1,s_{init})
$s_{0.2}$ = eval(0.2,$s_{0.1}$)
$s_{0.3}$ = eval(0.3,$s_{0.2}$)
$s_{2.1}$ = eval(2.1,$s_{0.3}$)
$s_{2.2}$ = eval(2.2,$s_{2.1}$)
...



Assessment

- SPIN implements a depth-first stack of transitions.
- This reduces amount of required memory and meshes well with its other space optimizations (e.g., bit-state hashing – discussed in following lectures).
- `mysys.prom.trail` contains SPIN's transition markers corresponding to the contents of the depth-first stack at the point of e.g., an assertion violation.
- `spin –t mysys.prom` runs the "simulation guided by transition list" (as represented by the `.trail` file) described on the previous slide.



SPIN Trail File (transition list)

`anyn.prom.trail`

(corresponding to run in "For You To Do…")

```
-4:-4:-4
1:0:14
2:0:15
3:0:16
4:2:7
5:2:8
6:1:0
7:2:9
8:1:1
9:1:2
10:2:7
11:2:8
12:1:0
13:2:9
…
```

Depth of position in stack (step # in execution trace)

Process IDentifier (PID) of process that moved in the current step

Transition identifier for transition taken in the current step

Note: the transition list continues for 1358 steps

SPIN Output

```
pan: assertion violated (x!=3) (at depth 1358)
pan: wrote anyn.prom.trail
(Spin Version 3.4.16 -- 2 June 2002)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim              - (no
        assertion violations     +
        acceptance   cycles      - (not selected)
        invalid endstates   +

State-vector 24 byte, depth reached 3267, errors: 1
   14478 states, stored
   19167 states, matched
   33645 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 243 (resolved)
(max size 2^18 states)
```

...depth in computation tree (i.e., transition stack) where assertion violation was found (i.e., number of steps in error trace)

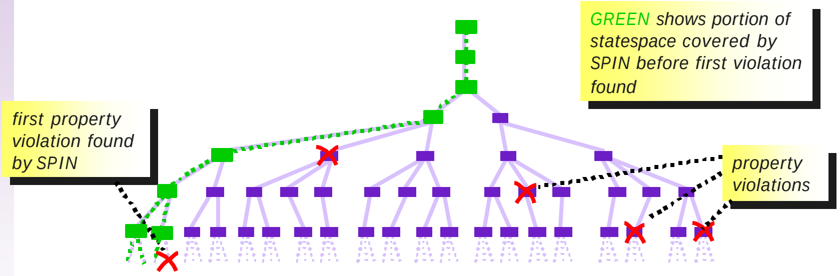
...deepest stack depth reached during search

Error Trace Length

- From SPIN's output, we can see that it has found a execution trace that violates the assertion and that the trace is 1358 steps long (using SPIN version 3.4.16).
  - Having to reason about how the assertion can be violated along a trace of 1358 steps is quite painful!
  - You have previously discovered a much shorter violating trace using SPIN's simulation mode.
  - Does this mean that the SPIN analyzer is not very useful?
    - Not at all!!
- We will see in a little bit how to tell SPIN to search for shorter violating traces (as well as minimal length violating traces) .

Error Trace Length

- In general, a system may have many different traces that lead to the same property violation.


GREEN shows portion of statespace covered by SPIN before first violation found
first property violation found by SPIN
property violations

- Because SPIN does a depth-first search (instead of a bread-first search), the first violating trace that it finds is usually not of minimal length.

Setting SPIN’s Depth Bound

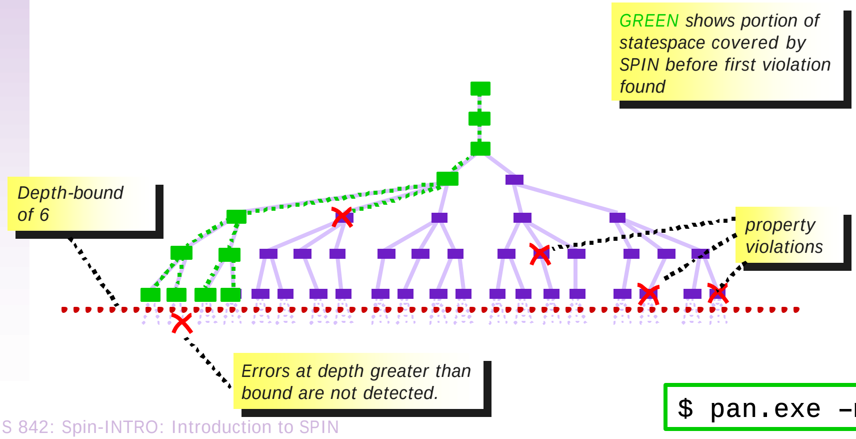
- Users can set a bound on the depth of SPIN’s search (i.e., entries in SPIN’s depth-first stack)
  - `pan.exe –mDEPTHBOUND`
- This is often useful…
  - …after a counterexample has been found and you want to see if a shorter one exists.
    - look at SPIN’s output to see the size, then rerun pan.exe with an appropriate depth bound.
  - …before a counterexample has been found and SPIN is taking too long or is running out of memory.

Setting SPIN's Depth Bound

- Be careful!
  - when you bound SPIN's search, SPIN will not be exploring part of the system state-space, and the omitted part may contain property violations that you want to detect.
  - If SPIN tells you that there are no violations, but you have bounded the search, you *cannot assume that the system has no violations.* You only know that SPIN has found no violations in the part of the state-space that it searched.
  - SPIN displays "`error: max search depth too small`" to let you know that the depth bound prevented it from searching the complete state-space.

Bounded Depth-first Search

- When analyzing a system and given a depth bound as a command-line argument, SPIN will backtrack when that depth is reached.



GREEN *shows portion of statespace covered by SPIN before first violation found*

Depth-bound of 6

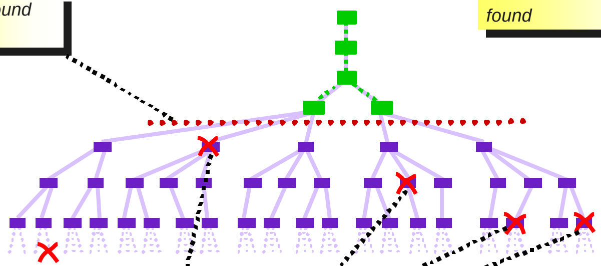
property violations

Errors at depth greater than bound are not detected.

```
$ pan.exe –m6
```

Bounded Depth-first Search

- When analyzing a system and given a depth bound as a command-line argument, SPIN will backtrack when that depth is reached.



GREEN shows portion of statespace covered by SPIN before first violation found

Depth-bound of 3

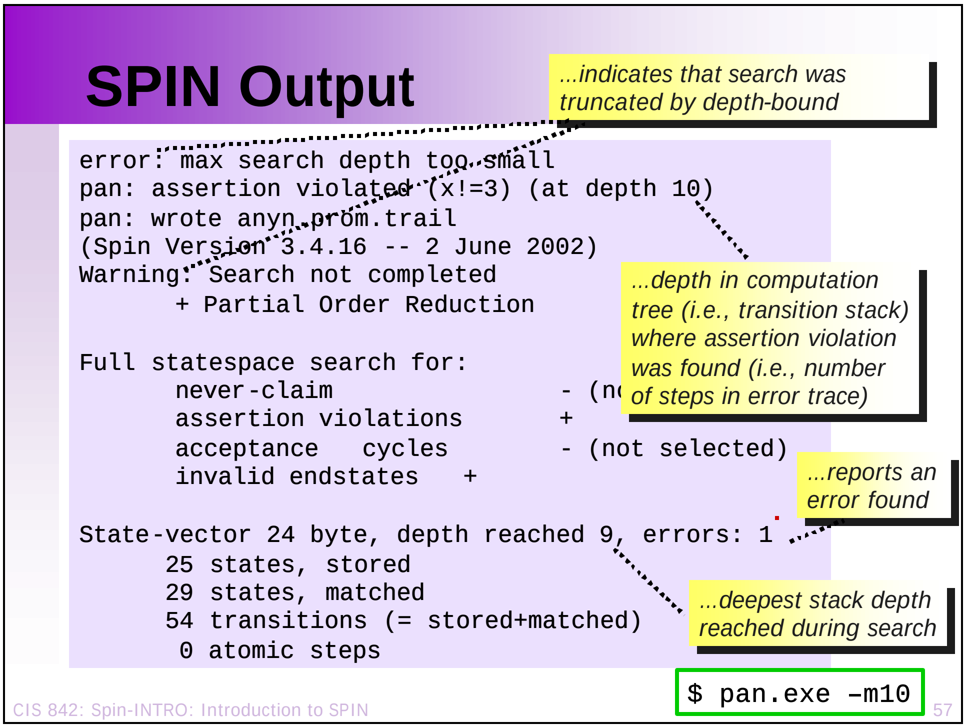
In this case, SPIN reports that no violations were found.

Errors at depth greater than bound are not detected.

```
$ pan.exe –m3
```

For You To Do…

- Pause the lecture…
- Edit `anyn.prom` and change the assertion to `x != 3.`
- Use SPIN (pan.exe –mDEPTHBOUND) to find an error trace of minimal length.
  - start with a depth bound that allows an error
  - successively choose smaller versions of the bound until SPIN reports no error
  - determine a bound B such that running SPIN with bound B-1 reveals no errors, but running with B reveals an error
- How does this error trace compare to the one (i.e., size and state vectors encountered) to the error trace that you discovered earlier using SPIN's guided simulation mode?
- Note that there may be multiple minimal length error traces.



SPIN Output

...indicates that search was truncated by depth-bound

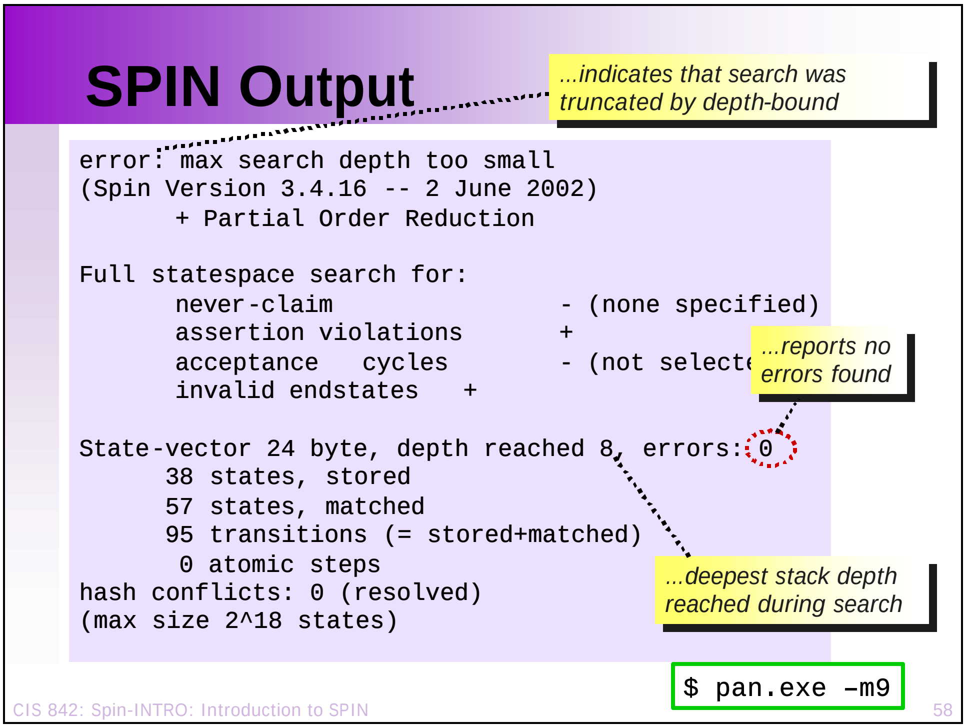
```
error: max search depth too small
pan: assertion violated (x!=3) (at depth 10)
pan: wrote anyn_prom.trail
(Spin Version 3.4.16 -- 2 June 2002)
Warning: Search not completed
       + Partial Order Reduction

Full statespace search for:
       never-claim                - (n
       assertion violations       +
       acceptance   cycles        - (not selected)
       invalid endstates   +

State-vector 24 byte, depth reached 9, errors: 1
      25 states, stored
      29 states, matched
      54 transitions (= stored+matched)
       0 atomic steps
```

...depth in computation tree (i.e., transition stack) where assertion violation was found (i.e., number of steps in error trace)

...reports an error found

...deepest stack depth reached during search

```
$ pan.exe –m10
```



SPIN Output

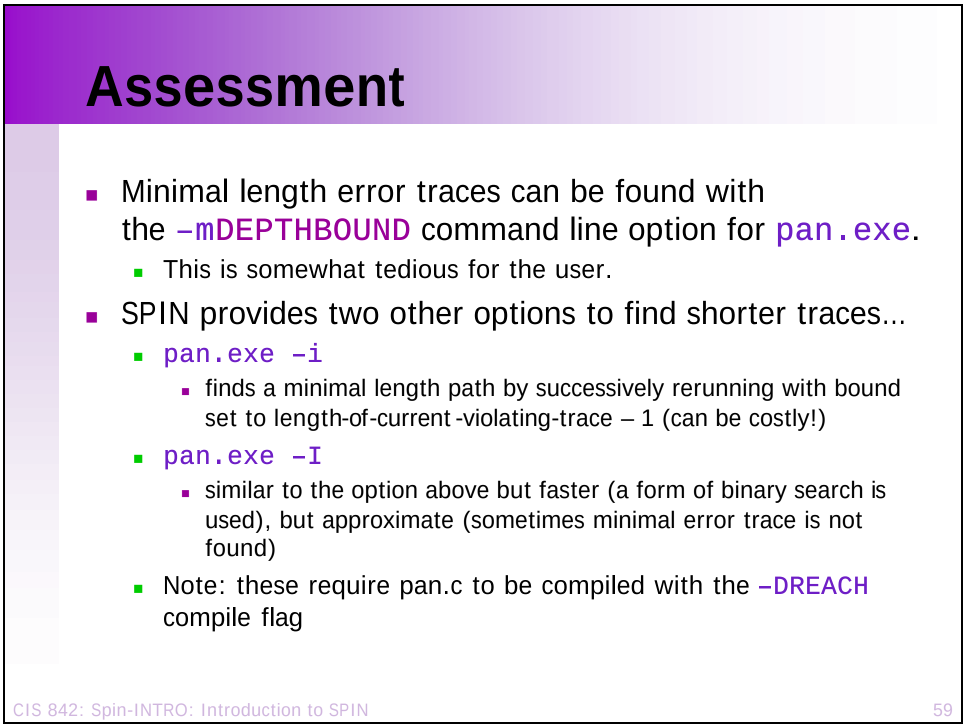
...indicates that search was truncated by depth-bound

```
error: max search depth too small
(Spin Version 3.4.16 -- 2 June 2002)
       + Partial Order Reduction

Full statespace search for:
       never-claim                - (none specified)
       assertion violations       +
       acceptance   cycles        - (not selecte
       invalid endstates   +

State-vector 24 byte, depth reached 8, errors: 0
      38 states, stored
      57 states, matched
      95 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
```

...reports no errors found

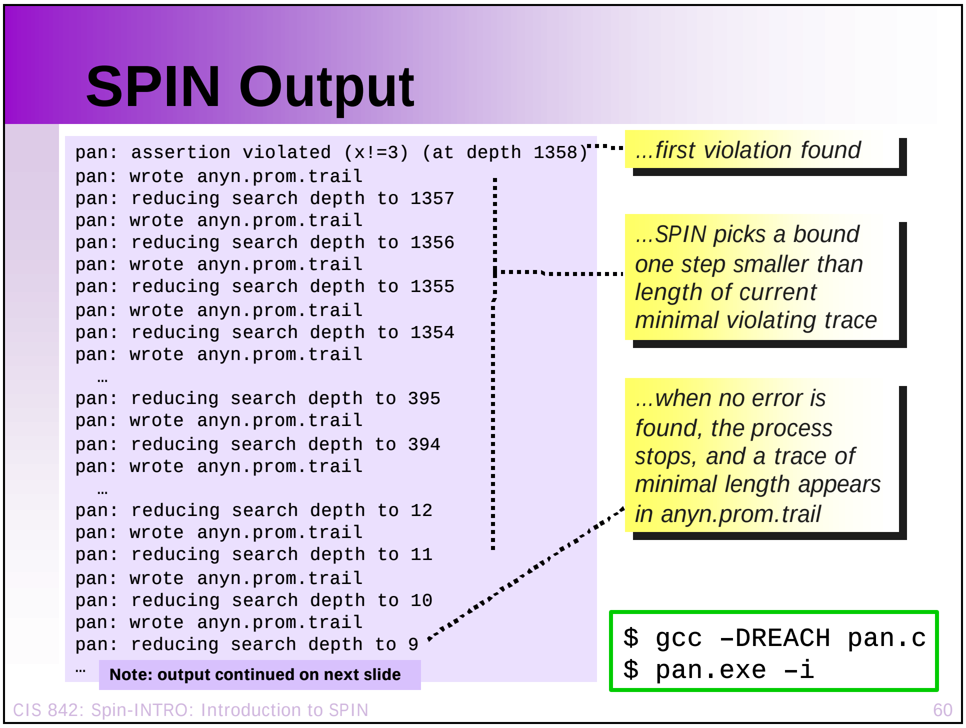
...deepest stack depth reached during search

```
$ pan.exe –m9
```



Assessment

- Minimal length error traces can be found with the –mDEPTHBOUND command line option for pan.exe.
  - This is somewhat tedious for the user.
- SPIN provides two other options to find shorter traces…
  - pan.exe –i
    - finds a minimal length path by successively rerunning with bound set to length-of-current-violating-trace – 1 (can be costly!)
  - pan.exe –I
    - similar to the option above but faster (a form of binary search is used), but approximate (sometimes minimal error trace is not found)
  - Note: these require pan.c to be compiled with the –DREACH compile flag



SPIN Output

```
pan: assertion violated (x!=3) (at depth 1358)
pan: wrote anyn.prom.trail
pan: reducing search depth to 1357
pan: wrote anyn.prom.trail
pan: reducing search depth to 1356
pan: wrote anyn.prom.trail
pan: reducing search depth to 1355
pan: wrote anyn.prom.trail
pan: reducing search depth to 1354
pan: wrote anyn.prom.trail
  …
pan: reducing search depth to 395
pan: wrote anyn.prom.trail
pan: reducing search depth to 394
pan: wrote anyn.prom.trail
  …
pan: reducing search depth to 12
pan: wrote anyn.prom.trail
pan: reducing search depth to 11
pan: wrote anyn.prom.trail
pan: reducing search depth to 10
pan: wrote anyn.prom.trail
pan: reducing search depth to 9
…
```

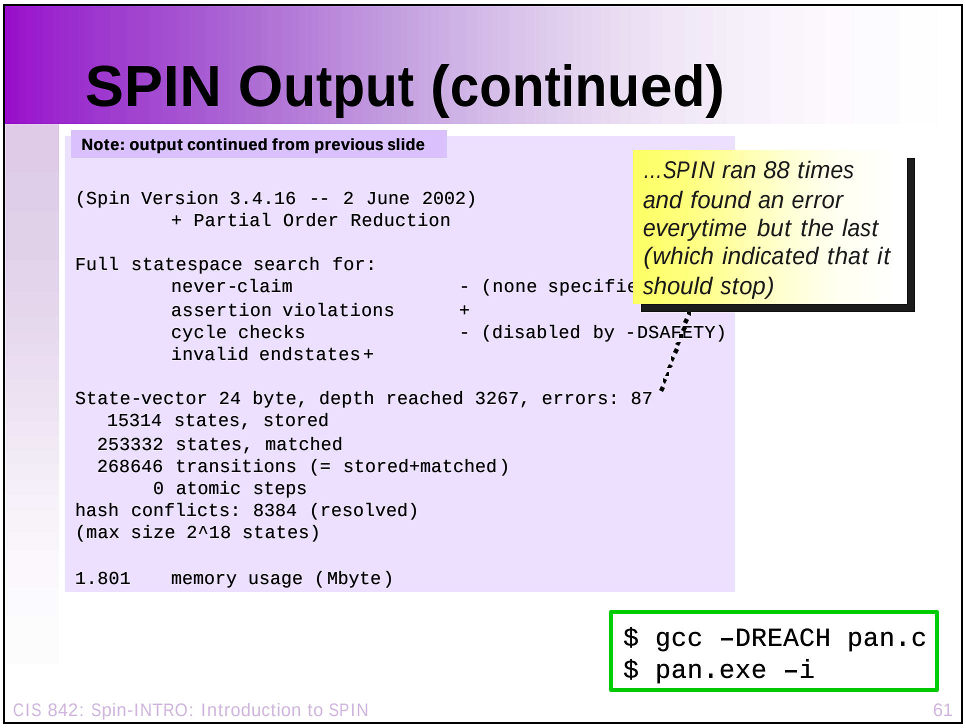
Note: output continued on next slide

…first violation found

…SPIN picks a bound one step smaller than length of current minimal violating trace

…when no error is found, the process stops, and a trace of minimal length appears in anyn.prom.trail

```
$ gcc –DREACH pan.c
$ pan.exe –i
```



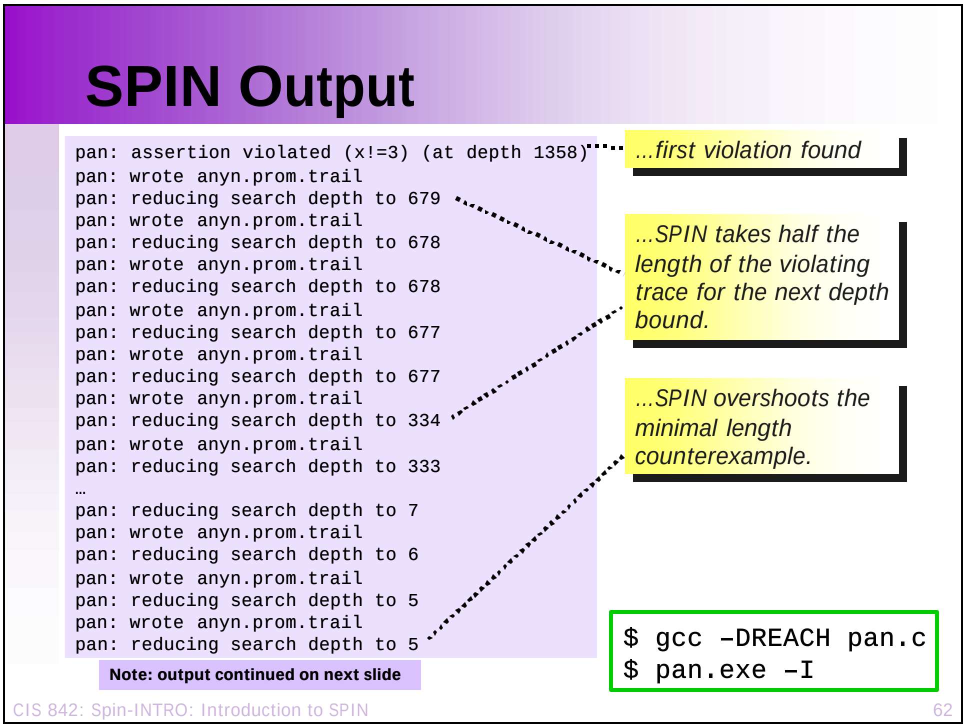
SPIN Output (continued)

Note: output continued from previous slide

```
(Spin Version 3.4.16 -- 2 June 2002)
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specifie
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates+

State-vector 24 byte, depth reached 3267, errors: 87
   15314 states, stored
  253332 states, matched
  268646 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 8384 (resolved)
(max size 2^18 states)

1.801   memory usage (Mbyte)
```

...SPIN ran 88 times and found an error everytime but the last (which indicated that it should stop)

```
$ gcc –DREACH pan.c
$ pan.exe –i
```



SPIN Output

```
pan: assertion violated (x!=3) (at depth 1358)
pan: wrote anyn.prom.trail
pan: reducing search depth to 679
pan: wrote anyn.prom.trail
pan: reducing search depth to 678
pan: wrote anyn.prom.trail
pan: reducing search depth to 678
pan: wrote anyn.prom.trail
pan: reducing search depth to 677
pan: wrote anyn.prom.trail
pan: reducing search depth to 677
pan: wrote anyn.prom.trail
pan: reducing search depth to 334
pan: wrote anyn.prom.trail
pan: reducing search depth to 333
…
pan: reducing search depth to 7
pan: wrote anyn.prom.trail
pan: reducing search depth to 6
pan: wrote anyn.prom.trail
pan: reducing search depth to 5
pan: wrote anyn.prom.trail
pan: reducing search depth to 5
```

Note: output continued on next slide

...first violation found

...SPIN takes half the length of the violating trace for the next depth bound.

...SPIN overshoots the minimal length counterexample.

```
$ gcc –DREACH pan.c
$ pan.exe –I
```

SPIN Output (continued)

Note: output continued from previous slide

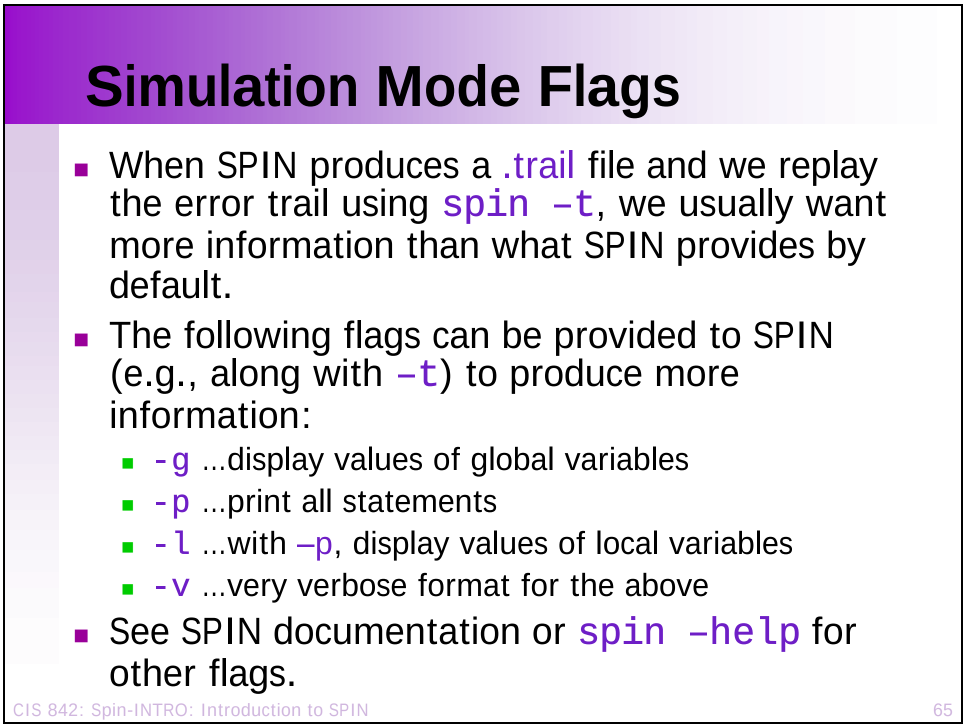
```
(Spin Version 3.4.16 -- 2 June 2002)
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specifie
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates+

State-vector 24 byte, depth reached 3267, errors: 22
   14828 states, stored
  157190 states, matched
  172018 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 3980 (resolved)
(max size 2^18 states)

1.801   memory usage (Mbyte)
```

…SPIN ran 23 times and found an error everytime but the last (which indicated that it should stop)

```
$ gcc –DREACH pan.c
$ pan.exe –I
```

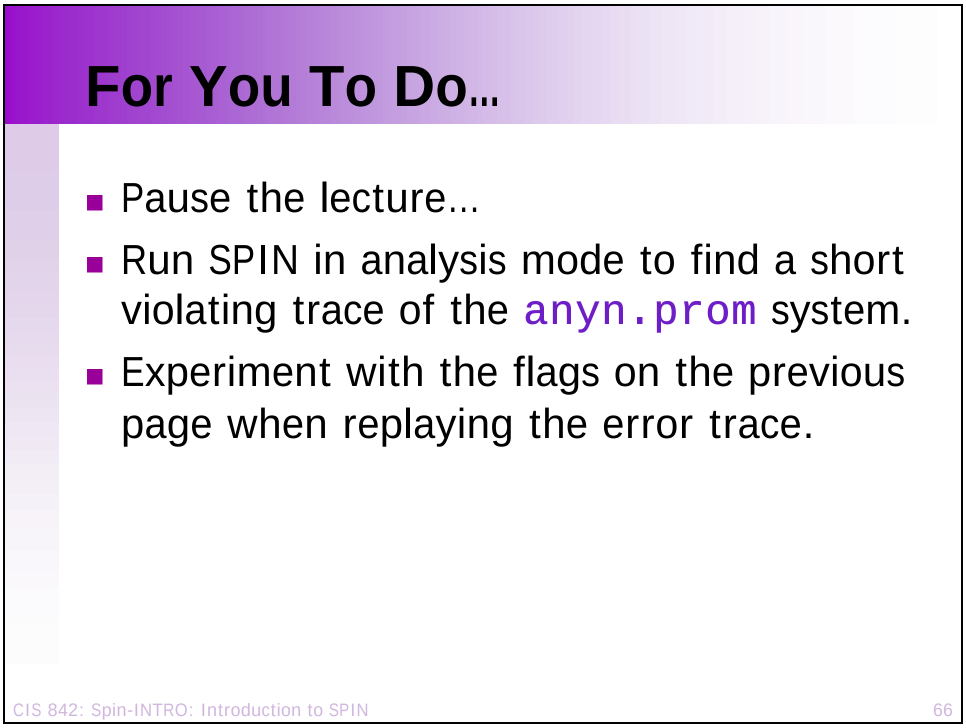
For You To Do…

- Pause the lecture…
- Edit `anyn.prom` and change the assertion to `x != 6.`
- Use SPIN (pan.exe –i) to find an error trace of minimal length.
- Use SPIN (pan.exe –I) to find a short error trace.
- Did the two options above produce the same error trace at completion?
- Did the two options run the analyzer the same number of times?



Simulation Mode Flags

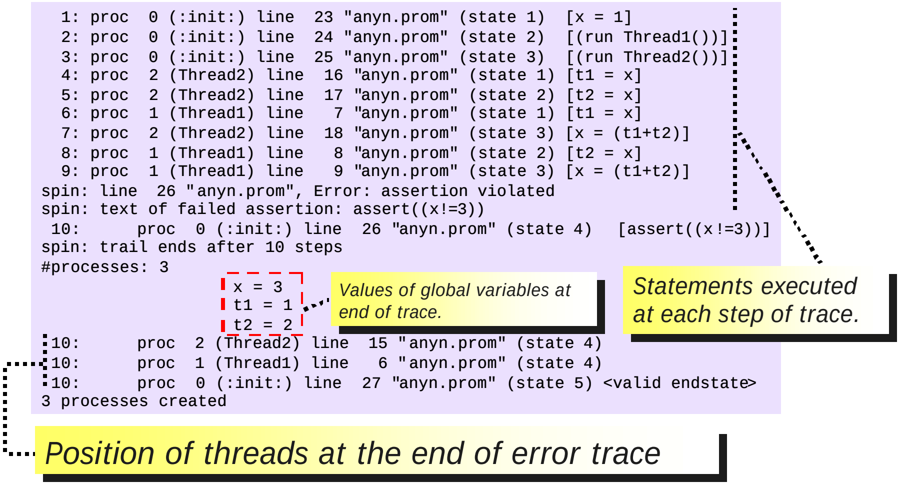
- When SPIN produces a .trail file and we replay the error trail using `spin –t`, we usually want more information than what SPIN provides by default.
- The following flags can be provided to SPIN (e.g., along with `–t`) to produce more information:
  - `-g` ...display values of global variables
  - `-p` ...print all statements
  - `-l` ...with –p, display values of local variables
  - `-v` ...very verbose format for the above
- See SPIN documentation or `spin –help` for other flags.



For You To Do...

- Pause the lecture...
- Run SPIN in analysis mode to find a short violating trace of the `anyn.prom` system.
- Experiment with the flags on the previous page when replaying the error trace.

Error Trail



```
   1: proc  0 (:init:) line  23 "anyn.prom" (state 1)  [x = 1]
   2: proc  0 (:init:) line  24 "anyn.prom" (state 2)  [(run Thread1())]
   3: proc  0 (:init:) line  25 "anyn.prom" (state 3)  [(run Thread2())]
   4: proc  2 (Thread2) line  16 "anyn.prom" (state 1) [t1 = x]
   5: proc  2 (Thread2) line  17 "anyn.prom" (state 2) [t2 = x]
   6: proc  1 (Thread1) line   7 "anyn.prom" (state 1) [t1 = x]
   7: proc  2 (Thread2) line  18 "anyn.prom" (state 3) [x = (t1+t2)]
   8: proc  1 (Thread1) line   8 "anyn.prom" (state 2) [t2 = x]
   9: proc  1 (Thread1) line   9 "anyn.prom" (state 3) [x = (t1+t2)]
spin: line  26 "anyn.prom", Error: assertion violated
spin: text of failed assertion: assert((x!=3))
 10:      proc  0 (:init:) line  26 "anyn.prom" (state 4)  [assert((x!=3))]
spin: trail ends after 10 steps
#processes: 3
                  x = 3
                  t1 = 1
                  t2 = 2
 10:      proc  2 (Thread2) line  15 "anyn.prom" (state 4)
 10:      proc  1 (Thread1) line   6 "anyn.prom" (state 4)
 10:      proc  0 (:init:) line  27 "anyn.prom" (state 5) <valid endstate>
3 processes created
```

Values of global variables at end of trace.

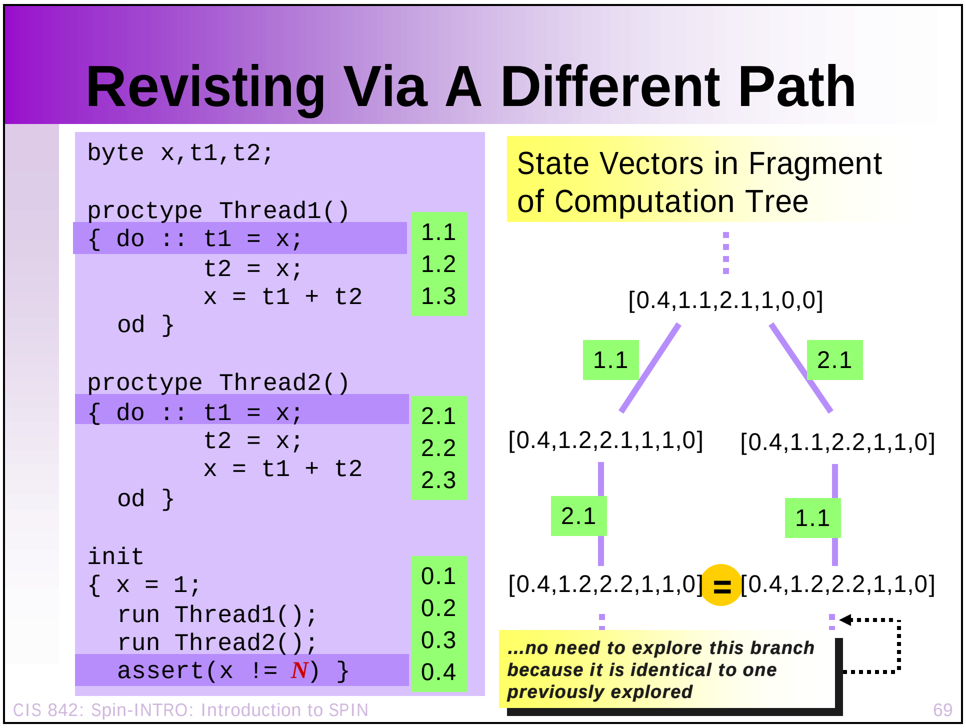
Statements executed at each step of trace.

Position of threads at the end of error trace

```
$ spin –t –p anyn.prom
```

Seen State Set

- Often the analyzer will proceed along a different path to a state S that it has checked before.
- In such a case, there is no need to check S again (or any of S's children in the computation tree) since these have been checked before.
- SPIN maintains a Seen State set (implemented as a hash table) of states that have been seen before, and it consults this set to avoid exploring/checking a part of the computation tree that is identical to a part that has already been explored before.

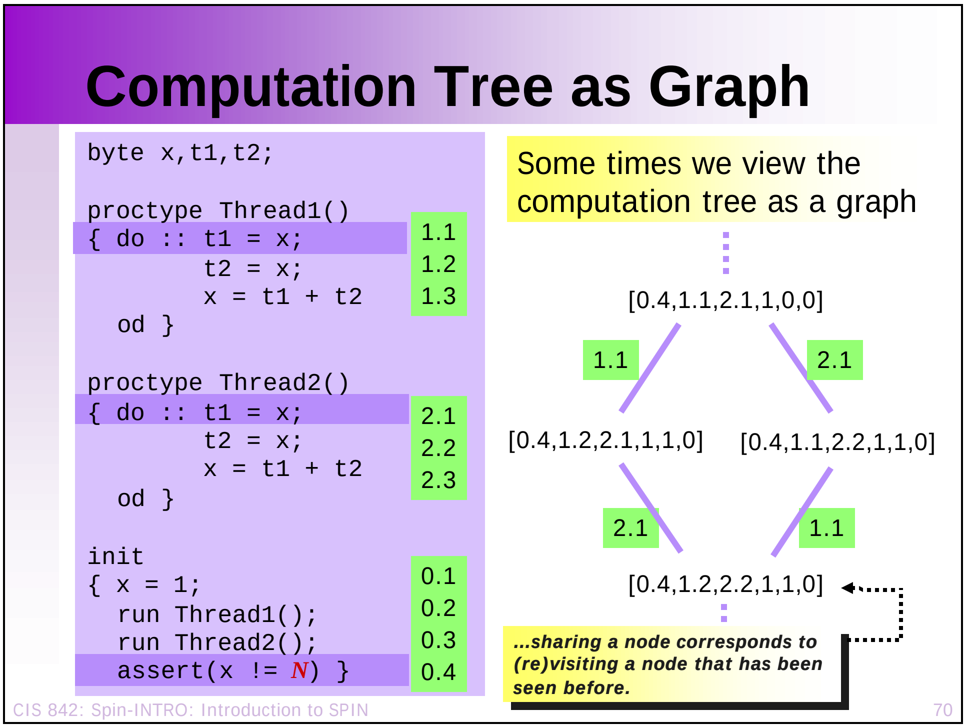


Revisting Via A Different Path

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;            1.1
        t2 = x;            1.2
        x = t1 + t2        1.3
  od }

proctype Thread2()
{ do :: t1 = x;            2.1
        t2 = x;            2.2
        x = t1 + t2        2.3
  od }

init
{ x = 1;                   0.1
  run Thread1();           0.2
  run Thread2();           0.3
  assert(x != N) }         0.4
```

State Vectors in Fragment of Computation Tree

...no need to explore this branch because it is identical to one previously explored

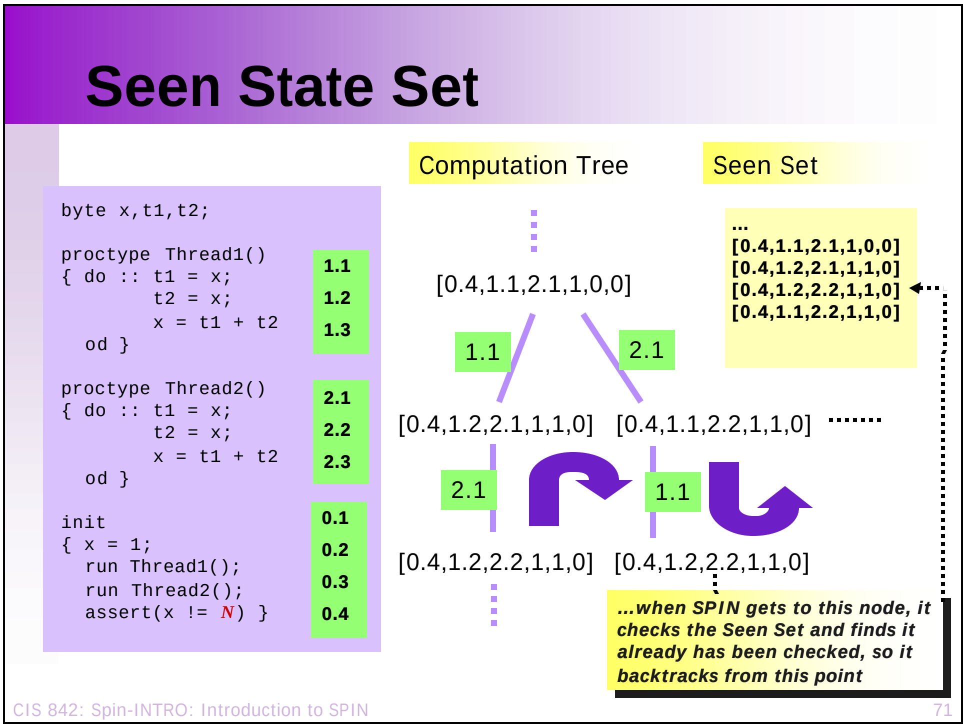


Computation Tree as Graph

```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;            1.1
        t2 = x;            1.2
        x = t1 + t2        1.3
  od }

proctype Thread2()
{ do :: t1 = x;            2.1
        t2 = x;            2.2
        x = t1 + t2        2.3
  od }

init
{ x = 1;                   0.1
  run Thread1();           0.2
  run Thread2();           0.3
  assert(x != N) }         0.4
```

Some times we view the computation tree as a graph

...sharing a node corresponds to (re)visiting a node that has been seen before.



Seen State Set

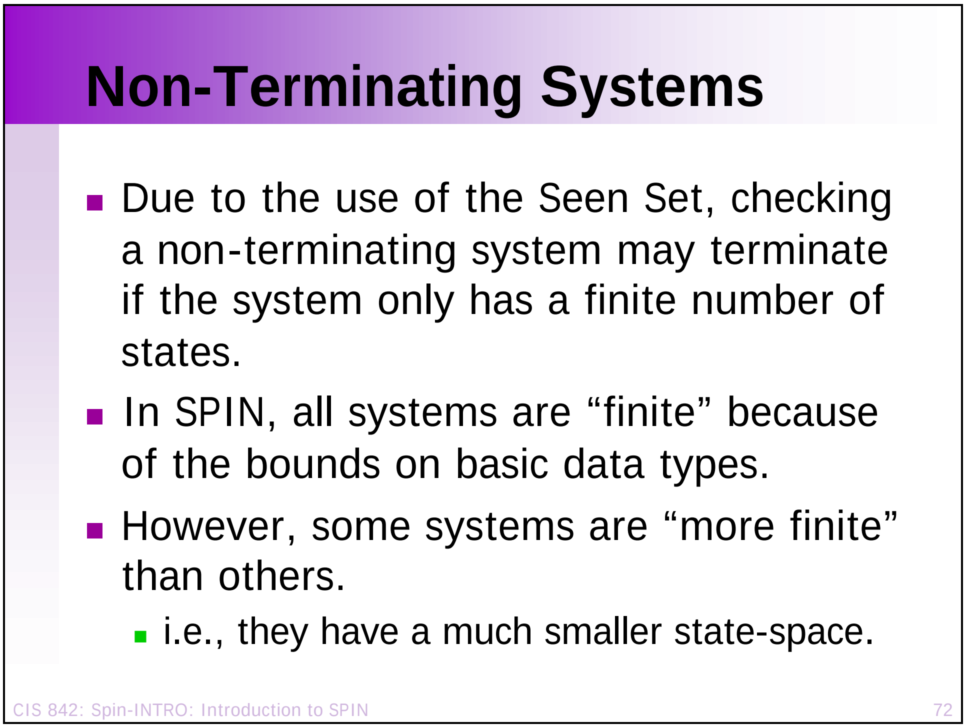
```
byte x,t1,t2;

proctype Thread1()
{ do :: t1 = x;          1.1
        t2 = x;          1.2
        x = t1 + t2      1.3
  od }

proctype Thread2()
{ do :: t1 = x;          2.1
        t2 = x;          2.2
        x = t1 + t2      2.3
  od }

init
{ x = 1;                 0.1
  run Thread1();         0.2
  run Thread2();         0.3
  assert(x != N) }       0.4
```

Computation Tree

Seen Set

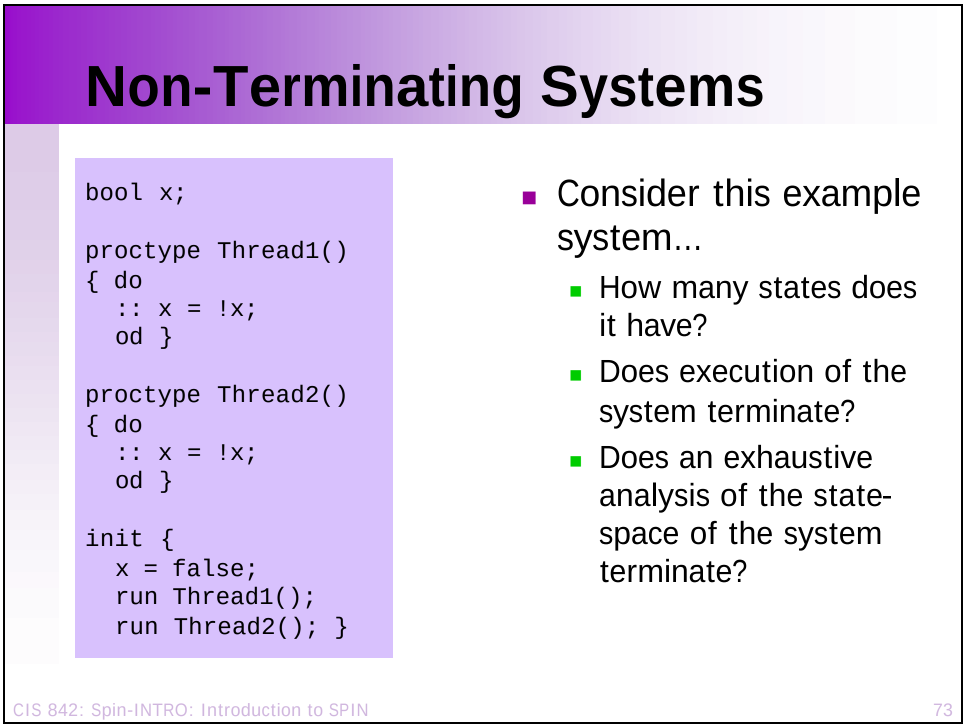
...
[0.4,1.1,2.1,1,0,0]
[0.4,1.2,2.1,1,1,0]
[0.4,1.2,2.2,1,1,0]
[0.4,1.1,2.2,1,1,0]

...when SPIN gets to this node, it checks the Seen Set and finds it already has been checked, so it backtracks from this point



Non-Terminating Systems

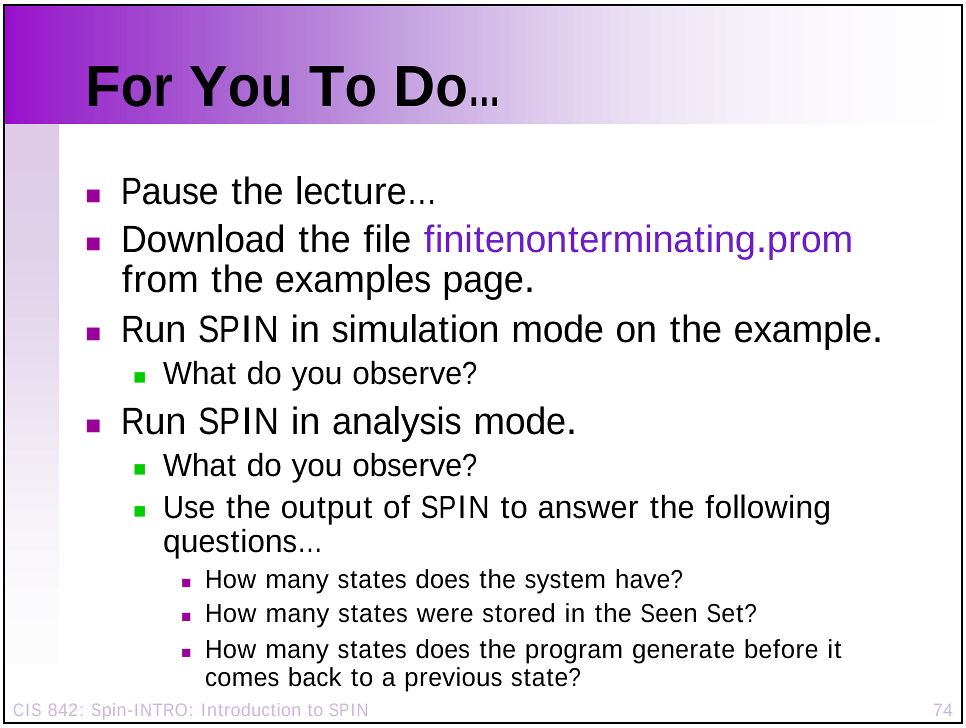
- Due to the use of the Seen Set, checking a non-terminating system may terminate if the system only has a finite number of states.
- In SPIN, all systems are “finite” because of the bounds on basic data types.
- However, some systems are “more finite” than others.
  - i.e., they have a much smaller state-space.



Non-Terminating Systems

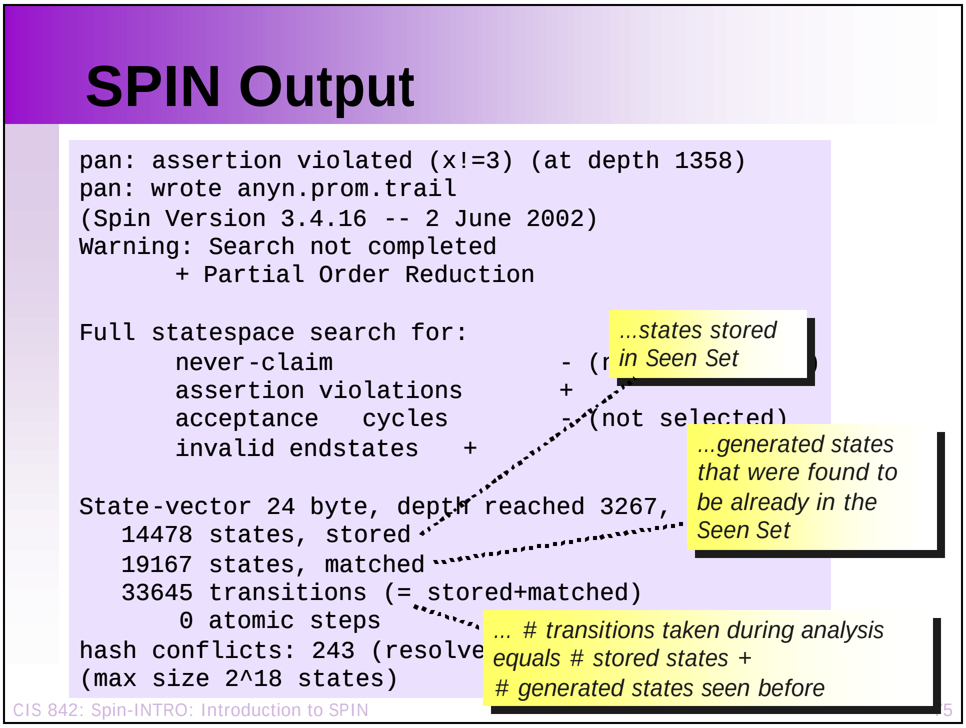
```
bool x;

proctype Thread1()
{ do
  :: x = !x;
  od }

proctype Thread2()
{ do
  :: x = !x;
  od }

init {
  x = false;
  run Thread1();
  run Thread2(); }
```

- Consider this example system…
  - How many states does it have?
  - Does execution of the system terminate?
  - Does an exhaustive analysis of the state-space of the system terminate?



For You To Do…

- Pause the lecture…
- Download the file finitenonterminating.prom from the examples page.
- Run SPIN in simulation mode on the example.
  - What do you observe?
- Run SPIN in analysis mode.
  - What do you observe?
  - Use the output of SPIN to answer the following questions…
    - How many states does the system have?
    - How many states were stored in the Seen Set?
    - How many states does the program generate before it comes back to a previous state?



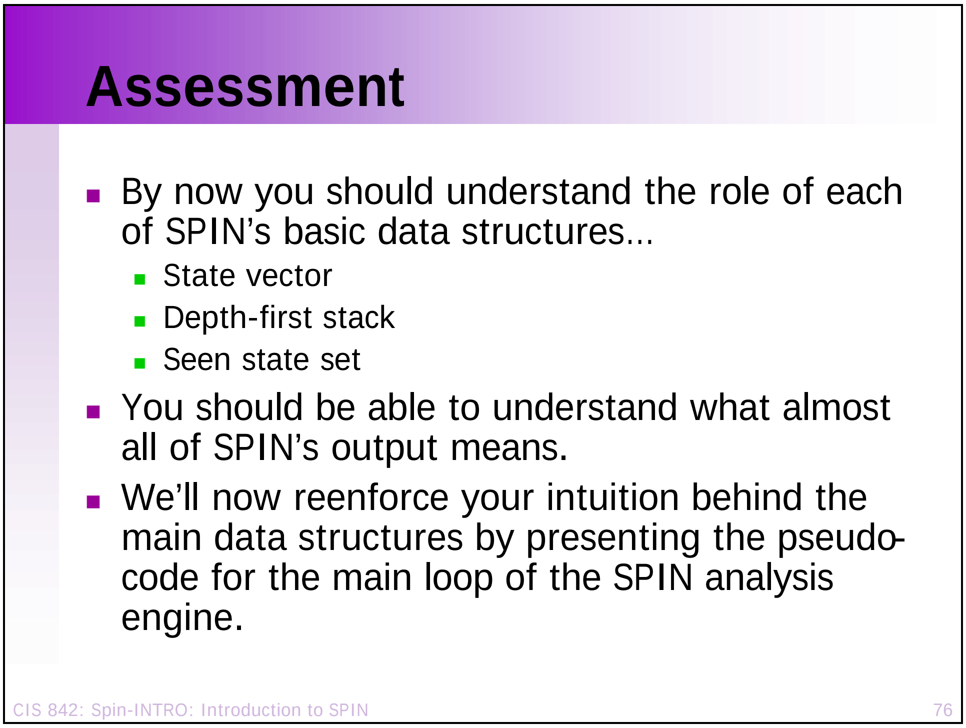
SPIN Output

```
pan: assertion violated (x!=3) (at depth 1358)
pan: wrote anyn.prom.trail
(Spin Version 3.4.16 -- 2 June 2002)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never-claim                 - (r
        assertion violations        +
        acceptance   cycles         - (not selected)
        invalid endstates   +

State-vector 24 byte, depth reached 3267,
   14478 states, stored
   19167 states, matched
   33645 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 243 (resolve
(max size 2^18 states)
```

...states stored in Seen Set

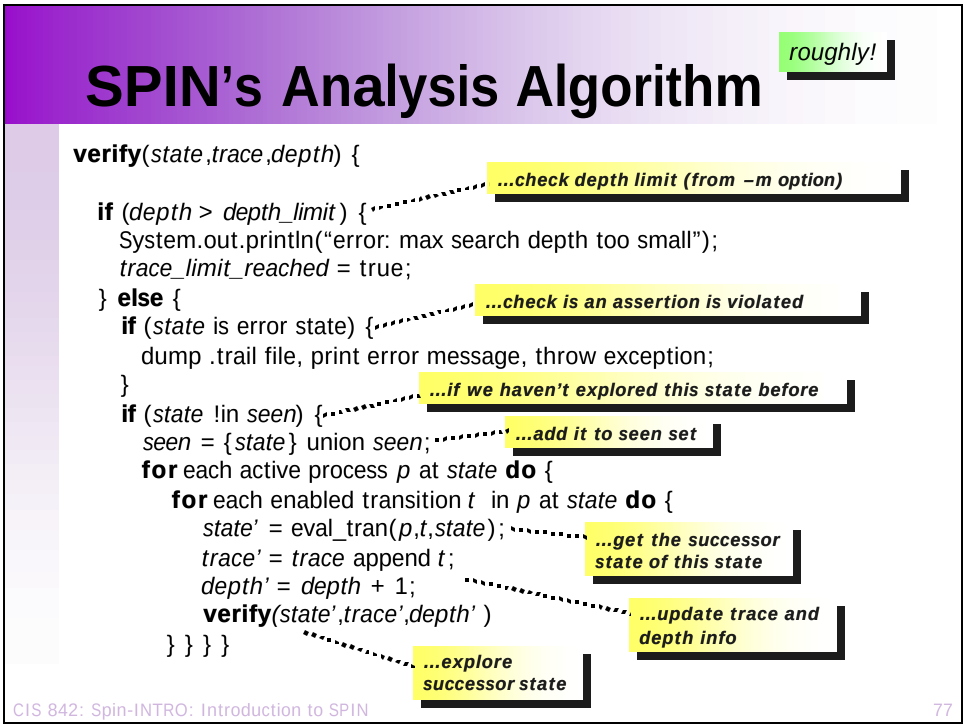
...generated states that were found to be already in the Seen Set

... # transitions taken during analysis equals # stored states + # generated states seen before



Assessment

- By now you should understand the role of each of SPIN’s basic data structures…
  - State vector
  - Depth-first stack
  - Seen state set
- You should be able to understand what almost all of SPIN’s output means.
- We’ll now reenforce your intuition behind the main data structures by presenting the pseudo-code for the main loop of the SPIN analysis engine.



SPIN's Analysis Algorithm

roughly!

```
verify(state,trace,depth) {

  if (depth > depth_limit) {
    System.out.println("error: max search depth too small");
    trace_limit_reached = true;
  } else {
    if (state is error state) {
      dump .trail file, print error message, throw exception;
    }
    if (state !in seen) {
      seen = {state} union seen;
      for each active process p at state do {
        for each enabled transition t  in p at state do {
          state' = eval_tran(p,t,state);
          trace' = trace append t;
          depth' = depth + 1;
          verify(state',trace',depth' )
    } } } }
```

...check depth limit (from –m option)

...check is an assertion is violated

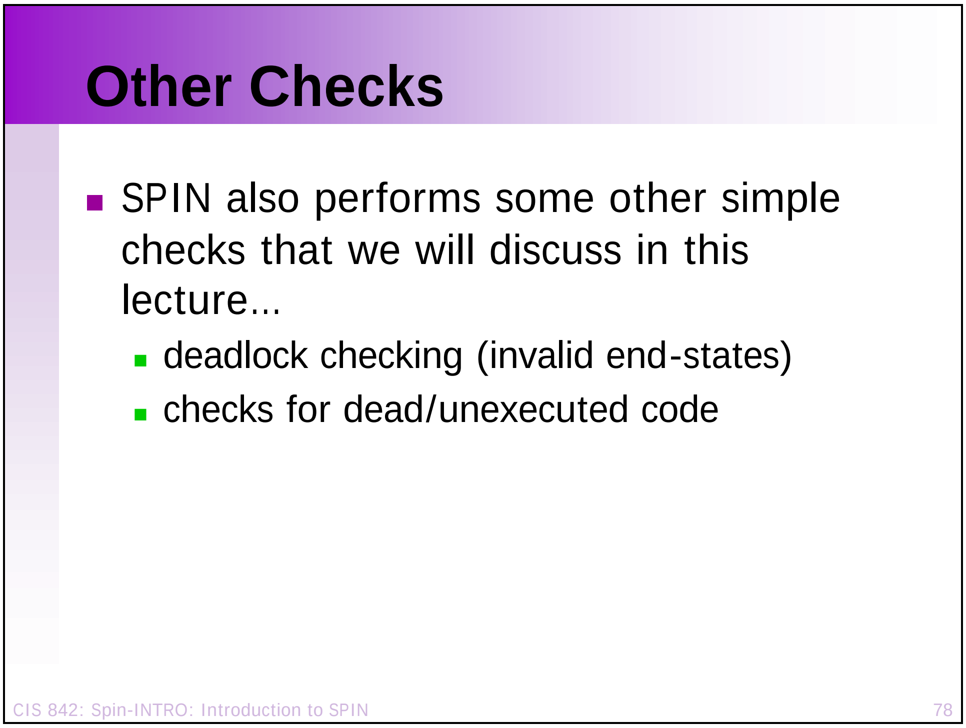
...if we haven't explored this state before

...add it to seen set

...get the successor state of this state

...update trace and depth info

...explore successor state



Other Checks

- SPIN also performs some other simple checks that we will discuss in this lecture...
  - deadlock checking (invalid end-states)
  - checks for dead/unexecuted code

Invalid End States

```
byte x,y;

proctype Thread1()
{
 if
 :: (x == 0)
     -> {x++;
         if :: (y == 0)
                 -> y++; y--
         fi}
         x--;
 fi
}
```

```
byte x,y;

proctype Thread1()
{
 if
 :: (y == 0)
     -> {y++;
         if :: (x == 0)
                 -> x++; x--
         fi}
         y--;
 fi
}
```

```
init {
  run Thread1();
  run Thread2();
}
```

Is there a schedule where this program reaches a deadlock?

For You To Do…

- Pause the lecture…
- Download the file simple-deadlock.prom (the example from the previous slide) from the examples page.
- Use SPIN to analyze the program simple-deadlock.prom and find an error trace of minimal length.
- What can you infer from SPIN’s output?
- Use the error-trace guided simulation mode to view the error trace execution.
- What can you infer from the error trace information?

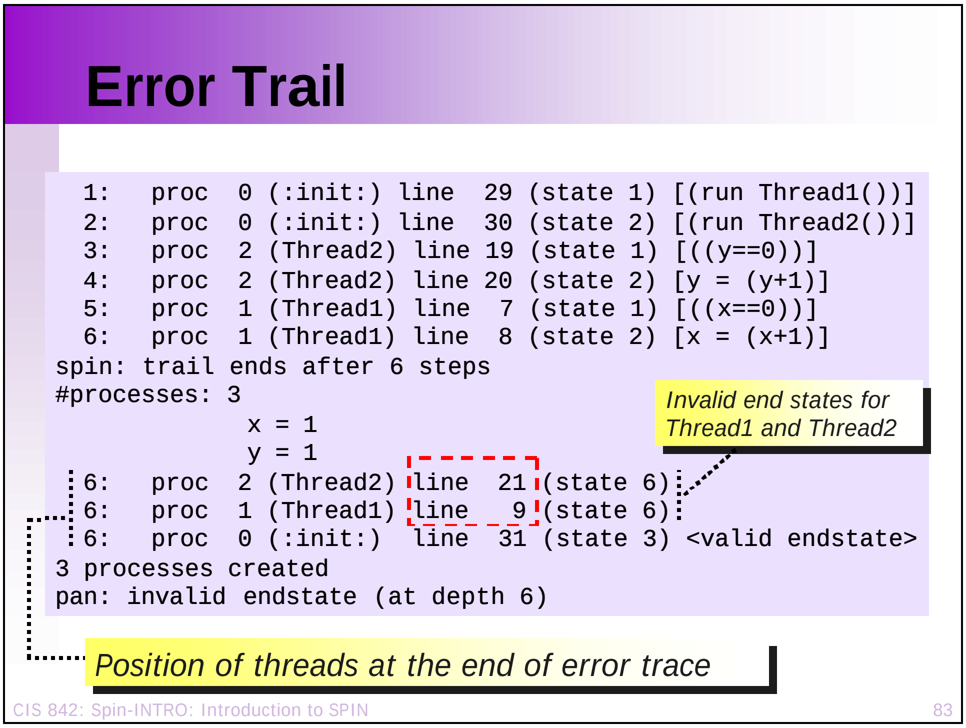
SPIN Output

```
pan: invalid endstate (at depth 6)
pan: wrote simple-deadlock.prom.trail
pan: reducing search depth to 5
(Spin Version 3.4.16 -- 2 June 2002)
       + Partial Order Reduction

Full statespace search for:
       never-claim              - (none specified)
       assertion violations     +
       cycle checks      - (disabled by -DSAFETY)
       invalid endstates   +

State-vector 20 byte, depth reached 17, errors: 1
      31 states, stored
      11 states, matched
      42 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
```

Indicates that there is a deadlock.

Assessment

- "Invalid end state" means that SPIN found an execution where the program cannot take another step, and at the point where it has stopped at least one of the processes is at a point that is not the end of its execution.
- Usually, this means that the discovered execution is a deadlock error (at least two threads are stuck).
- However, we will see later on in the course that you may specifically design a program where one or more threads are designed to run forever. In this case, we can attach special labels to statements to tell Spin that some "invalid end states" are actually OK.

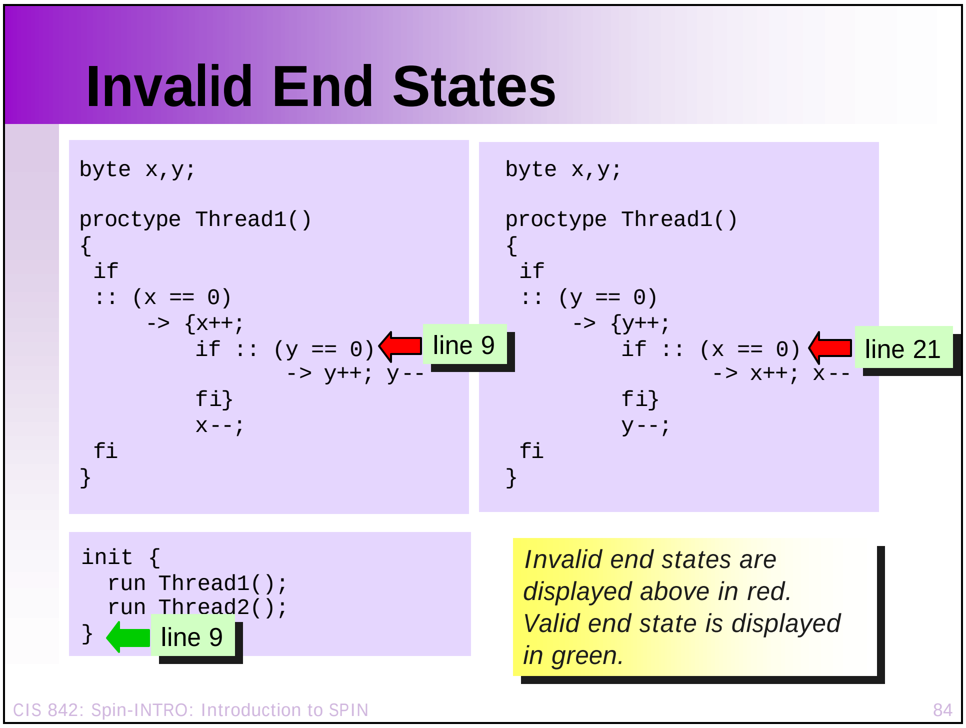


Error Trail

```
  1:   proc  0 (:init:) line  29 (state 1) [(run Thread1())]
  2:   proc  0 (:init:) line  30 (state 2) [(run Thread2())]
  3:   proc  2 (Thread2) line 19 (state 1) [((y==0))]
  4:   proc  2 (Thread2) line 20 (state 2) [y = (y+1)]
  5:   proc  1 (Thread1) line  7 (state 1) [((x==0))]
  6:   proc  1 (Thread1) line  8 (state 2) [x = (x+1)]
spin: trail ends after 6 steps
#processes: 3
             x = 1
             y = 1
  6:   proc  2 (Thread2) line   21 (state 6)
  6:   proc  1 (Thread1) line    9 (state 6)
  6:   proc  0 (:init:)  line   31 (state 3) <valid endstate>
3 processes created
pan: invalid endstate (at depth 6)
```

Invalid end states for Thread1 and Thread2

Position of threads at the end of error trace



Invalid End States

```
byte x,y;

proctype Thread1()
{
 if
 :: (x == 0)
     -> {x++;
         if :: (y == 0)
                -> y++; y--
         fi}
         x--;
 fi
}
```

line 9

```
byte x,y;

proctype Thread1()
{
 if
 :: (y == 0)
     -> {y++;
         if :: (x == 0)
                -> x++; x--
         fi}
         y--;
 fi
}
```

line 21

```
init {
  run Thread1();
  run Thread2();
}
```

line 9

Invalid end states are displayed above in red. Valid end state is displayed in green.

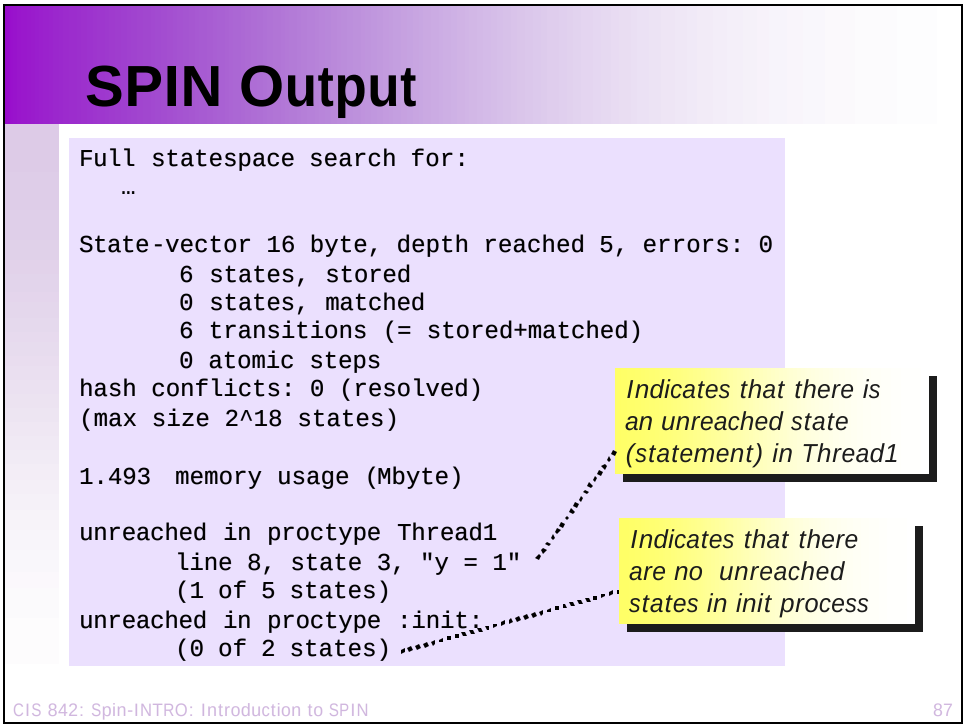
Dead/Unexecuted Code

```
byte x,y;

proctype Thread1()
{
    x = 0;
    goto l1;
    y = 1;
l1: y = 0;
}

init {
  run Thread1();
}
```

Is there any dead code in this program (statements that can never be executed)?

For You To Do…

- Pause the lecture…
- Download the file deadcode.prom from the examples web page.
- Use SPIN to analyze the program.
- What can you infer from the output?

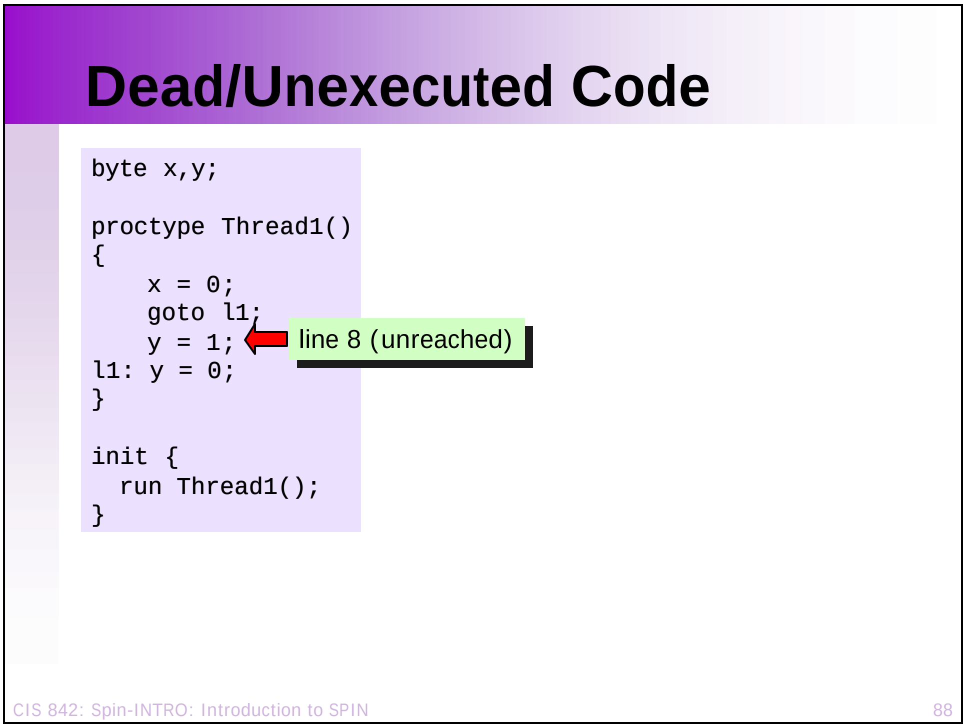


SPIN Output

```
Full statespace search for:
   …

State-vector 16 byte, depth reached 5, errors: 0
       6 states, stored
       0 states, matched
       6 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493  memory usage (Mbyte)

unreached in proctype Thread1
       line 8, state 3, "y = 1"
       (1 of 5 states)
unreached in proctype :init:
       (0 of 2 states)
```

Indicates that there is an unreached state (statement) in Thread1

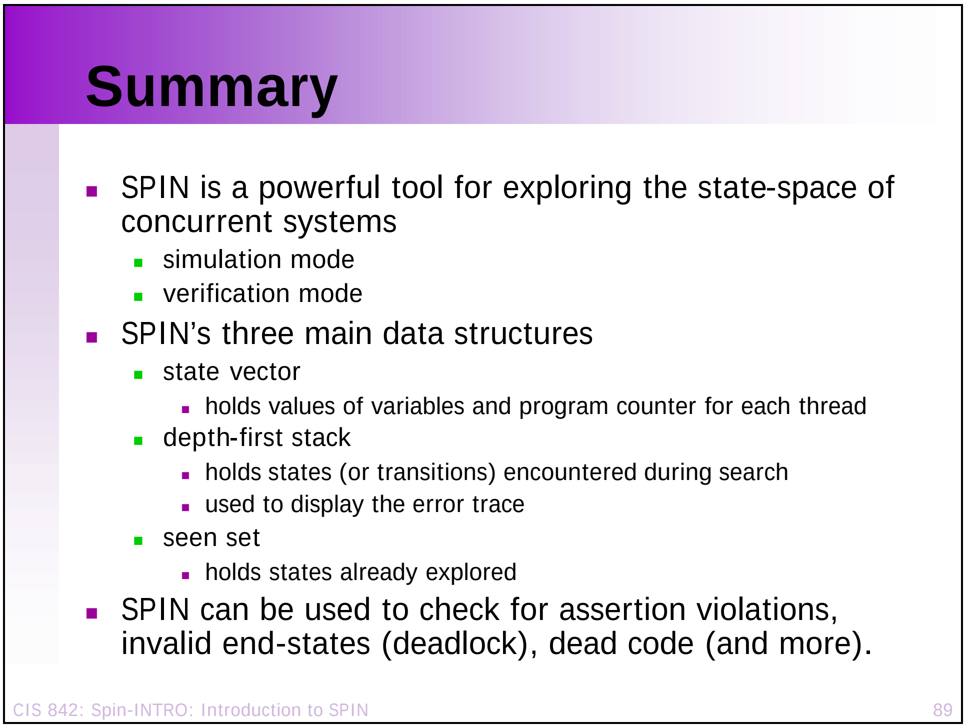
Indicates that there are no unreached states in init process



Dead/Unexecuted Code

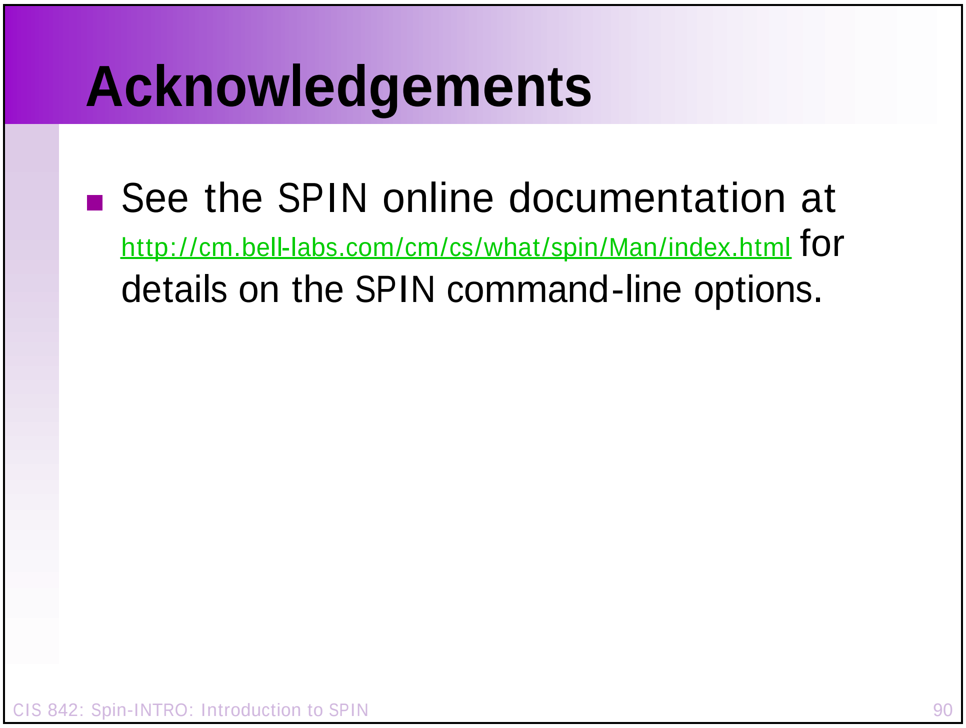
```
byte x,y;

proctype Thread1()
{
    x = 0;
    goto l1;
    y = 1;
l1: y = 0;
}

init {
  run Thread1();
}
```

line 8 (unreached)



Summary

- SPIN is a powerful tool for exploring the state-space of concurrent systems
  - simulation mode
  - verification mode
- SPIN's three main data structures
  - state vector
    - holds values of variables and program counter for each thread
  - depth-first stack
    - holds states (or transitions) encountered during search
    - used to display the error trace
  - seen set
    - holds states already explored
- SPIN can be used to check for assertion violations, invalid end-states (deadlock), dead code (and more).



Acknowledgements

- See the SPIN online documentation at http://cm.bell-labs.com/cm/cs/what/spin/Man/index.html for details on the SPIN command-line options.