MACHINE-ASSISTED CONCURRENT PROGRAMMING

Martin Vechev

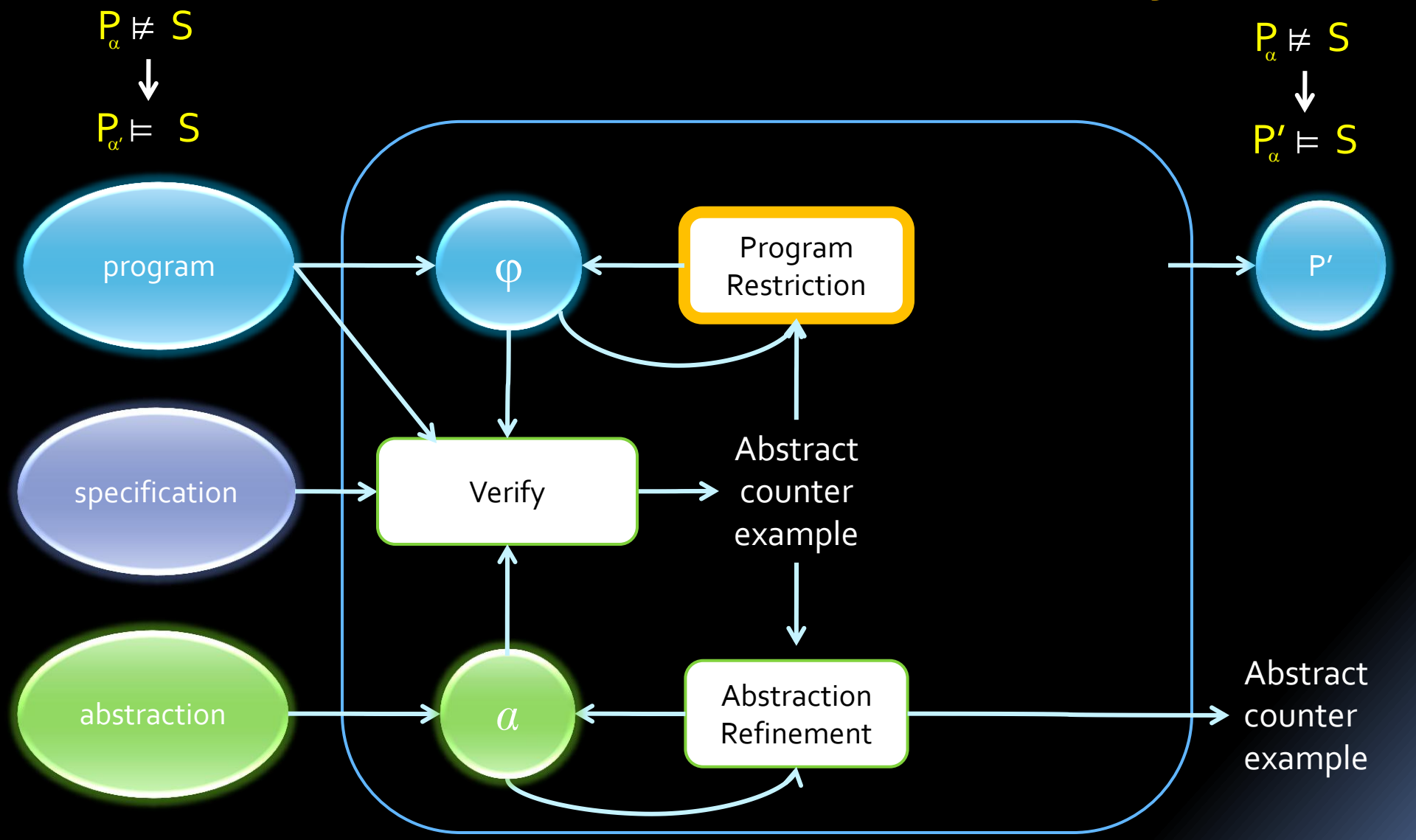
ETH Zürich

(Lecture 2)

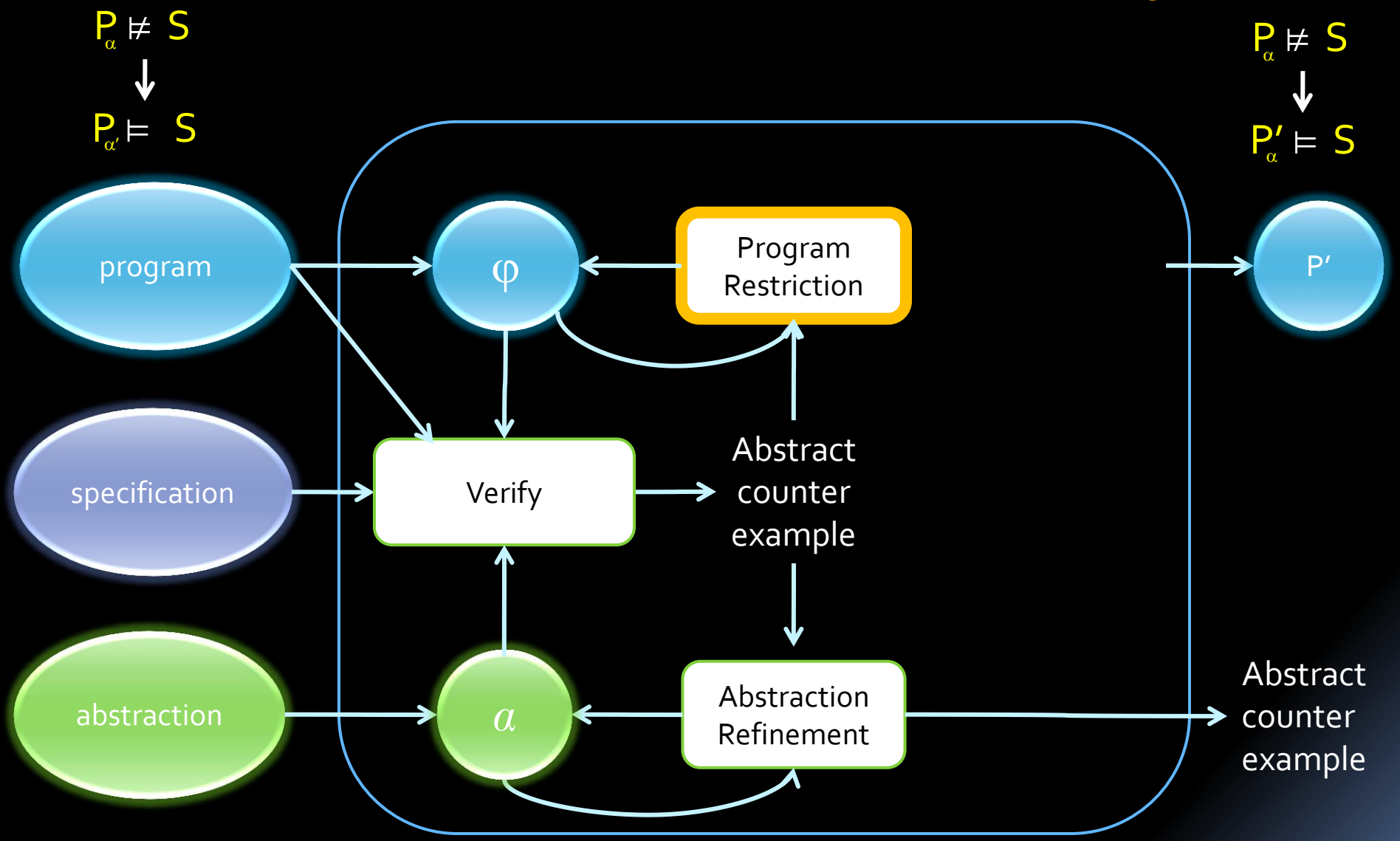
Plan

- Setting
- Program analysis by abstract interpretation
- Synthesis based on abstract interpretation
- Analysis + synthesis for weak memory models

Instantiate for Concurrency

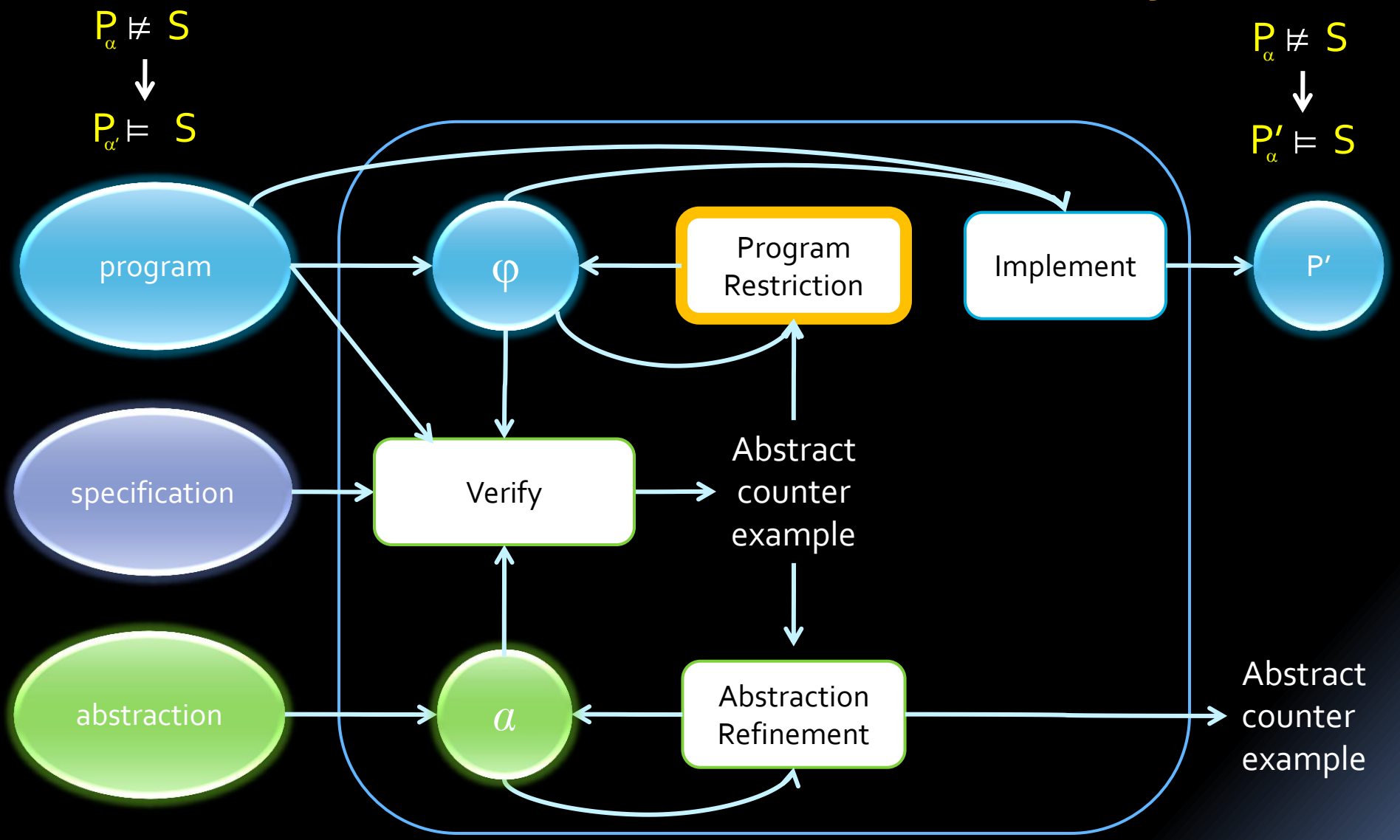


Instantiate for Concurrency



Change the **program** to match the **abstraction**

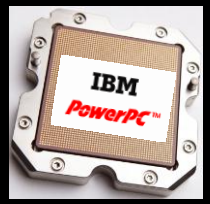
Instantiate for Concurrency



Change the **program** to match the **abstraction**

Another Interesting Area

- Relaxed Memory Models
  - Provided by modern chips to improve performance



Textbook Example: Dekker's Algorithm

```
p0:
flag[0] := true
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

Specification: mutual exclusion over critical section

Textbook Example: Dekker's Algorithm

```
p0:
flag[0] := true
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

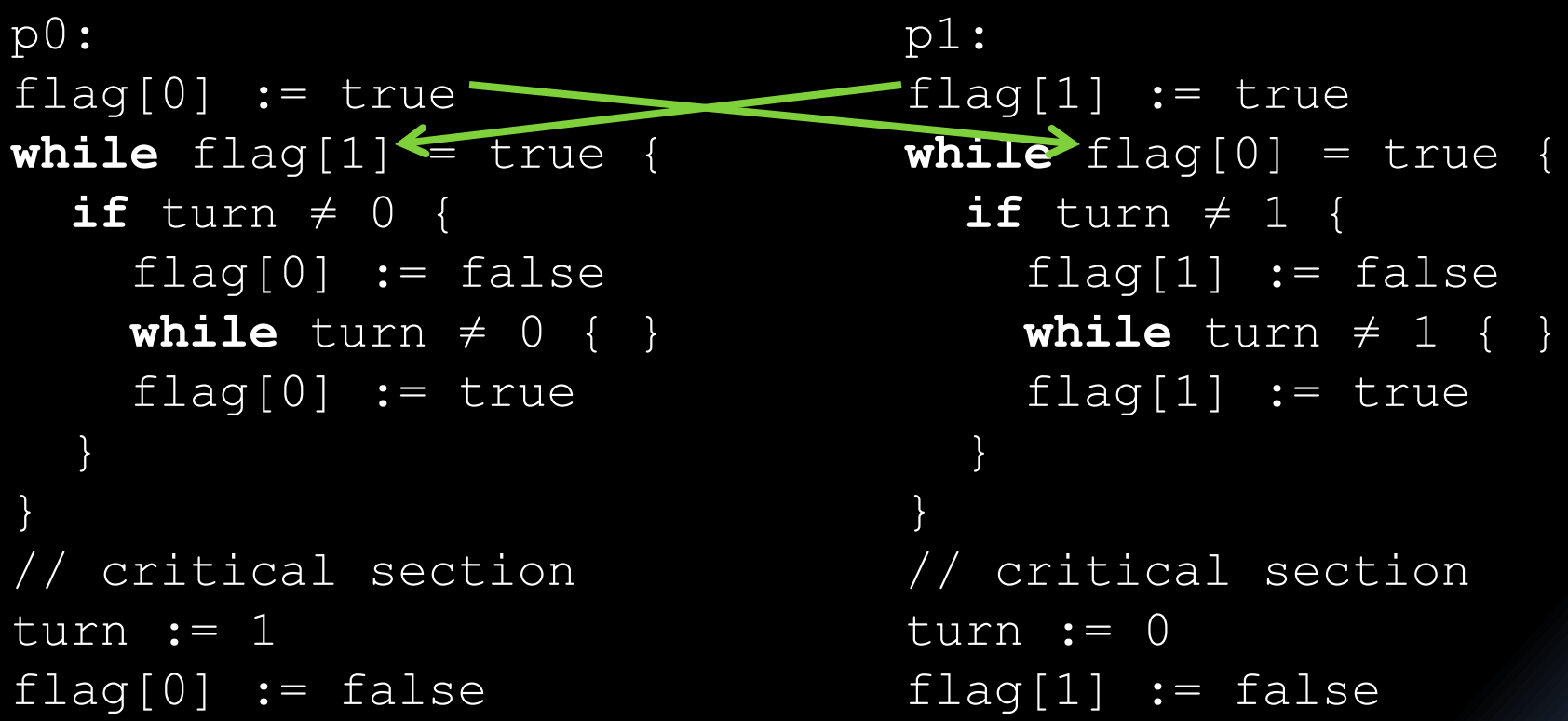
Specification: mutual exclusion over critical section

Textbook Example: Dekker's Algorithm

```
p0:                                  p1:
flag[0] := true                      flag[1] := true
while flag[1] = true {               while flag[0] = true {
  if turn ≠ 0 {                        if turn ≠ 1 {
    flag[0] := false                     flag[1] := false
    while turn ≠ 0 { }                   while turn ≠ 1 { }
    flag[0] := true                      flag[1] := true
  }                                    }
}                                    }
// critical section                  // critical section
turn := 1                            turn := 0
flag[0] := false                     flag[1] := false
```

Specification: mutual exclusion over critical section

Beyond Textbooks: Weak Memory Models



```
p0:
flag[0] := true
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

- Re-ordering of operations
- Non-atomic stores

Beyond Textbooks: Weak Memory Models

```
p0:
flag[0] := true
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

- Re-ordering of operations
- Non-atomic stores

Beyond Textbooks: Weak Memory Models

```
p0:
flag[0] := true
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

- Re-ordering of operations
- Non-atomic stores

Beyond Textbooks: Weak Memory Models

```
p0:
flag[0] := true
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

- Re-ordering of operations
- Non-atomic stores

Beyond Textbooks: Weak Memory Models

```
p0:
flag[0] := true
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

- Re-ordering of operations
- Non-atomic stores

Fences

- Enforce order… at a cost
- Fences are expensive
  - 10s-100s of cycles
  - collateral damage (e.g., prevent compiler opts)
- example: removing a single fence yields 3x speedup in a work-stealing queue
- Required fences depend on memory model

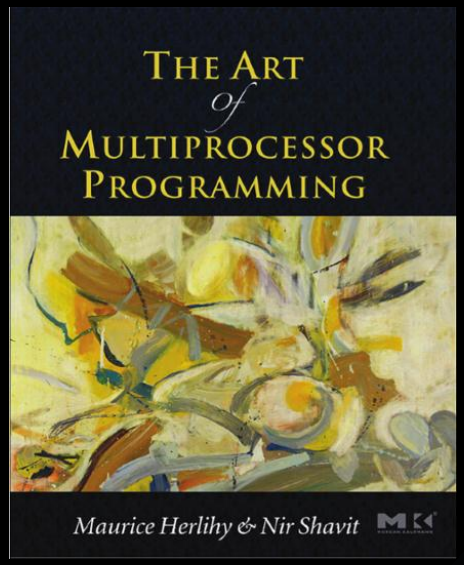
- Where should I put fences?

Where should I put fences?

On the one hand, memory barriers are expensive (100s of cycles, maybe more), and should be used only when necessary.

On the other, synchronization bugs can be very difficult to track down, so memory barriers should be used liberally, rather than relying on complex platform-specific guarantees about limits to memory instruction reordering.

– Herlihy and Shavit



Easy!

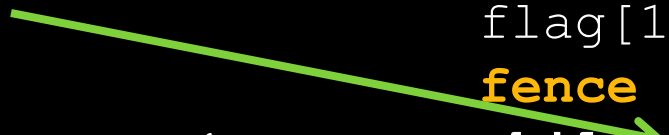
```
p0:
flag[0] := true
fence
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
fence
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

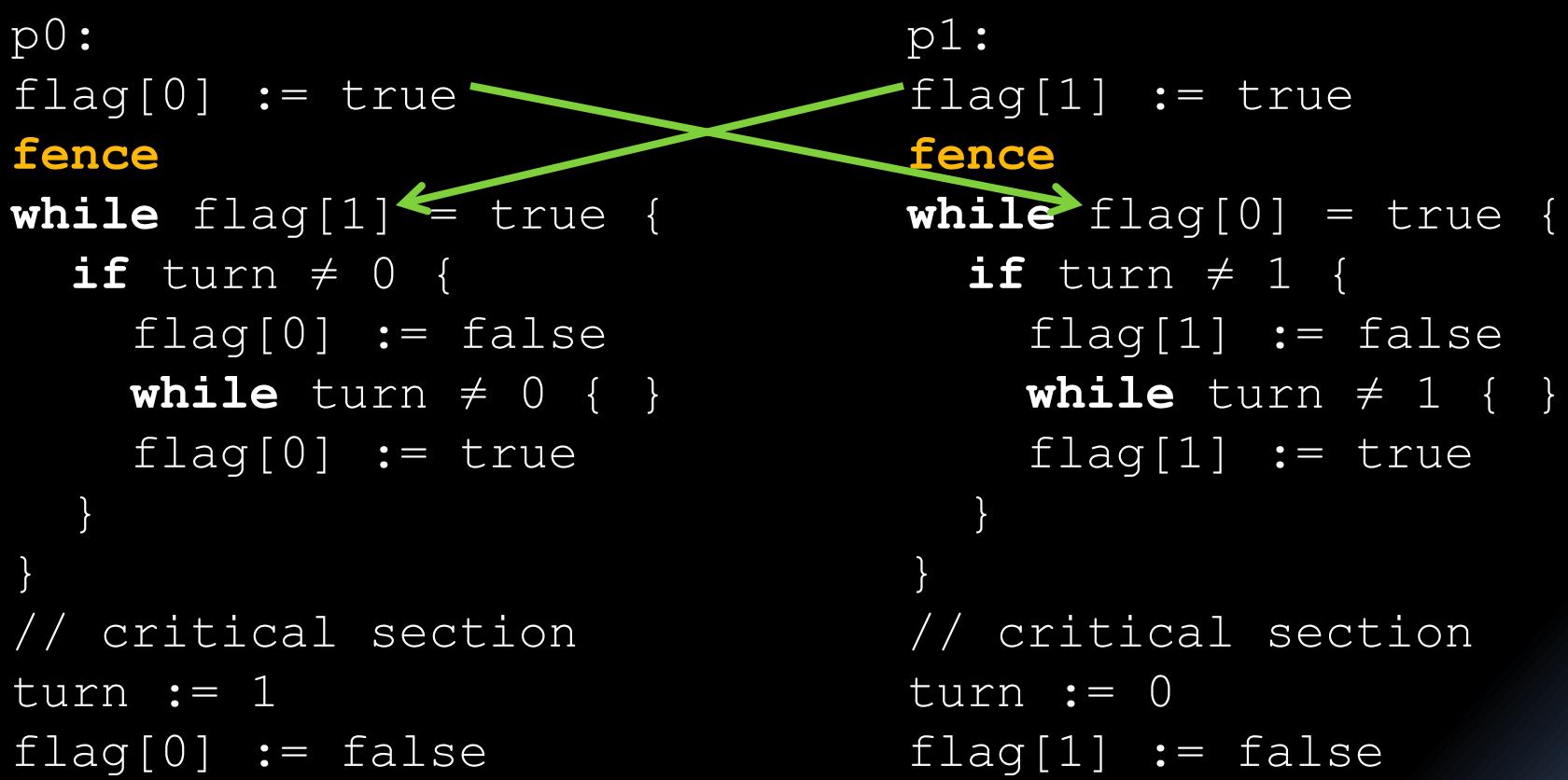
Easy!

```
p0:
flag[0] := true
fence
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
fence
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```



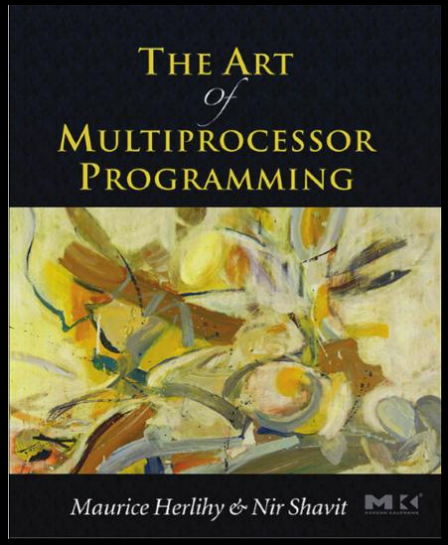
Easy!



```
p0:
flag[0] := true
fence
while flag[1] = true {
  if turn ≠ 0 {
    flag[0] := false
    while turn ≠ 0 { }
    flag[0] := true
  }
}
// critical section
turn := 1
flag[0] := false
```

```
p1:
flag[1] := true
fence
while flag[0] = true {
  if turn ≠ 1 {
    flag[1] := false
    while turn ≠ 1 { }
    flag[1] := true
  }
}
// critical section
turn := 0
flag[1] := false
```

Chase-Lev Work-Stealing Queue

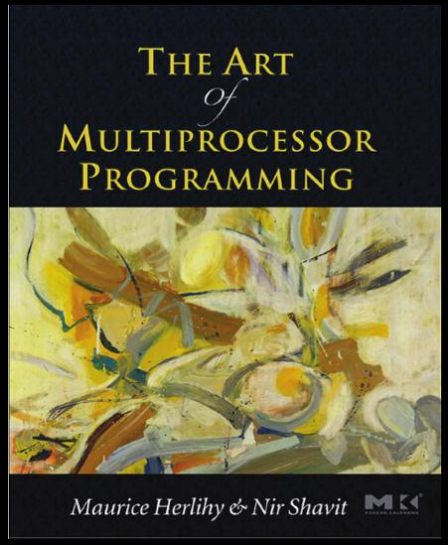



```
1  int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b

5      long t = top
6      if (b < t) {
7          bottom = t;
8          return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12         return task
13     if (!CAS(&top, t, t+1))
14         return EMPTY;
15     bottom = t + 1;
16     return task;
17   }
```

```
1  void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6          wsq = expand();
7          q = wsq;
8      }
9      q->ap[b % q->size] = task;

10     bottom = b + 1;
11   }
```

```
1  int steal() {
2      long t = top;

3      long b = bottom;

4      item_t * q = wsq;
5      if (t >= b)
6          return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9          return ABORT;
10     return task;
11   }
```

Chase-Lev Work-Stealing Queue

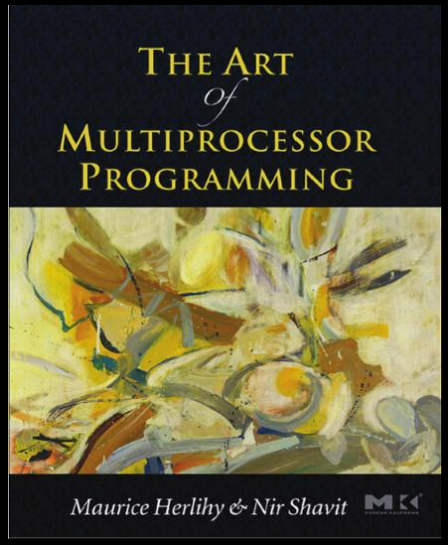



```
1  int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b

5      long t = top
6      if (b < t) {
7          bottom = t;
8          return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12         return task
13     if (!CAS(&top, t, t+1))
14         return EMPTY;
15     bottom = t + 1;
16     return task;
17   }
```

```
1  void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6          wsq = expand();
7          q = wsq;
8      }
9      q->ap[b % q->size] = task;

10     bottom = b + 1;
11   }
```

```
1  int steal() {
2      long t = top;

3      long b = bottom;

4      item_t * q = wsq;
5      if (t >= b)
6          return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9          return ABORT;
10     return task;
11   }
```

Chase-Lev Work-Stealing Queue

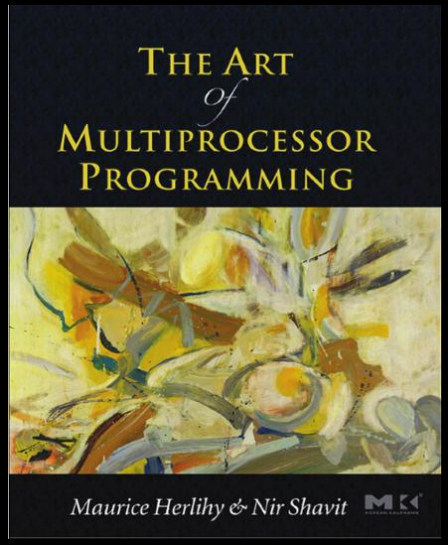



```
1   int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b
       fence
5      long t = top
6      if (b < t) {
7         bottom = t;
8         return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12        return task
13     if (!CAS(&top, t, t+1))
14        return EMPTY;
15     bottom = t + 1;
16     return task;
17   }
```

```
1   void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6         wsq = expand();
7         q = wsq;
8      }
9      q->ap[b % q->size] = task;

10     bottom = b + 1;
11  }
```

```
1   int steal() {
2      long t = top;

3      long b = bottom;

4      item_t * q = wsq;
5      if (t >= b)
6         return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9         return ABORT;
10     return task;
11  }
```

Chase-Lev Work-Stealing Queue

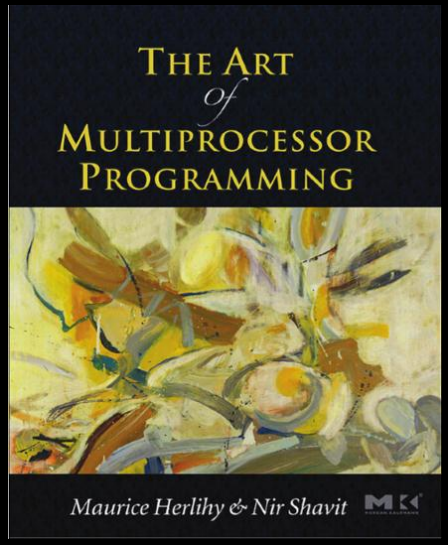



```
1   int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b
       fence
5      long t = top
6      if (b < t) {
7         bottom = t;
8         return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12        return task
13     if (!CAS(&top, t, t+1))
14        return EMPTY;
15     bottom = t + 1;
16     return task;
17   }
```

```
1   void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6          wsq = expand();
7          q = wsq;
8      }
9      q->ap[b % q->size] = task;

10     bottom = b + 1;
11  }
```

```
1   int steal() {
2      long t = top;

3      long b = bottom;

4      item_t * q = wsq;
5      if (t >= b)
6          return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9          return ABORT;
10     return task;
11  }
```

Chase-Lev Work-Stealing Queue

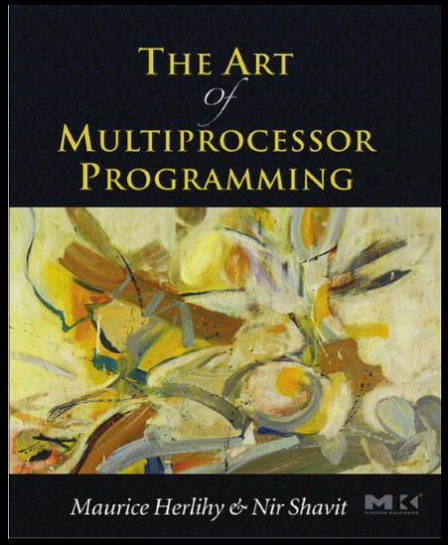



```
1   int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b
       fence
5      long t = top
6      if (b < t) {
7         bottom = t;
8         return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12        return task
13     if (!CAS(&top, t, t+1))
14        return EMPTY;
15     bottom = t + 1;
16     return task;
17   }
```


```
1   void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6         wsq = expand();
7         q = wsq;
8      }
9      q->ap[b % q->size] = task;

10     bottom = b + 1;
11  }
```

```
1   int steal() {
2      long t = top;

3      long b = bottom;

4      item_t * q = wsq;
5      if (t >= b)
6         return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9         return ABORT;
10     return task;
11  }
```

Chase-Lev Work-Stealing Queue

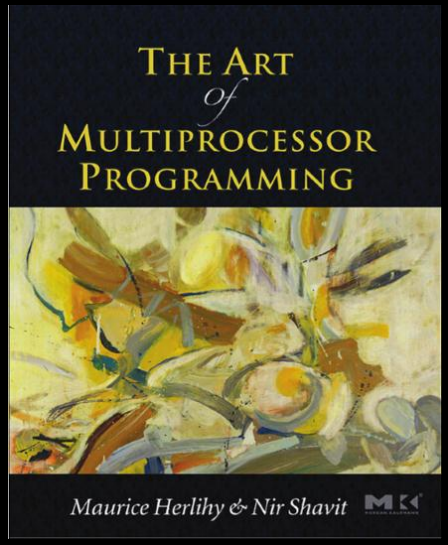



```
1   int take() {
2       long b = bottom - 1;
3       item_t * q = wsq;
4       bottom = b
        fence
5       long t = top
6       if (b < t) {
7          bottom = t;
8          return EMPTY;
9       }
10      task = q->ap[b % q->size];
11      if (b > t)
12         return task
13      if (!CAS(&top, t, t+1))
14         return EMPTY;
15      bottom = t + 1;
16      return task;
17    }
```


```
1   void push(int task) {
2       long b = bottom;
3       long t = top;
4       item_t * q = wsq;
5       if (b - t ≥ q->size - 1) {
6           wsq = expand();
7           q = wsq;
8       }
9       q->ap[b % q->size] = task;
        fence
10      bottom = b + 1;
11    }
```

```
1   int steal() {
2       long t = top;

3       long b = bottom;

4       item_t * q = wsq;
5       if (t >= b)
6           return EMPTY;
7       task = q->ap[t % q->size];

8       if (!CAS(&top, t, t+1))
9           return ABORT;
10      return task;
11    }
```

Chase-Lev Work-Stealing Queue

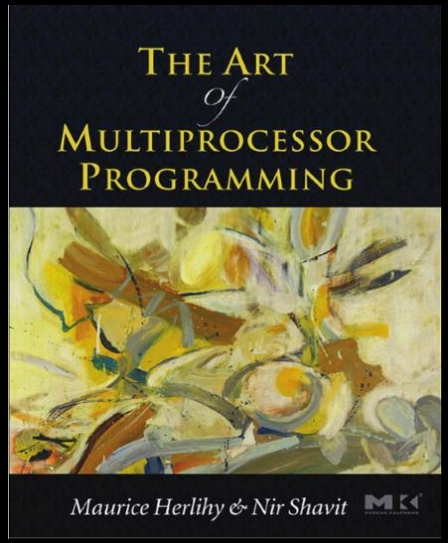



```
1   int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b
       fence
5      long t = top
6      if (b < t) {
7         bottom = t;
8         return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12        return task
13     if (!CAS(&top, t, t+1))
14        return EMPTY;
15     bottom = t + 1;
16     return task;
17   }
```

```
1   void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6         wsq = expand();
7         q = wsq;
8      }
9      q->ap[b % q->size] = task;
       fence
10     bottom = b + 1;
11  }
```

```
1   int steal() {
2      long t = top;

3      long b = bottom;

4      item_t * q = wsq;
5      if (t >= b)
6         return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9         return ABORT;
10     return task;
11  }
```

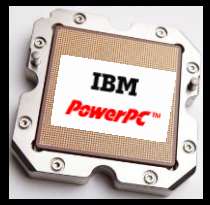
Chase-Lev Work-Stealing Queue



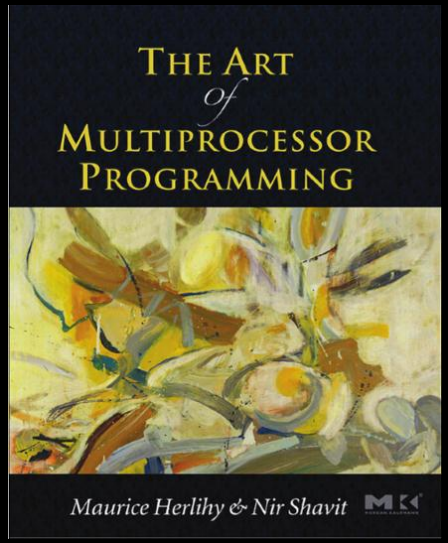

```
1   int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b
       fence
5      long t = top
6      if (b < t) {
7         bottom = t;
8         return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12        return task
13     if (!CAS(&top, t, t+1))
14        return EMPTY;
15     bottom = t + 1;
16     return task;
17  }
```

```
1   void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6         wsq = expand();
7         q = wsq;
8      }
9      q->ap[b % q->size] = task;
       fence
10     bottom = b + 1;
11  }
```

```
1   int steal() {
2      long t = top;

3      long b = bottom;

4      item_t * q = wsq;
5      if (t >= b)
6         return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9         return ABORT;
10     return task;
11  }
```



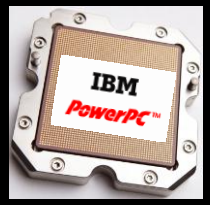
Chase-Lev Work-Stealing Queue



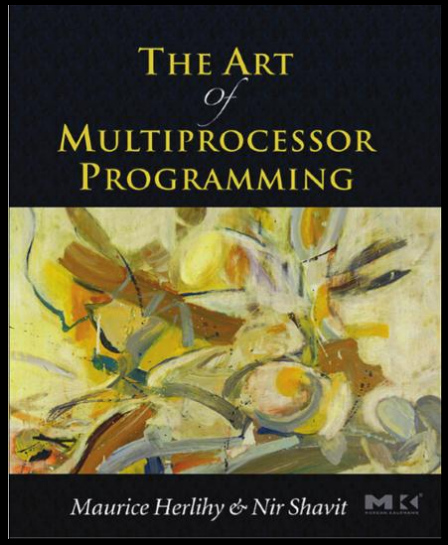

```
1   int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b
       fence
5      long t = top
6      if (b < t) {
7         bottom = t;
8         return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12        return task
13     if (!CAS(&top, t, t+1))
14        return EMPTY;
15     bottom = t + 1;
16     return task;
17  }
```

```
1   void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6         wsq = expand();
7         q = wsq;
8      }
9      q->ap[b % q->size] = task;
       fence
10     bottom = b + 1;
11  }
```

```
1   int steal() {
2      long t = top;
       fence
3      long b = bottom;

4      item_t * q = wsq;
5      if (t >= b)
6         return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9         return ABORT;
10     return task;
11  }
```



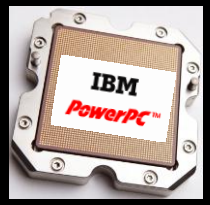
Chase-Lev Work-Stealing Queue



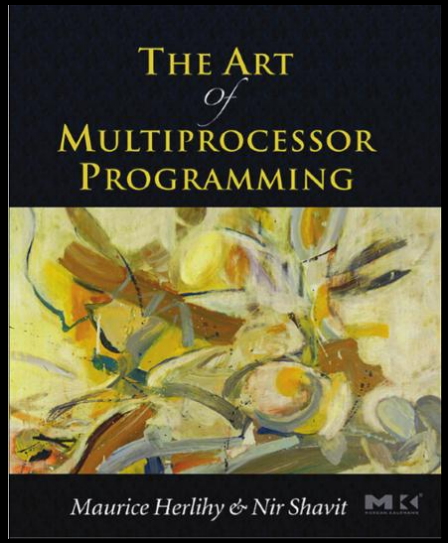

```
1   int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b
       fence
5      long t = top
6      if (b < t) {
7         bottom = t;
8         return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12        return task
13     if (!CAS(&top, t, t+1))
14        return EMPTY;
15     bottom = t + 1;
16     return task;
17  }
```

```
1   void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6         wsq = expand();
7         q = wsq;
8      }
9      q->ap[b % q->size] = task;
       fence
10     bottom = b + 1;
11  }
```

```
1   int steal() {
2      long t = top;
       fence
3      long b = bottom;
       fence
4      item_t * q = wsq;
5      if (t >= b)
6         return EMPTY;
7      task = q->ap[t % q->size];

8      if (!CAS(&top, t, t+1))
9         return ABORT;
10     return task;
11  }
```



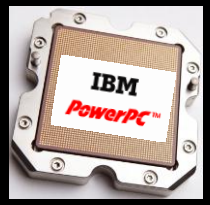
Chase-Lev Work-Stealing Queue




```
1  int take() {
2      long b = bottom - 1;
3      item_t * q = wsq;
4      bottom = b
       fence
5      long t = top
6      if (b < t) {
7          bottom = t;
8          return EMPTY;
9      }
10     task = q->ap[b % q->size];
11     if (b > t)
12         return task
13     if (!CAS(&top, t, t+1))
14         return EMPTY;
15     bottom = t + 1;
16     return task;
17   }
```

```
1  void push(int task) {
2      long b = bottom;
3      long t = top;
4      item_t * q = wsq;
5      if (b - t ≥ q->size - 1) {
6          wsq = expand();
7          q = wsq;
8      }
9      q->ap[b % q->size] = task;
       fence
10     bottom = b + 1;
11   }
```

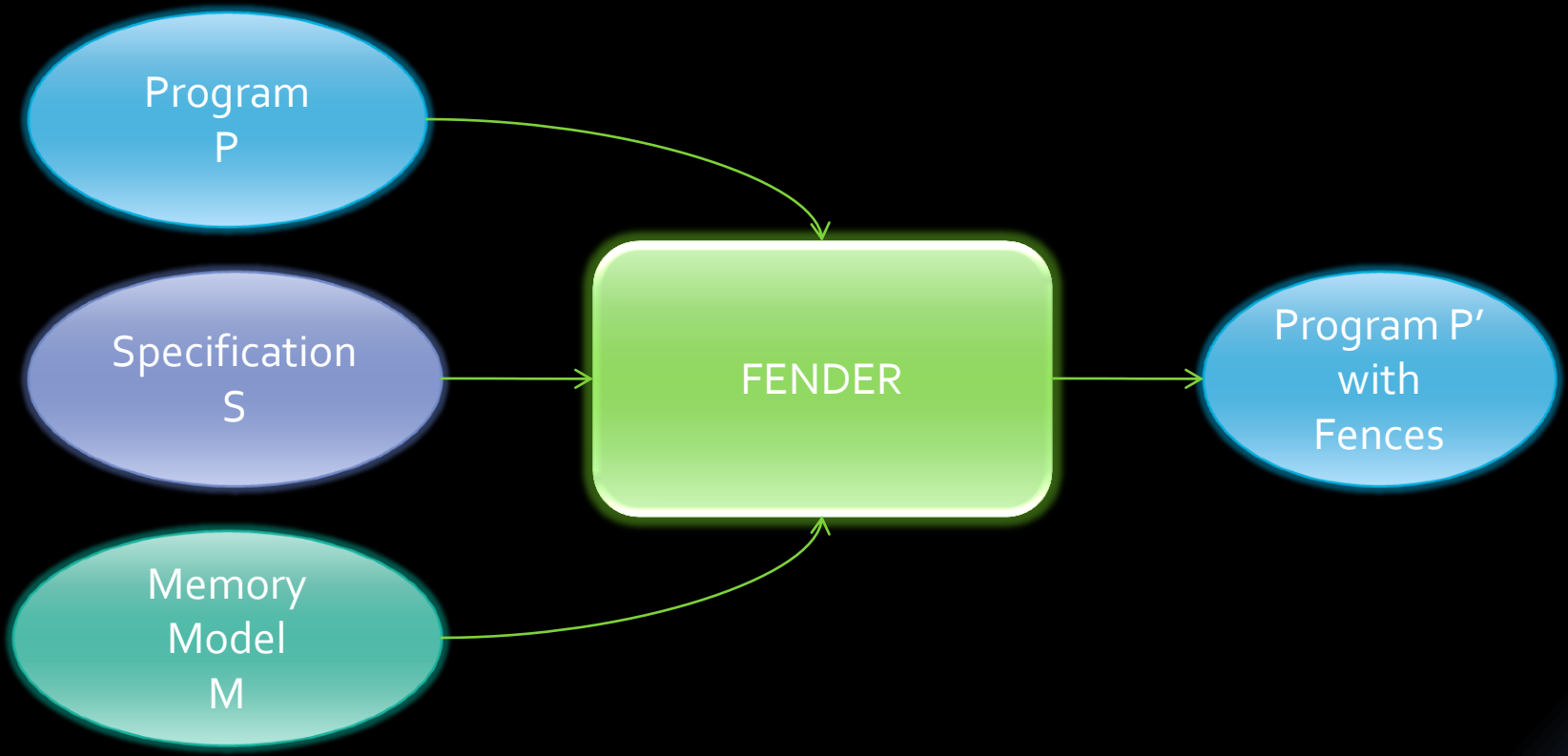
```
1  int steal() {
2      long t = top;
       fence
3      long b = bottom;
       fence
4      item_t * q = wsq;
5      if (t >= b)
6          return EMPTY;
7      task = q->ap[t % q->size];
       fence
8      if (!CAS(&top, t, t+1))
9          return ABORT;
10     return task;
11   }
```



Goal

- Help the programmer place fences in the program
  - Find optimal fence placement
- Principle
  - Restrict non-determinism s.t. program stays within set of safe executions

Our Approach: Overview

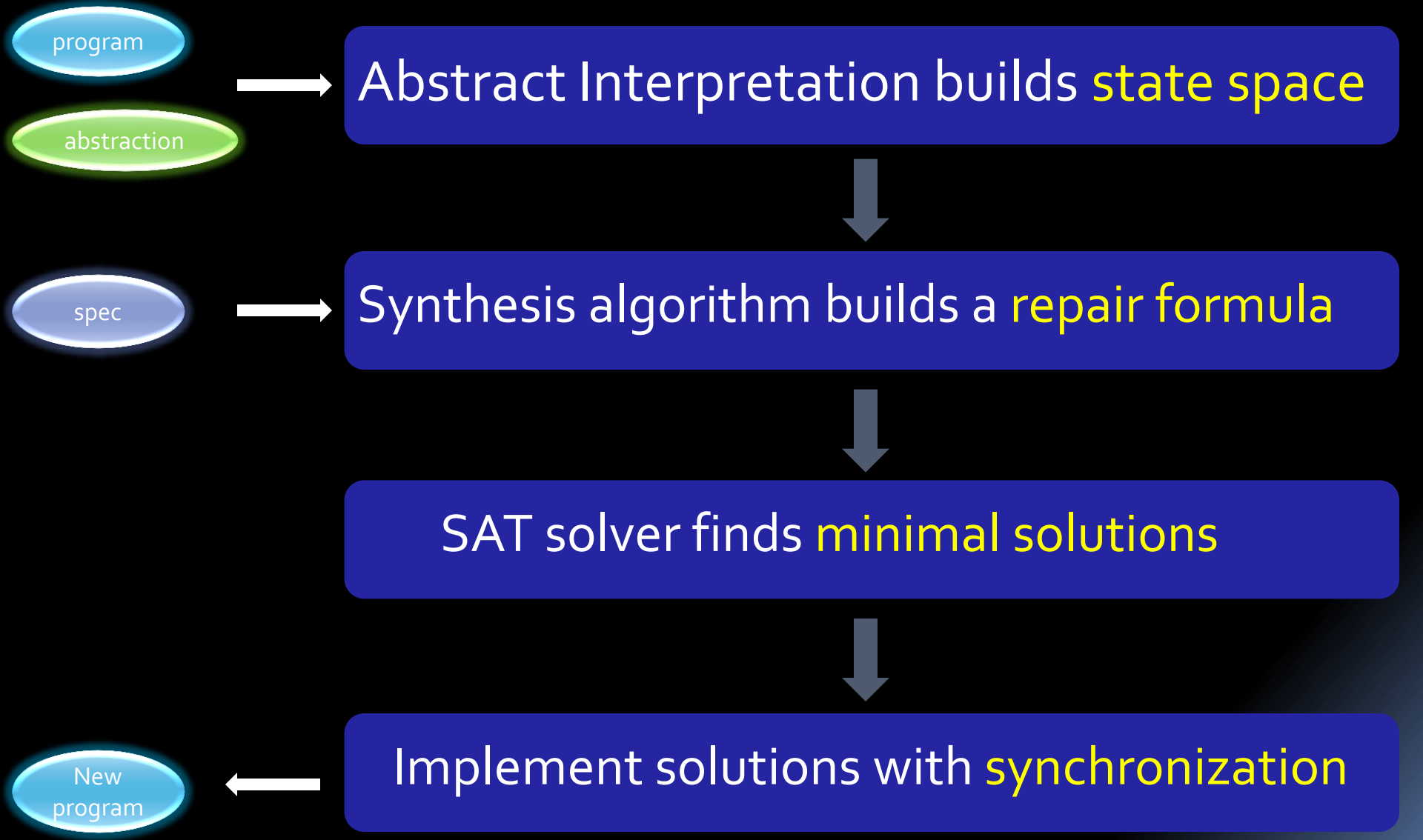


- P' satisfies the specification S under M

Our Approach: Recipe

- Compute reachable states for the program
  - (sometimes under a bound)
- Compute constraints on execution that guarantee that all “bad states” are avoided
- Implement the constraints with fences

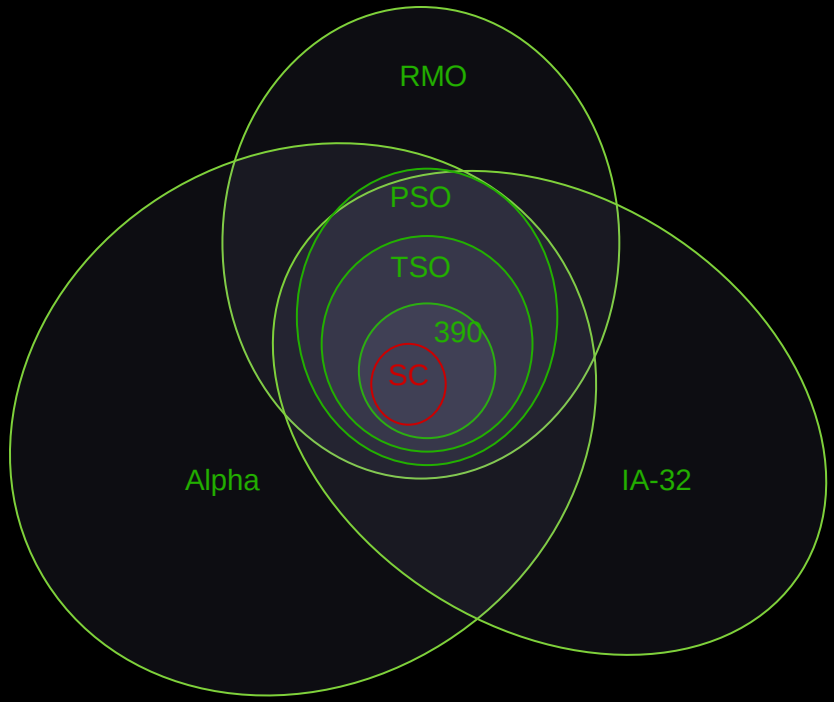
Flow of Synthesis



Our Approach: Ingredients

- Operational semantics for weak memory models
- An algorithm for finding fence constraints
- An algorithm for implementing fence constraints as fences in the program

Operational Semantics for WMM



- An ongoing research topic (e.g., Sewell et al.)
- Building a formal model by reverse engineering decisions made by hardware designers

Operational Semantics for WMM

Relaxation	W→R Order	W→W Order	R→RW Order	R others' W early	R own W early
SC					✓
IBM 370	✓				
TSO	✓				✓
PSO	✓	✓			✓
Alpha	✓	✓	✓		✓
RMO	✓	✓	✓		✓
PowerPC	✓	✓	✓	✓	✓

Classification due to Adve et al. IEEE Computer '95

Operational Semantics for WMM

Relaxation	W→R Order	W→W Order	R→RW Order	R others' W early	R own W early
SC					✓
IBM 370	✓				
TSO	✓				✓
PSO	✓	✓			✓
Alpha	✓	✓	✓		✓
RMO	✓	✓	✓		✓
PowerPC	✓	✓	✓	✓	✓

Classification due to Adve et al. IEEE Computer '95

Operational Semantics for WMM

- Challenges
  - Model store buffers
  - Model execution buffers
  - Variety of re-ordering rules
- Semantics on board

Operational Semantics: State

Initially X = Y = R1 = R2 = 0

Processor A

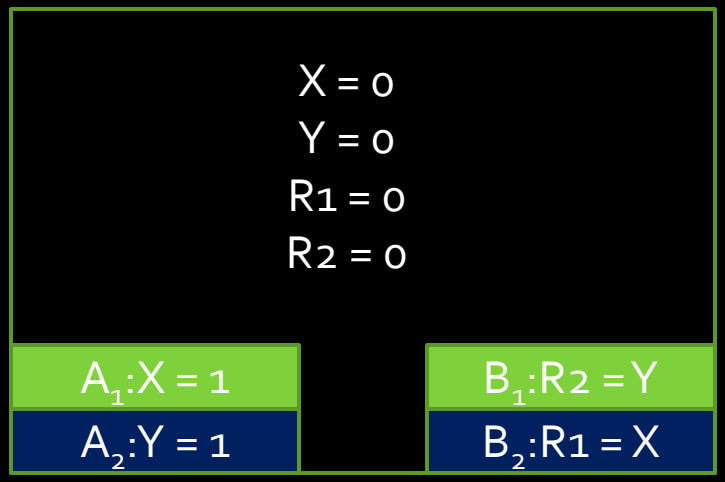
A_1: X = 1

A_2: Y = 1

Processor B

B_1: R2 = Y

B_2: R1 = X



Operational Semantics: Transition

Initially X = Y = R1 = R2 = 0

Processor A

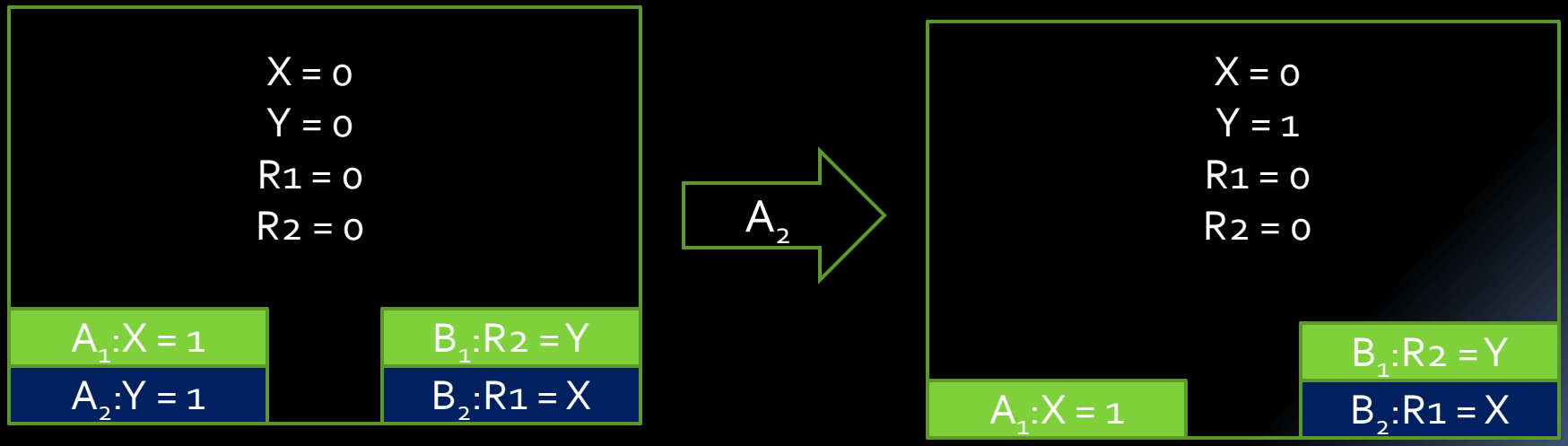
A_1: X = 1

A_2: Y = 1

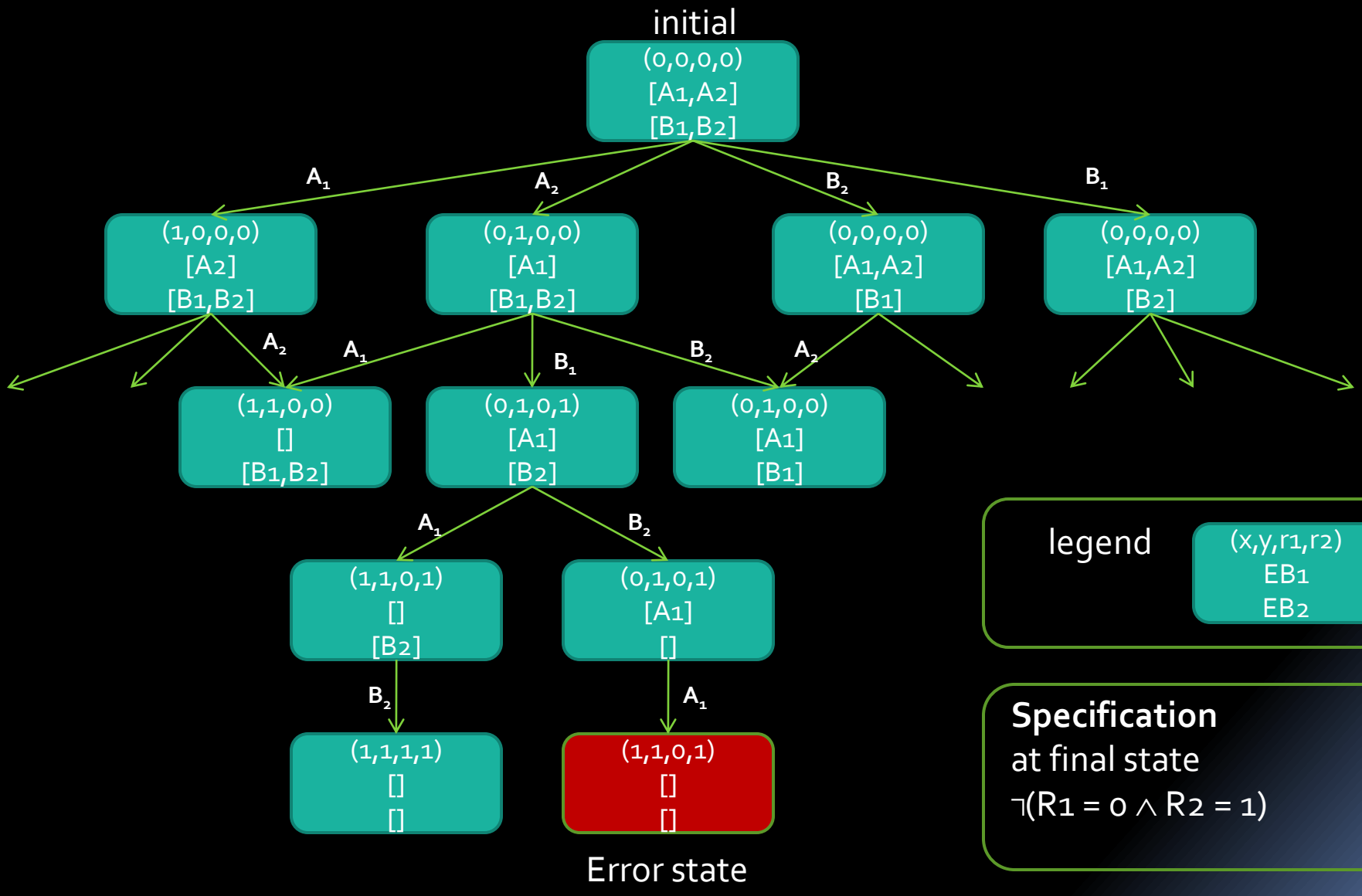
Processor B

B_1: R2 = Y

B_2: R1 = X

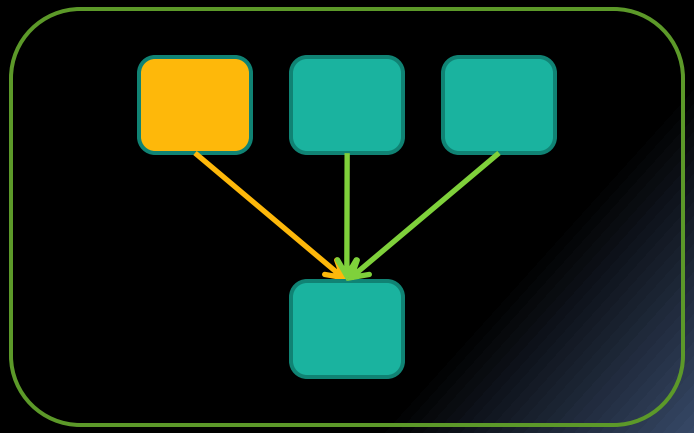


Compute Reachable States



Avoiding a State

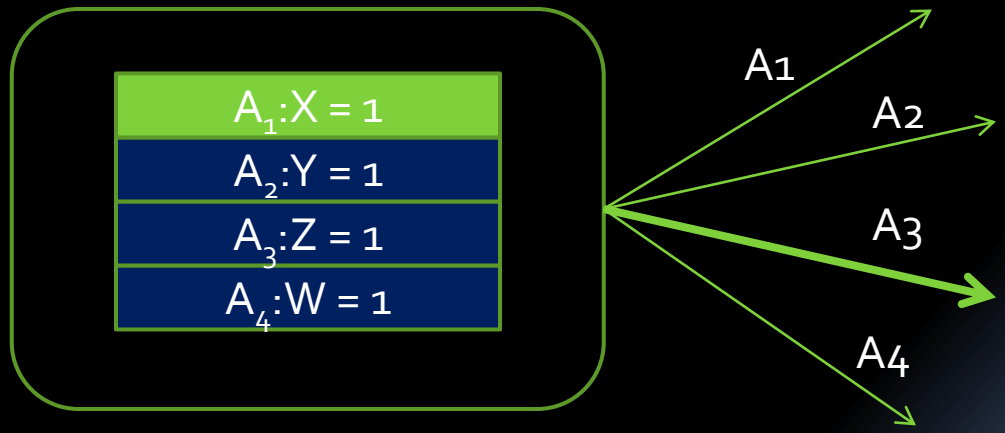
- To avoid a state
  - Avoid **all** incoming transitions
- To avoid an incoming transition
  - Either avoid the transition itself
  - **Or** avoid the source state



Avoidable Transitions

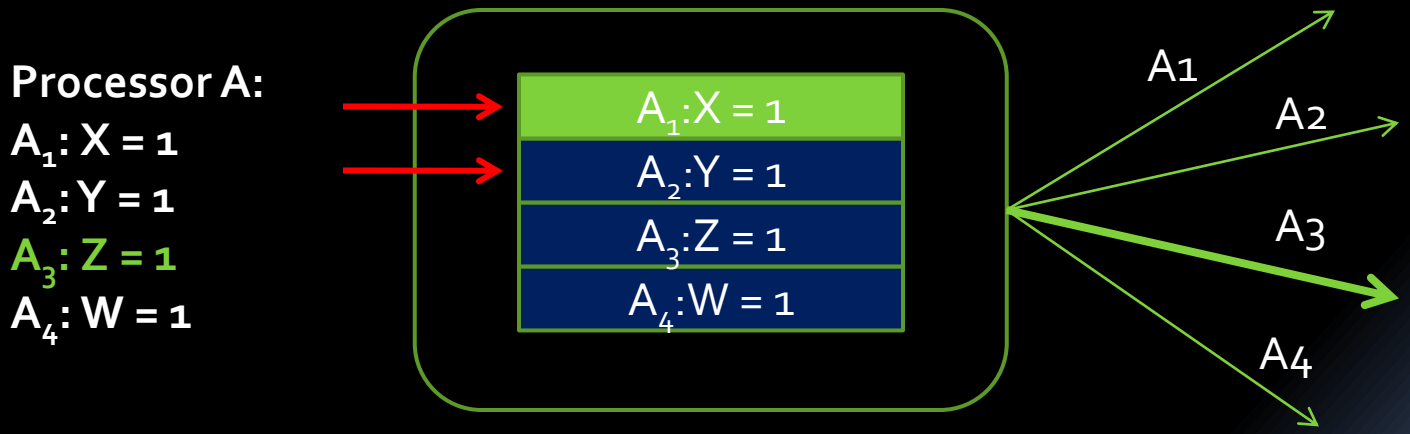
- Execution buffer is ordered
- A transition not executing first instruction in the execution buffer can be avoided
  - By forcing a different transition to execute

Processor A
A_1: X = 1
A_2: Y = 1
A_3: Z = 1
A_4: W = 1



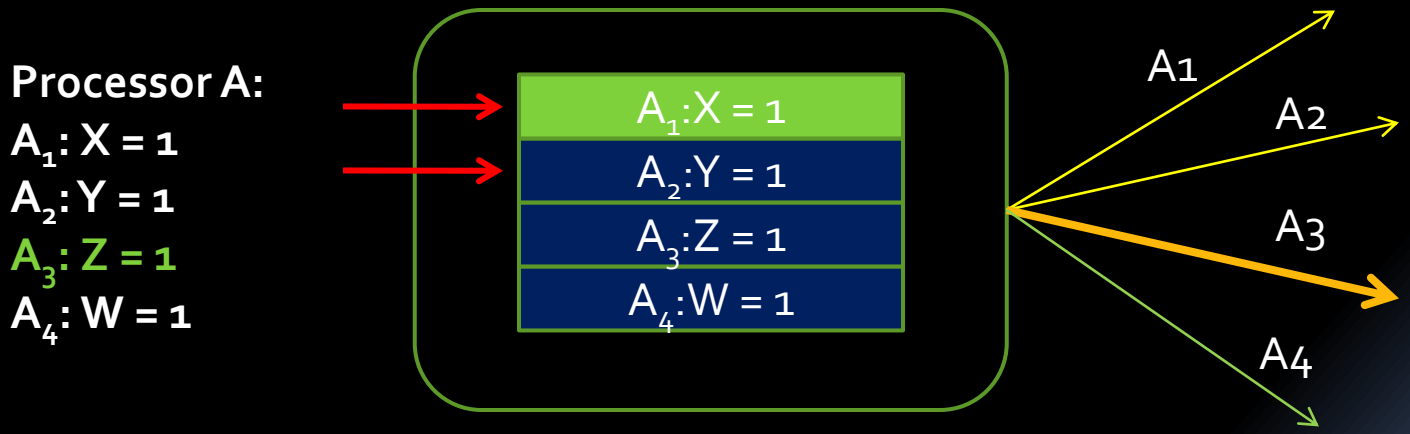
Avoidable Transitions

- Execution buffer is ordered
- A transition not executing first instruction in the execution buffer can be avoided
  - By forcing a different transition to execute



Avoidable Transitions

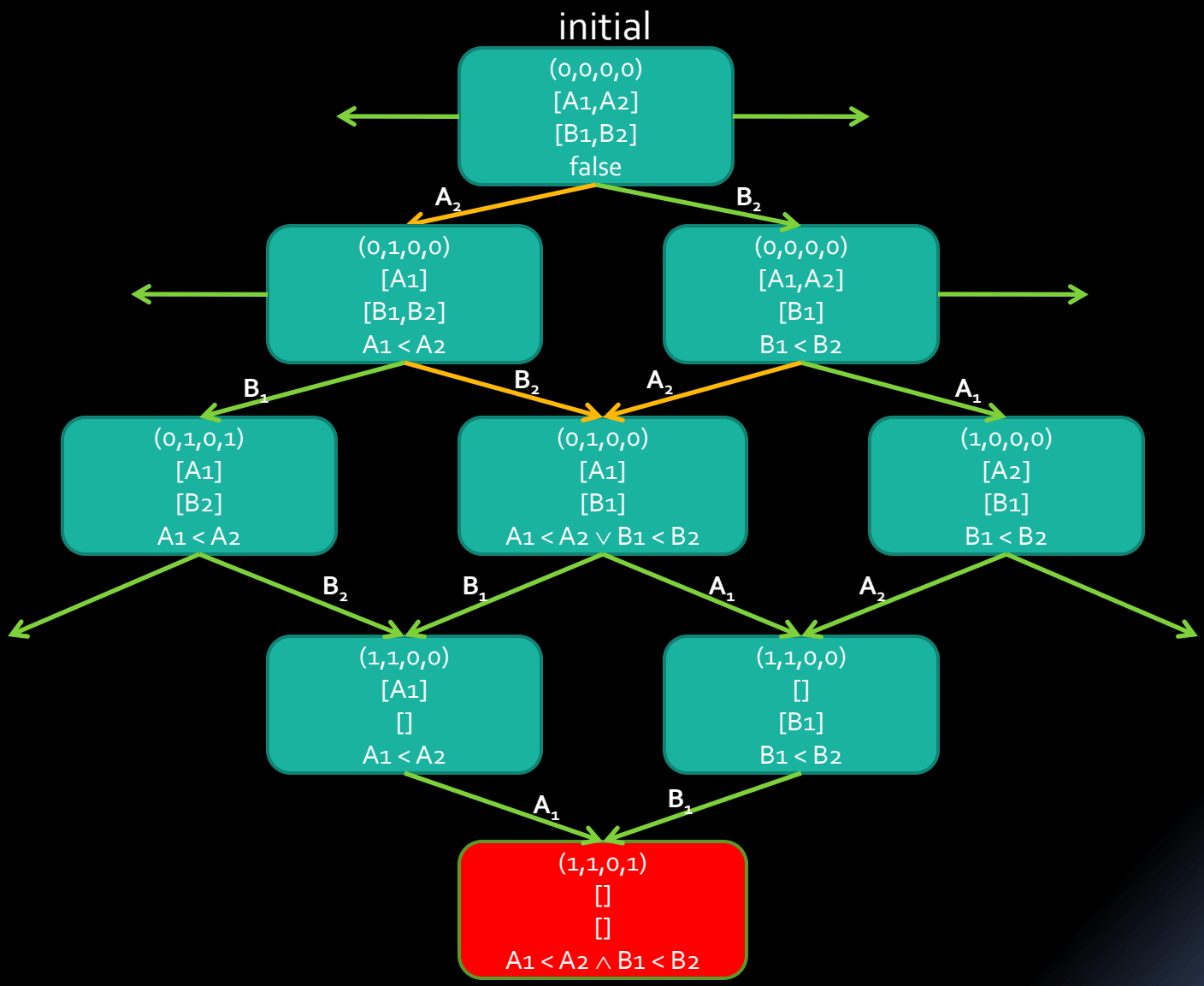
- To avoid A_3 in this state
  - Force A_1 to execute before A_3
  - Or force A_2 to execute before A_3
- Language of ordering constraints
  - $A_1 < A_3 \vee A_2 < A_3$



Computing Avoid Formulae

- Ordering predicates
  - $l_1 < l_2$
  - l_1 must execute before l_2
- Ordering constraints are (positive) Boolean combinations of ordering predicates
- Fixed-point computation computes an avoid formula for every state
- Final constraints is conjunction avoiding all "bad states"

Back to Our Example



Fence Placement

$$A1 < A2 \wedge B1 < B2$$

Processor A

A_1: X = 1
fence("store-store")
A_2: Y = 1

Processor B

B_1: R2 = Y
fence("load-load")
B_2: R1 = X

Fence Placement

- Trivial in the previous example
  - Satisfying assignment to the ordering constraint
  - Every ordering predicate realized as a fence
  - Only had to choose fence type
- Find minimal number of fences
  - Minimal satisfying assignments?
  - (obviously?) not good enough due to transitive dependencies

Fence Placement

```
L1: STORE X, 1
L2: LOAD R1, Y
L3: LOAD R2, Y
L4: LOAD R3, Y
L5: STORE Z, 1
```

Constraint: $L1 < L5$

Fence Placement

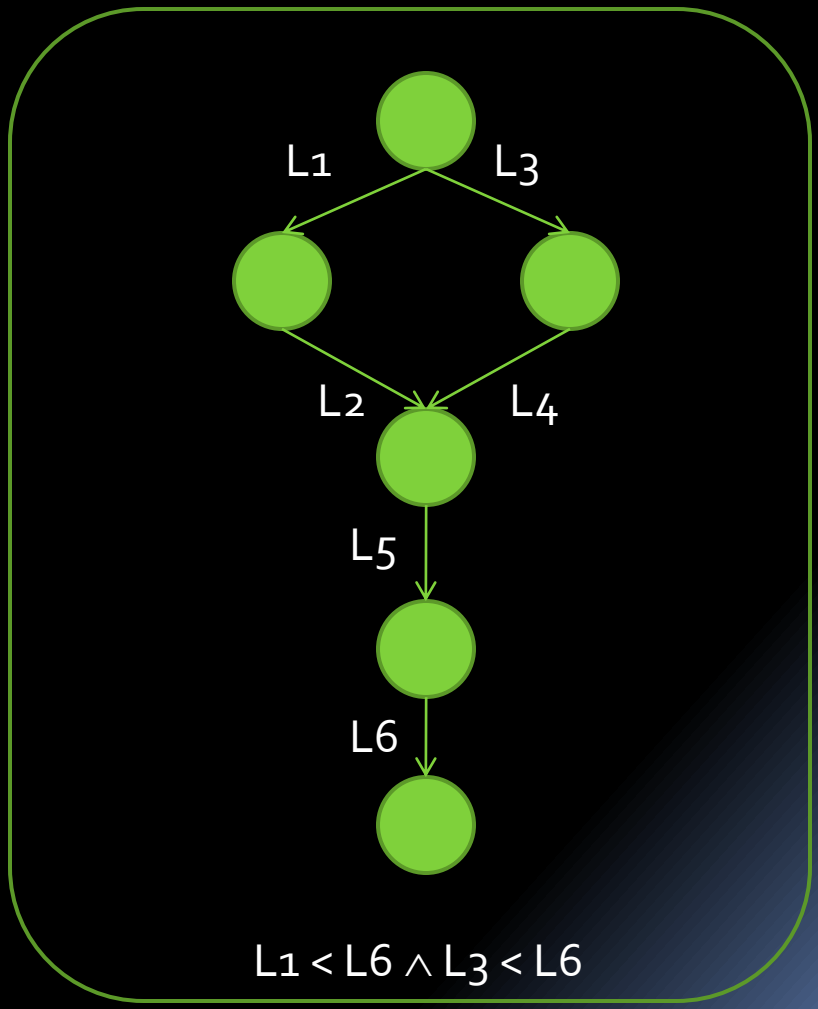
```
L1: STORE X, 1
L2: LOAD R1, Y
L3: LOAD R2, Y
L4: LOAD R3, Y
L5: STORE Z, 1
```

Constraint: $L1 < L5 \wedge L2 < L5$

Fence Placement

- A fence placed on every CFG path between ordered statements

- Fence types
  - Full
  - Partial (e.g., store-load)



Evaluation

- Pick a data structure
  - Preferably – one with known correct fences
- Pick a "reasonable" set of clients
  - Intuition
  - Exhaustive up to some bound
- Run
- Examine results

Results

	Initial	Client	$\|E\|$ Bound	RMO			PSO			TSO			SC	
				States	Edges	#C	States	Edges	#C	States	Edges	#C	States	Edges
MSN	empty	e|d	∞	1219	2671	2	455	743	1	228	316	0	146	180
	empty	e|e	∞	4934	12670	1	2678	6354	1	586	994	0	252	328
	empty	ee|dd	∞	24194	61514	3	7025	13689	2	1724	2512	0	1029	1325
	empty	ed|ed	∞	86574	242822	2	15450	35362	2	2476	3972	0	1538	2126
	empty	ed|de	∞	59119	167067	4	11023	24362	2	2570	4010	0	1541	2073
	empty	e|e|d	∞	233414	653094	3	51990	119050	2	9638	16822	0	4928	7632
Chase-Lev	empty	pppt(tpt|sss)	∞	386283	1030857	-	74533	256613	-	12348	20004	-	4961	6740
WSQ	empty	tttt(ptt|sss)	∞	1048498	2819355	-	124455	255390	-	6418	9380	-	3101	4069
	empty	pppt(ttp|sss)	∞	281314	878880	-	66960	241814	-	10564	16317	-	4199	5700
	empty	tttt(tpp|sss)	∞	1325858	4150650	-	361855	1080835	-	9878	13956	-	3473	4537
	empty	tttp(tptp|ss)	∞	280396	698398	-	29573	54696	-	9197	14499	-	4760	6455
"LIFO"	2/2	tp|ss	∞	2151	3190	2	882	1171	1	676	852	0	570	694
WSQ	2/2	tpt|ss	∞	9721	16668	2	3908	5811	1	2256	3116	0	1410	1786
	2/2	ptp|ss	∞	89884	195246	3	31289	64133	3	4045	5688	0	2317	3007
	2/2	ptt|ss	∞	85104	198353	4	29920	62020	3	4130	5987	0	2198	2866
	1/1	ptt|ss	∞	23913	48997	4	9976	18002	3	2353	3269	0	1314	1654
Dekker	-	-	1	1388	2702	2	1388	2702	2	489	674	1	388	490
	-	-	10	7504	14477	2	7504	14477	2	2560	3750	1	388	490
	-	-	20	13879	26422	2	13879	26422	2	4845	7115	1	388	490
	-	-	50	33004	62257	2	33004	62257	2	11770	17210	1	388	490
Treiber	empty	p|t	∞	71	93	2	71	93	2	43	48	0	36	38
	empty	pt|tp	∞	3054	6190	2	3041	6167	2	407	609	0	392	482
	empty	pp|tt	∞	1276	2250	2	1276	2250	2	325	407	0	270	323
VYSet	empty	ar|ra	10	4079	6247	2	4079	6247	2	1088	1308	0	1088	1308
	empty	aa|rr	10	20034	31623	2	20034	31623	2	1168	1411	0	1168	1411
	empty	ar|ar	10	6093	9905	2	6093	9905	2	1671	1968	0	1671	1968
	empty	aaa|rrr	10	41520	66533	2	41520	66533	2	3311	4072	0	3311	4072

Performance

	Initial State	Client	$\|E\|$ Bnd	Time (sec.) total	build	form.	solve
MSN	empty	e|d	∞	0.83	0.45	0.14	0.23
	empty	e|e	∞	1.78	1.15	0.54	0.07
	empty	ee|dd	∞	5.21	2.87	2.12	0.21
	empty	ed|ed	∞	13.05	9.14	3.73	0.18
	empty	ed|de	∞	9.26	6.54	2.52	0.19
	empty	e|e|d	∞	31.43	23.03	8.2	0.18
Chase-Lev WSQ	empty	pppt(tpt|sss)	∞	97.22	56.1	41.11	-
	empty	tttt(ptt|sss)	∞	255.5	160.06	95.44	-
	empty	pppt(ttp|sss)	∞	90.28	45.17	45.11	-
	empty	tttt(tpp|sss)	∞	355.95	212.29	143.65	-
	empty	tttp(tptp|ss)	∞	37.98	31.34	6.63	-
"LIFO" WSQ	2/2	tp|ss	∞	0.69	0.5	0.14	0.05
	2/2	tpt|ss	∞	1.94	1.27	0.61	0.06
	2/2	ptp|ss	∞	11.41	8.27	3.05	0.09
	2/2	ptt|ss	∞	11.31	8.26	2.99	0.06
	1/1	ptt|ss	∞	4.07	2.57	1.44	0.06
Dekker	-	-	1	0.64	0.39	0.19	0.05
	-	-	10	2.13	1.15	0.92	0.05
	-	-	20	2.71	1.68	0.98	0.05
	-	-	50	5.99	4.46	1.46	0.05
Treiber	empty	p|t	∞	1	0.07	0.02	0
	empty	pt|tp	∞	1.02	0.76	0.26	0
	empty	pp|tt	∞	0.6	0.46	0.14	0
VYSet	empty	ar|ra	10	1.98	1.49	0.41	0.08
	empty	aa|rr	10	4.56	3.41	1.1	0.05
	empty	ar|ar	10	2.19	1.7	0.43	0.05
	empty	aaa|rrr	10	7.98	5.61	2.24	0.13

Optimization and Scaling

- Order of iteration
  - For acyclic graphs, we have a simple recursive algorithm
- Constraints are shared
  - For several “instances” of the same inlined function
  - Between processors
- “Safe State” pruning

Sources of Unboundedness

- Unbounded heap
- Unbounded execution buffer

Basic Recipe

- Compute reachable states of the program
- Compute constraints on execution that guarantee that all “bad states” are avoided
- Implement the constraints with fences

Basic Recipe

- Compute reachable states of the program

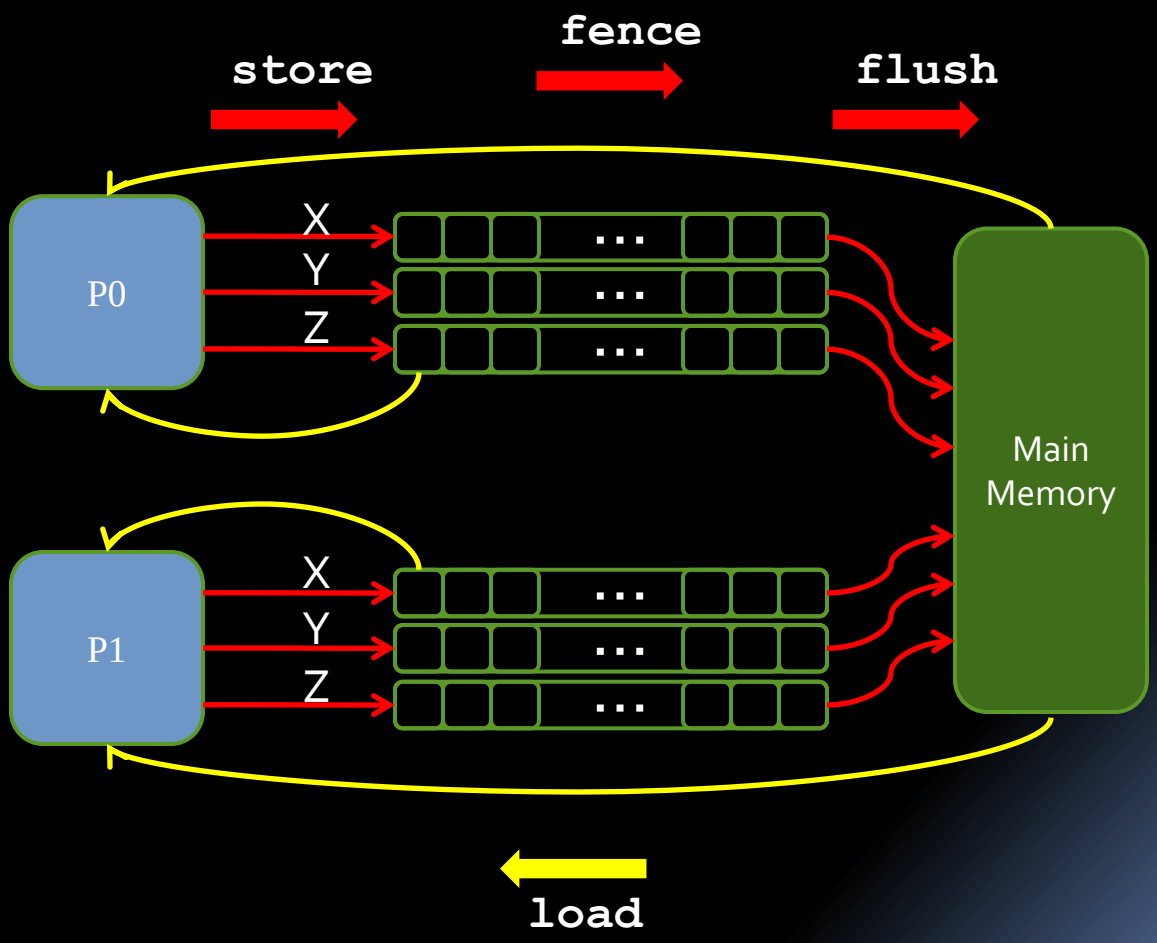
Bad News [Atig et. al POPL'10]

Reachability undecidable for RMO

Non-primitive recursive complexity for TSO/PSO

Example: store-buffer models

- TSO & PSO
  - Intel x86 is ~ TSO
- Memory Fences
  - Restore order
  - *Every store before the fence becomes globally visible before anything after the fence executes*



Mutual Exclusion Algorithm

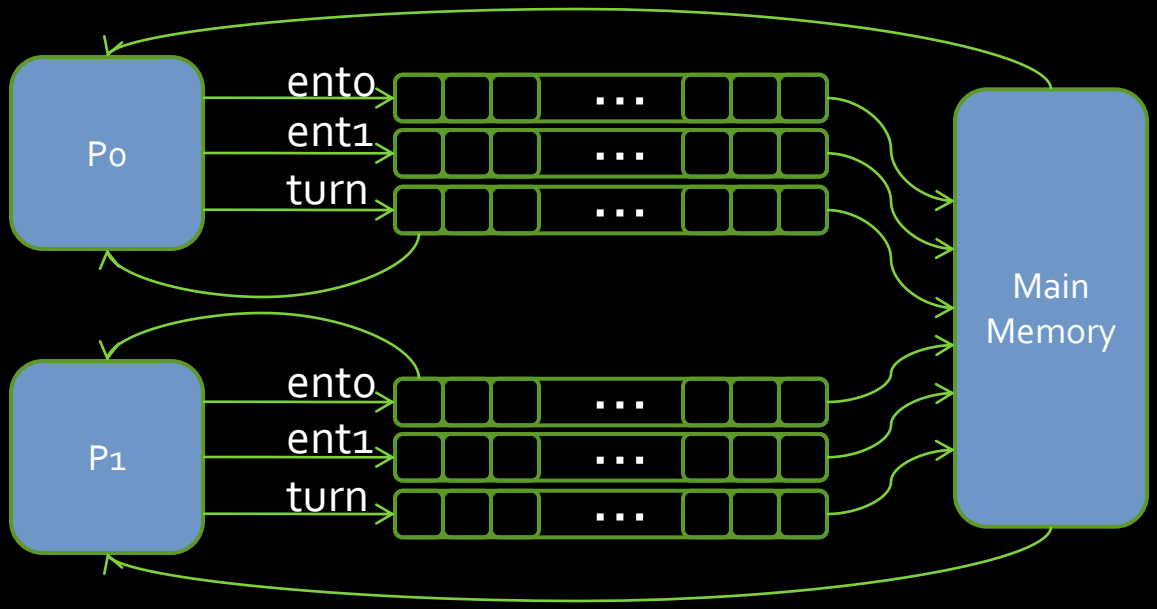
Process 0:

```
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```

Process 1:

```
while(true) {
 store ent1 = true;
 fence;
 store turn = 0;
 fence;
 do {
  load e = ento;
  load t = turn;
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

Store Buffers



Unbounded Store Buffers…

- Even for very simple patterns
  - e.g. spin-loops with writes in loop body

```
flag := true
while other_flag = true {
  flag := false
  //Do something
  flag := true
}
```

true

Unbounded Store Buffers…

- Even for very simple patterns
  - e.g. spin-loops with writes in loop body

```
flag := true
while other_flag = true {
  flag := false
  //Do something
  flag := true
}
```

false true

Unbounded Store Buffers…

- Even for very simple patterns
  - e.g. spin-loops with writes in loop body

```
flag := true
while other_flag = true {
  flag := false
  //Do something
  flag := true
}
```

true false true

Unbounded Store Buffers…

- Even for very simple patterns
  - e.g. spin-loops with writes in loop body

```
flag := true
while other_flag = true {
  flag := false
  //Do something
  flag := true
}
```

false | true | false | true

Unbounded Store Buffers…

- Even for very simple patterns
  - e.g. spin-loops with writes in loop body

```
flag := true
while other_flag = true {
  flag := false
  //Do something
  flag := true
}
```

What can we do?

- Under-approximate
  - Bound the length of execution buffers (previously mentioned). Implies a bound on state space
  - Bound context switches (other work)
  - Dynamic synthesis (PLDI'12)
- Over-approximate
  - Sound abstraction of buffer content, next

Abstract Interpretation for RMM

Main Idea: Bounded over-approximation of unbounded buffers

First Attempt: Set Abstraction

Process 0:

```
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```

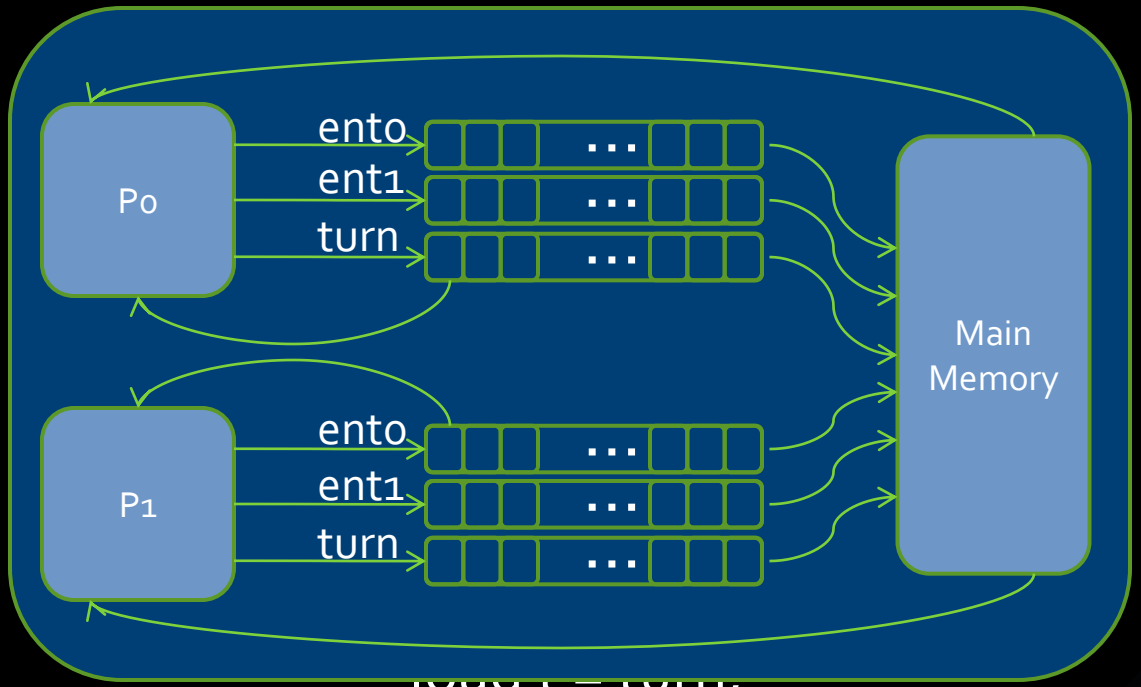
Process 1:

```
while(true) {
 store ent1 = true;
 fence;
 store turn = 0;
 fence;
 do {
  load e = ento;
  load t = turn;
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

```
Process 0:
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```

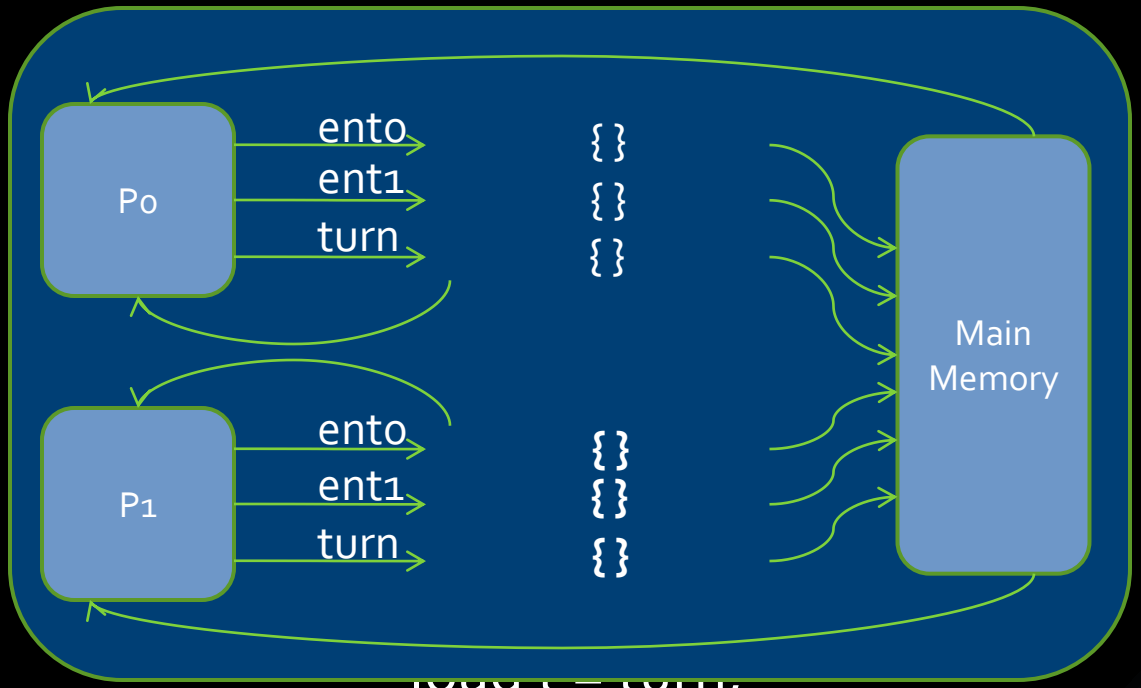



```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

```
Process 0:
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```



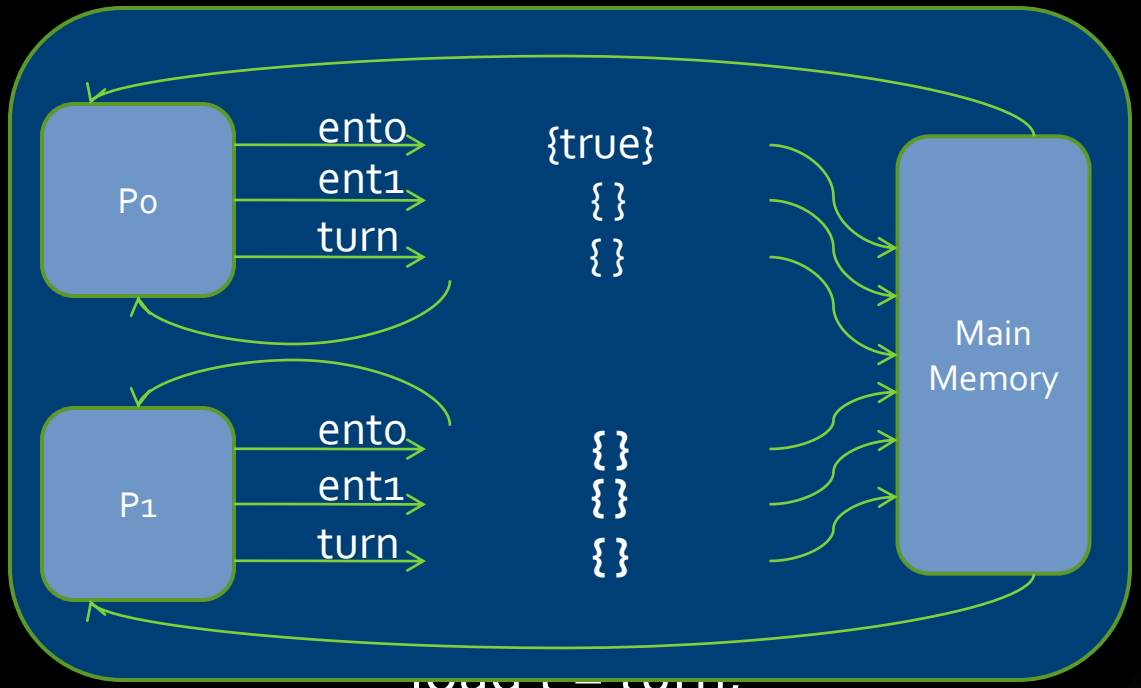

```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

Process 0:

```
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```

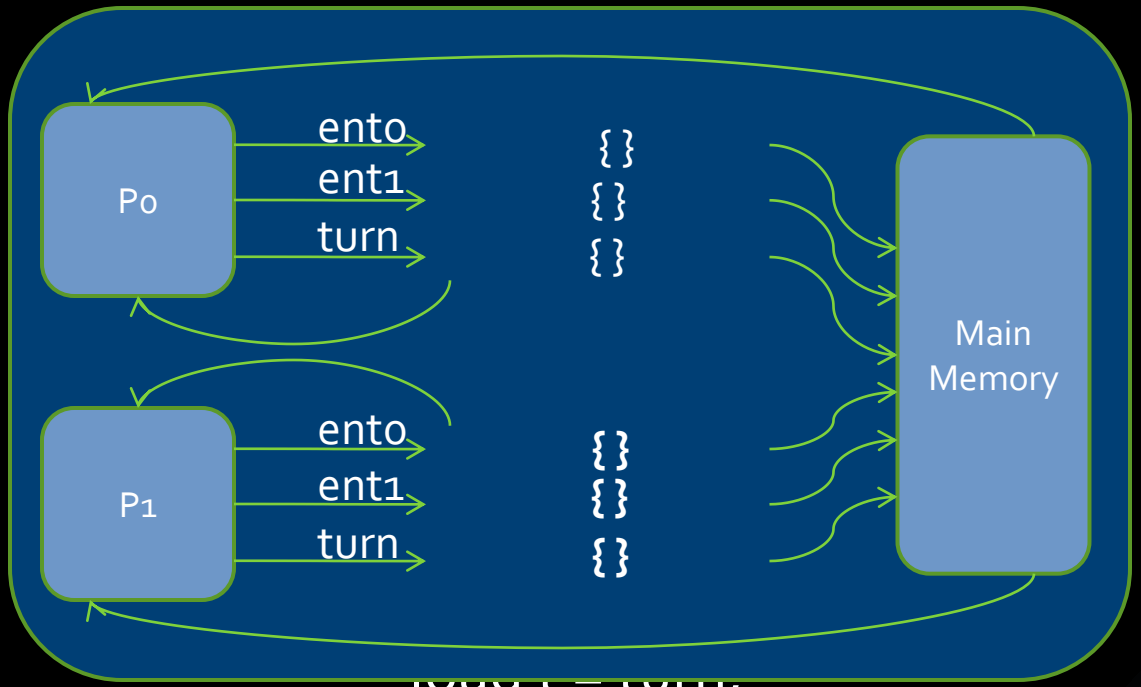



```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

```
Process 0:
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```

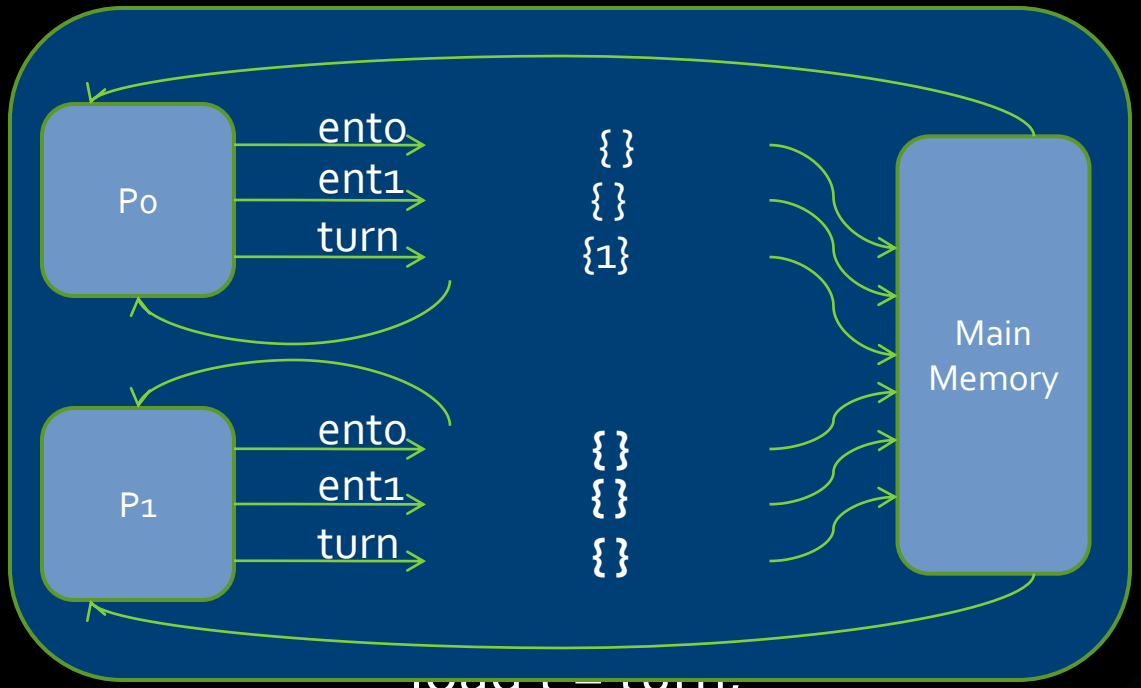



```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

```
Process 0:
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```

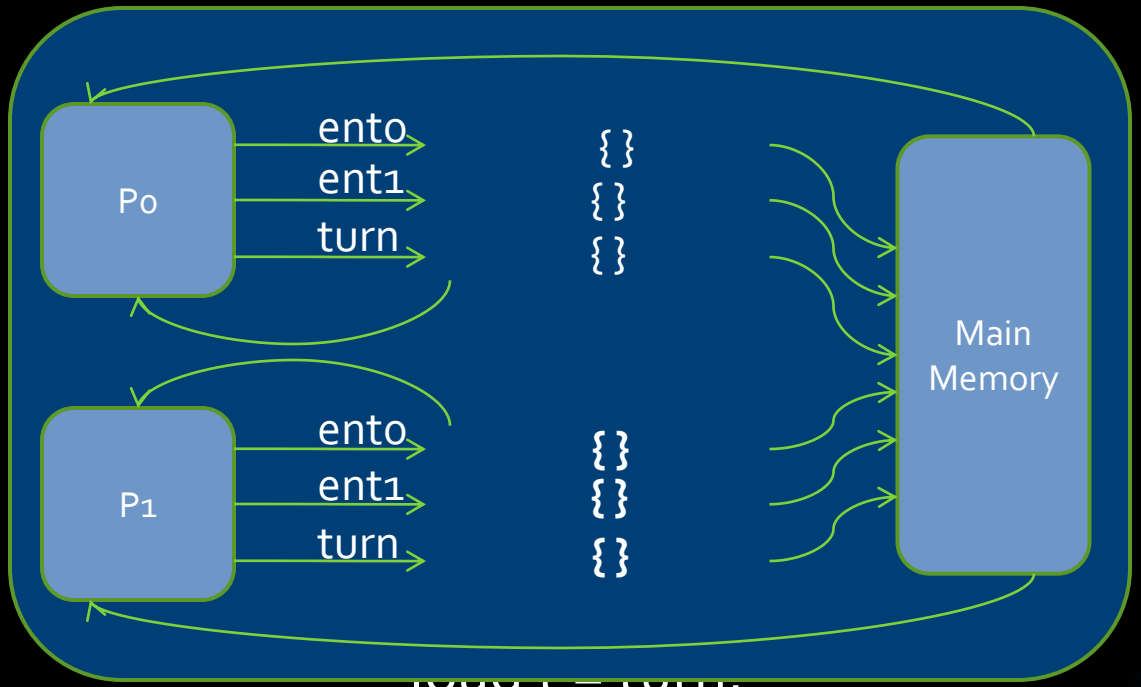



```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

```
Process 0:
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```

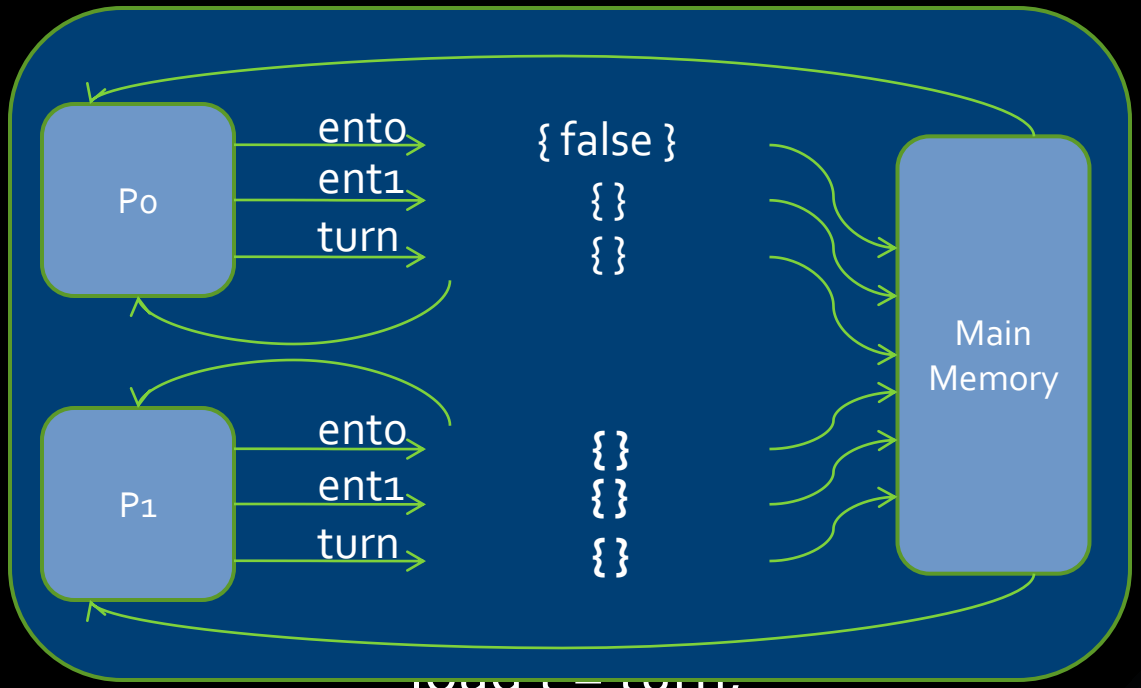



```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

```
Process 0:
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```



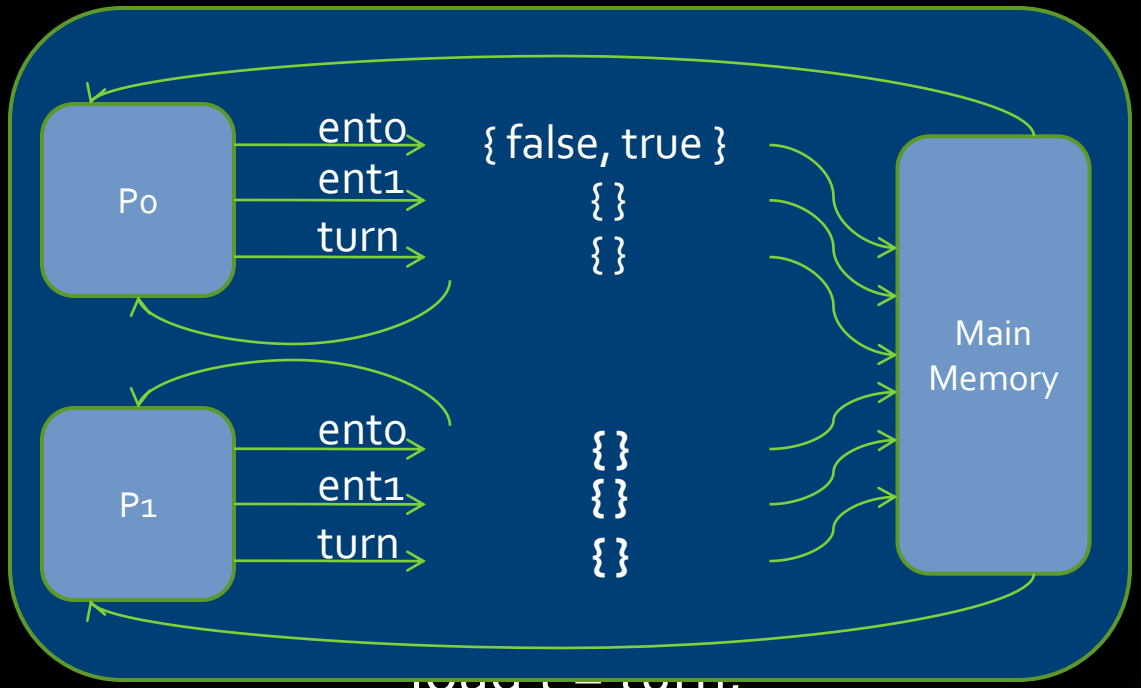

```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

Process 0:

```
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```

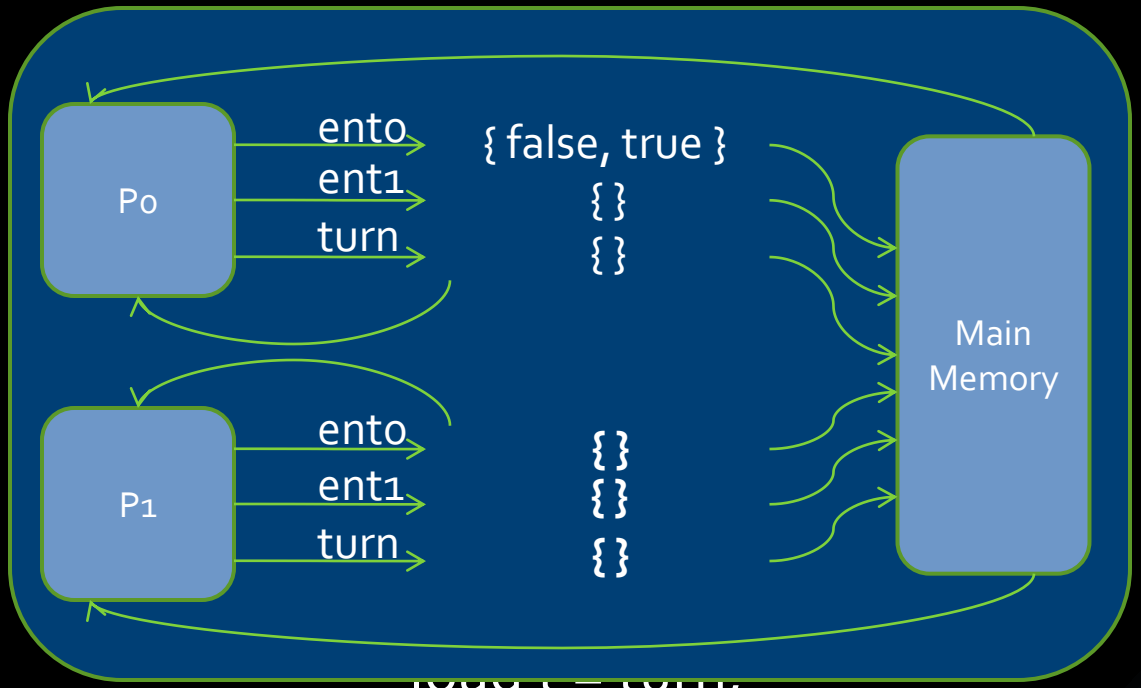



```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

- Abstract each store buffer as a set

First Attempt: Set Abstraction

```
Process 0:
while(true) {
 store ento = true;
 fence;
 store turn = 1;
 fence;
 do {
  load e = ent1;
  load t = turn;
 } while(e == true && t == 1);
 //critical section here
 store ento = false;
}
```




```
 } while(e == true && t == 0);
 //critical section here
 store ent1 = false;
}
```

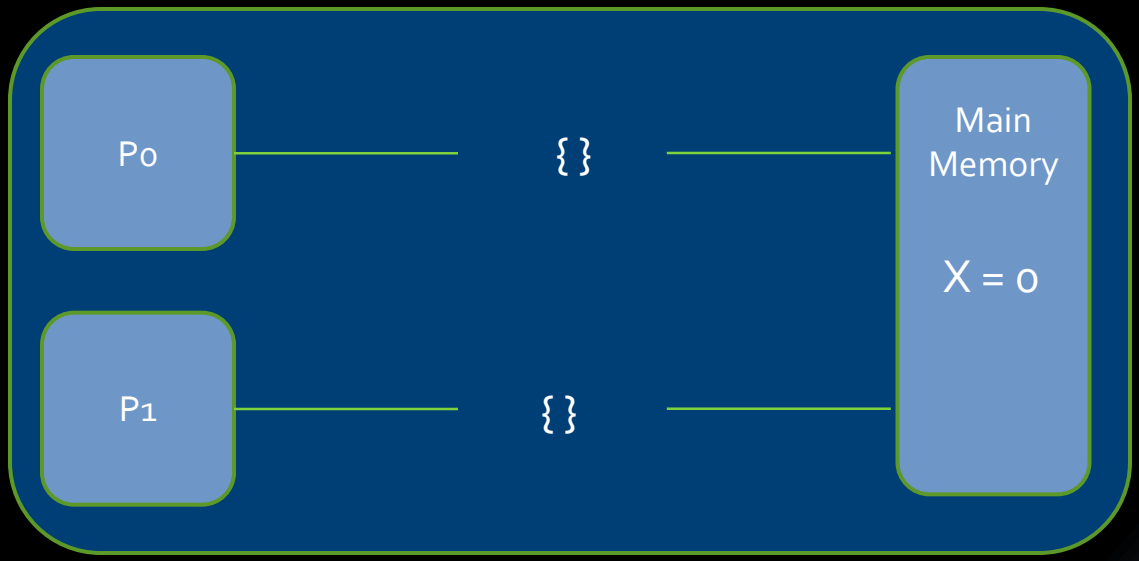
- Abstract each store buffer as a set

Second Attempt: record most recent value

Initially X == 0

Process 0	Process 1

X := 1	while (X != 1) { nop }
	X := 2
	fence
	e := X
	assert e == 2;

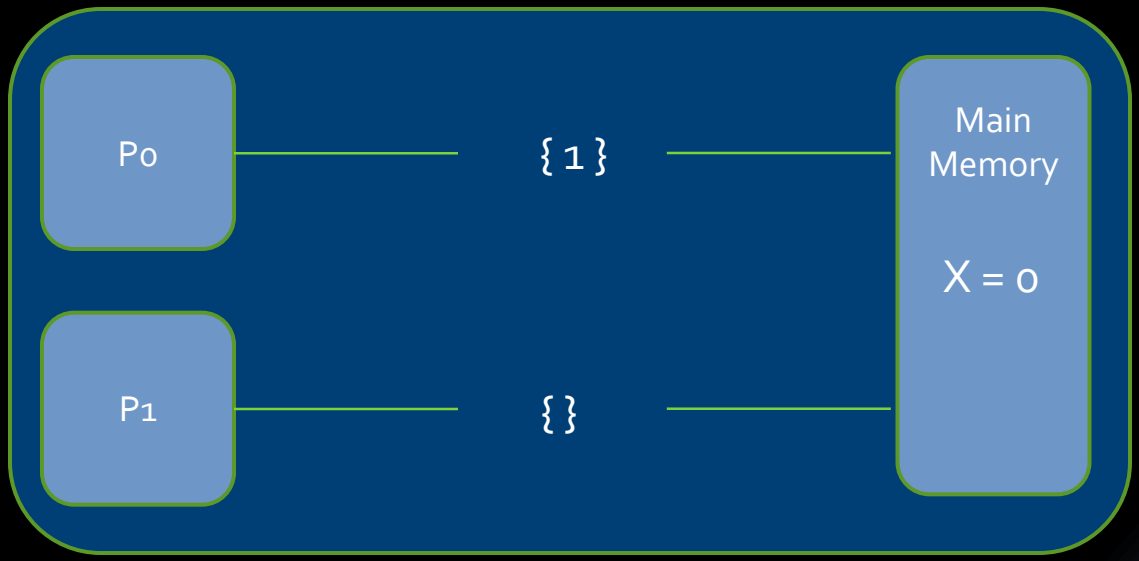


Second Attempt: record most recent value

Initially X == 0

Process 0	Process 1

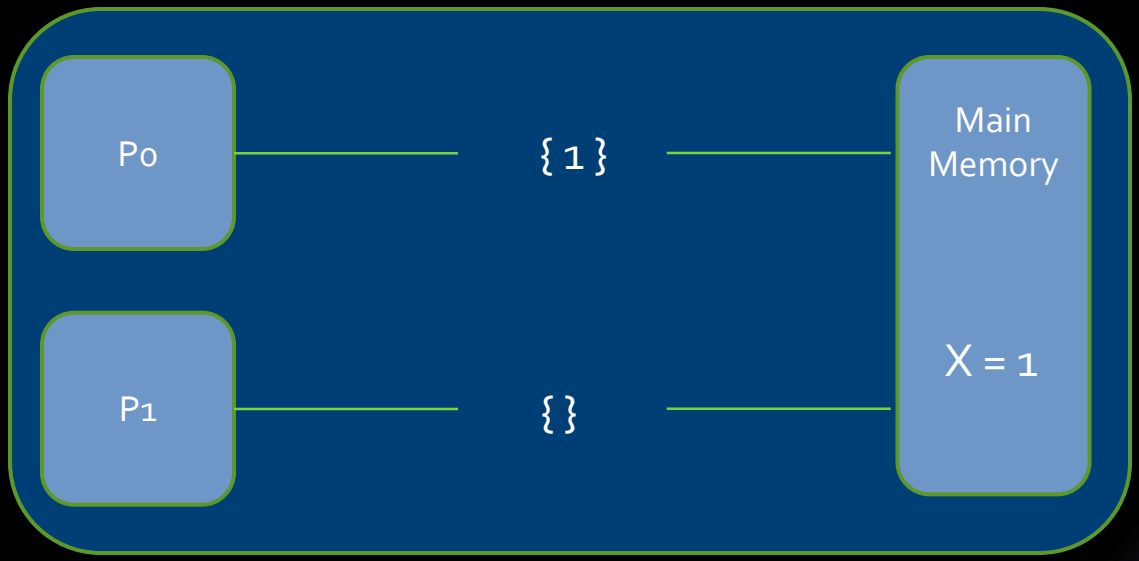
X := 1	while (X != 1) { nop }
	X := 2
	fence
	e := X
	assert e == 2;



Second Attempt: record most recent value

Initially X == 0

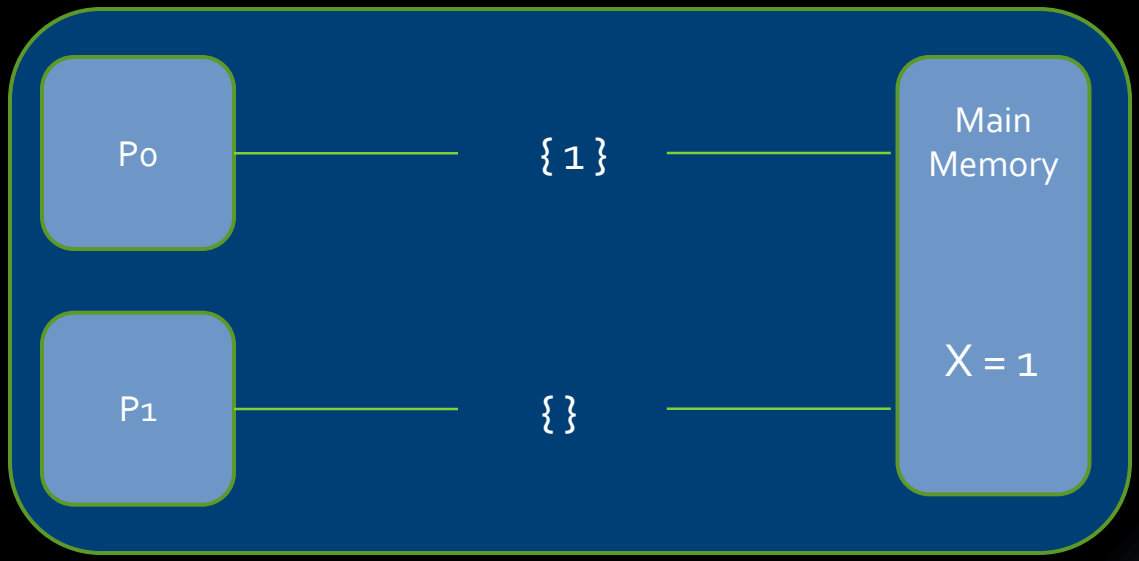
Process 0	Process 1
X := 1	while (X != 1) { nop }
flush (1st time)	X := 2
	fence
	e := X
	assert e == 2;



Second Attempt: record most recent value

Initially X == 0

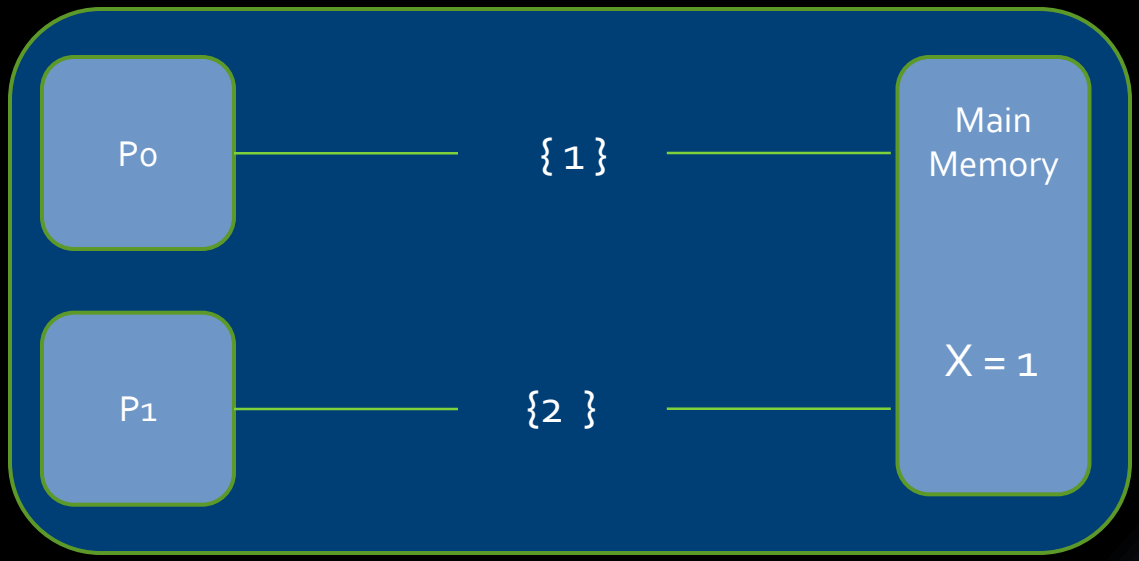
Process 0	Process 1
X := 1	while (X != 1) { nop }
flush (1st time)	X := 2
	fence
	e := X
	assert e == 2;



Second Attempt: record most recent value

Initially X == 0

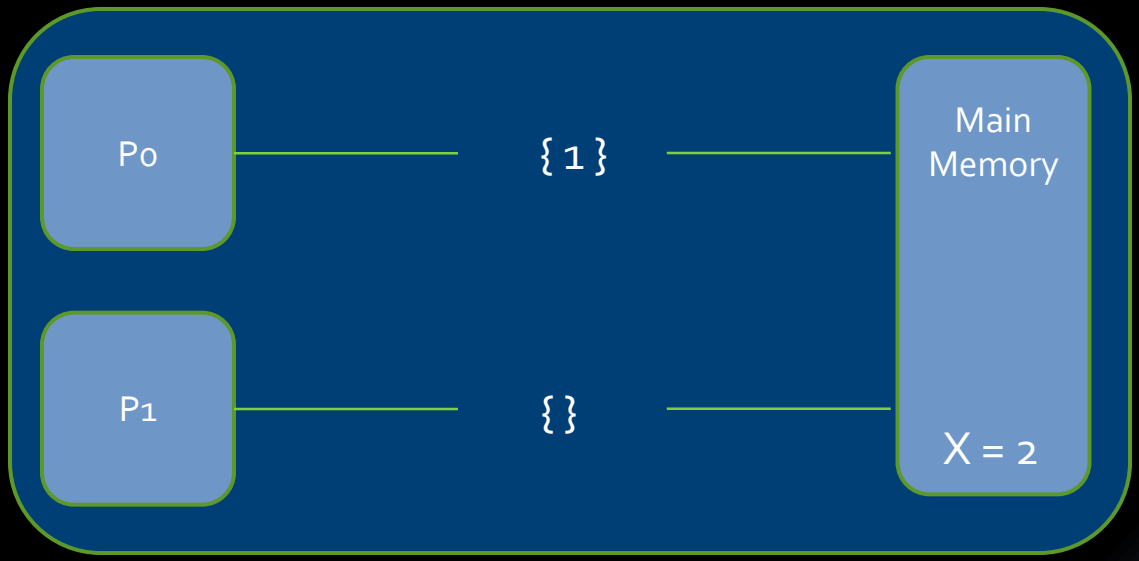
Process 0	Process 1
X := 1	while (X != 1) { nop }
flush (1st time)	X := 2
	fence
	e := X
	assert e == 2;



Second Attempt: record most recent value

Initially X == 0

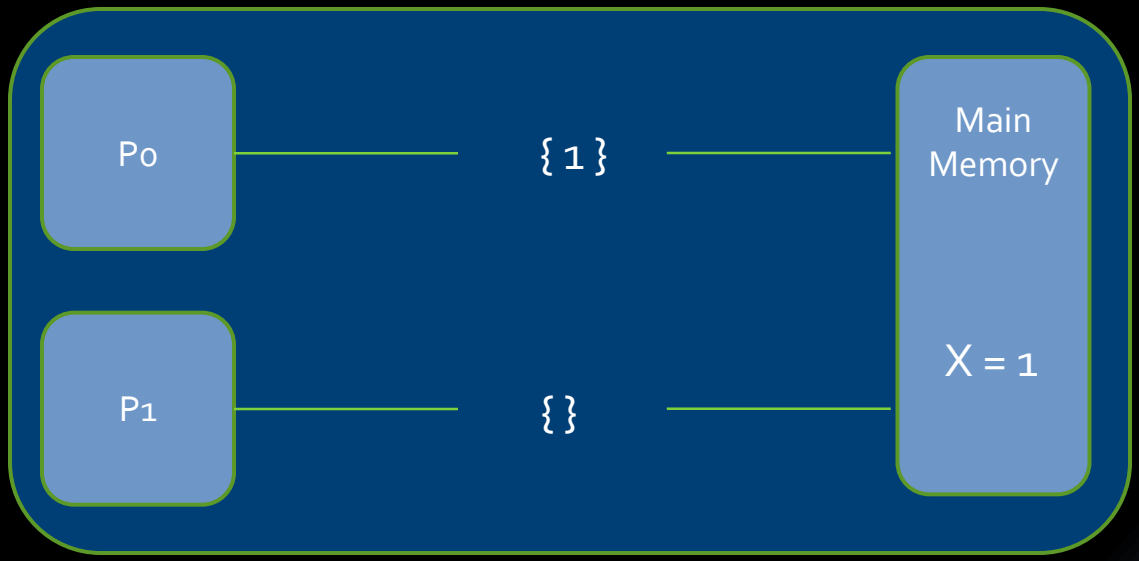
Process 0	Process 1
X := 1	while (X != 1) { nop }
	X := 2
flush (1st time)	**fence**
	e := X
	assert e == 2;



Second Attempt: record most recent value

Initially X == 0

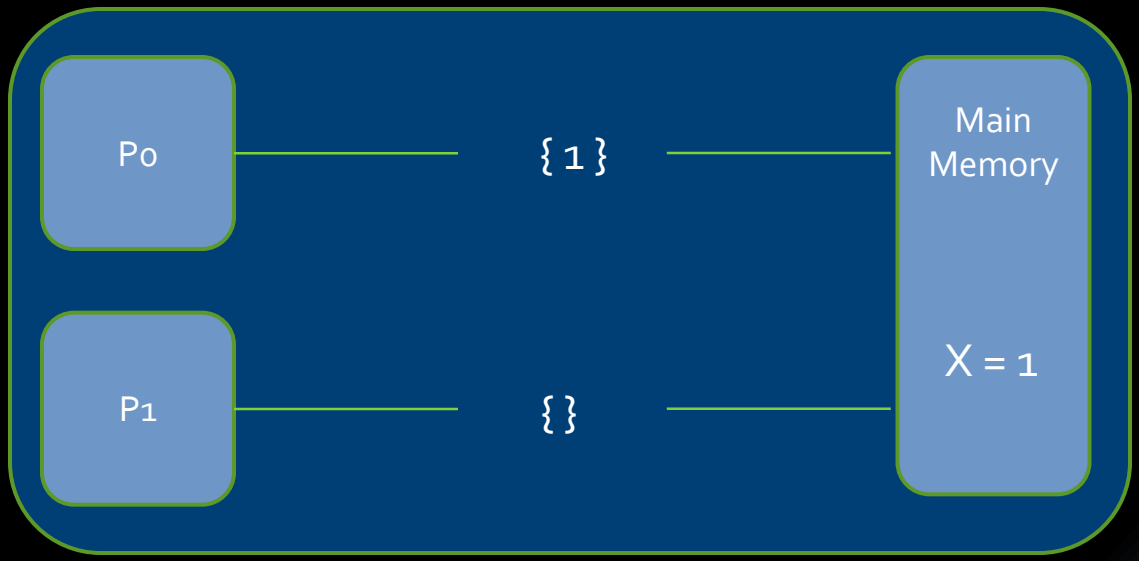
Process 0	Process 1
X := 1	while (X != 1) { nop }
	X := 2
flush (1st time)	**fence**
flush (2nd time)	e := X
	assert e == 2;



Second Attempt: record most recent value

Initially X == 0

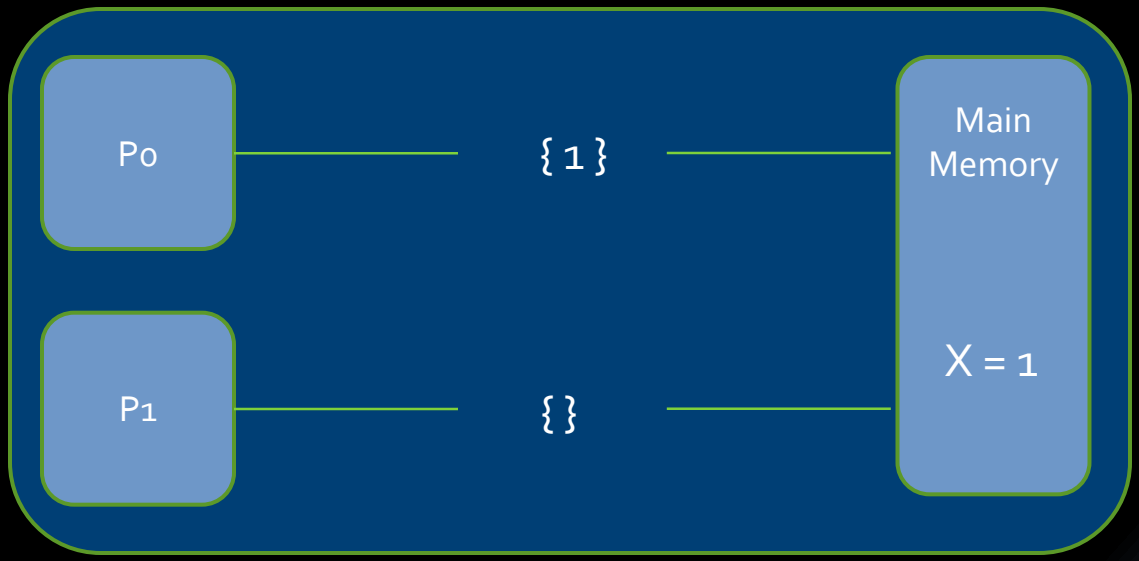
Process 0	Process 1
X := 1	while (X != 1) { nop }
	X := 2
flush (1st time)	**fence**
flush (2nd time)	e := X
	assert e == 2;



Second Attempt: record most recent value

Initially X == 0

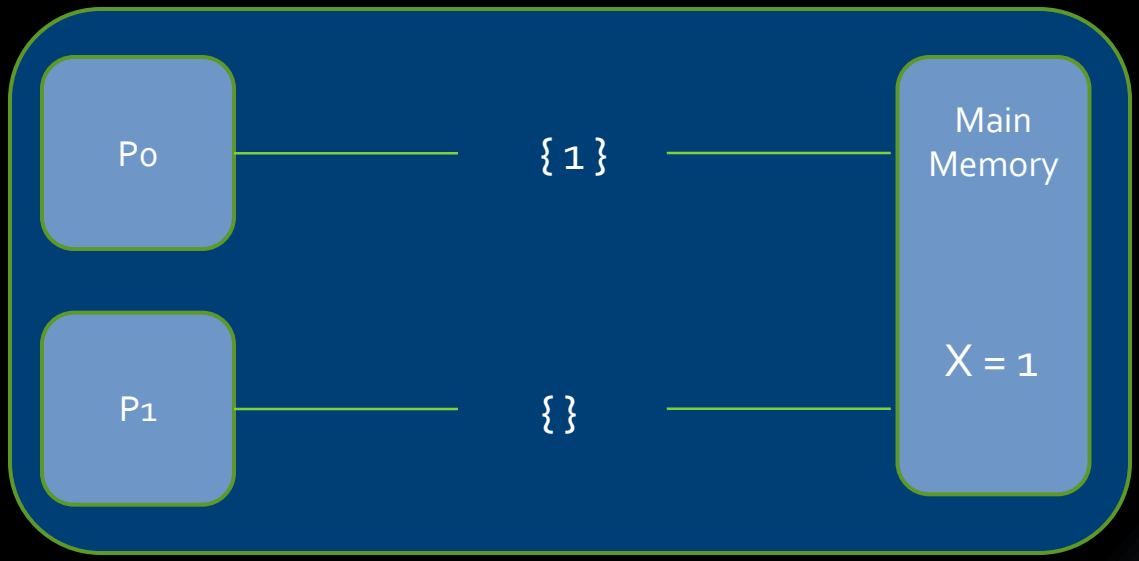
Process 0	Process 1
X := 1	while (X != 1) { nop }
	X := 2
flush (1st time)	**fence**
flush (2nd time)	e := X
	assert e == 2;



Second Attempt: record most recent value

Initially X == 0

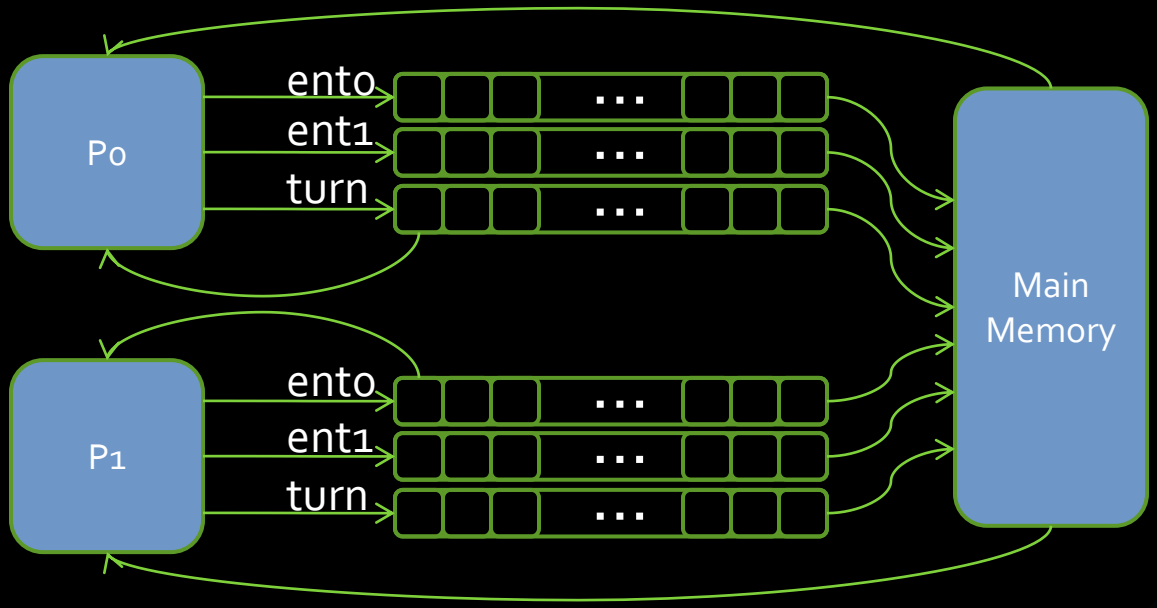
Process 0	Process 1
X := 1	while (X != 1) { nop }
flush (1st time)	X := 2
flush (2nd time)	**fence**
	e := X
	assert e == 2;



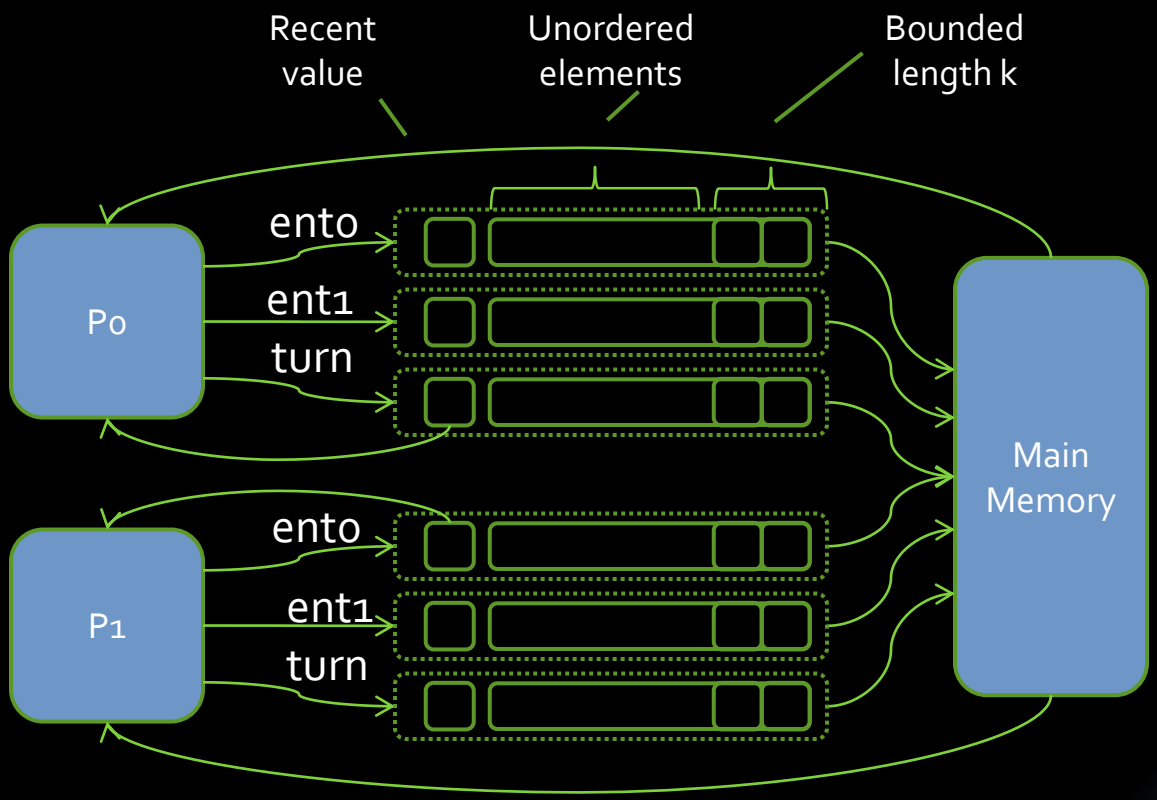
Abstract Memory Models - Requirements

- Intra-process coherence: a process should see the most recent value it wrote
- Preserve fence semantics
- Partial inter-process coherence: preserve as much order information as feasible (bounded)
  - Enable strong flushes
- Sound
- Simple construction!

Partial Coherence Abstractions



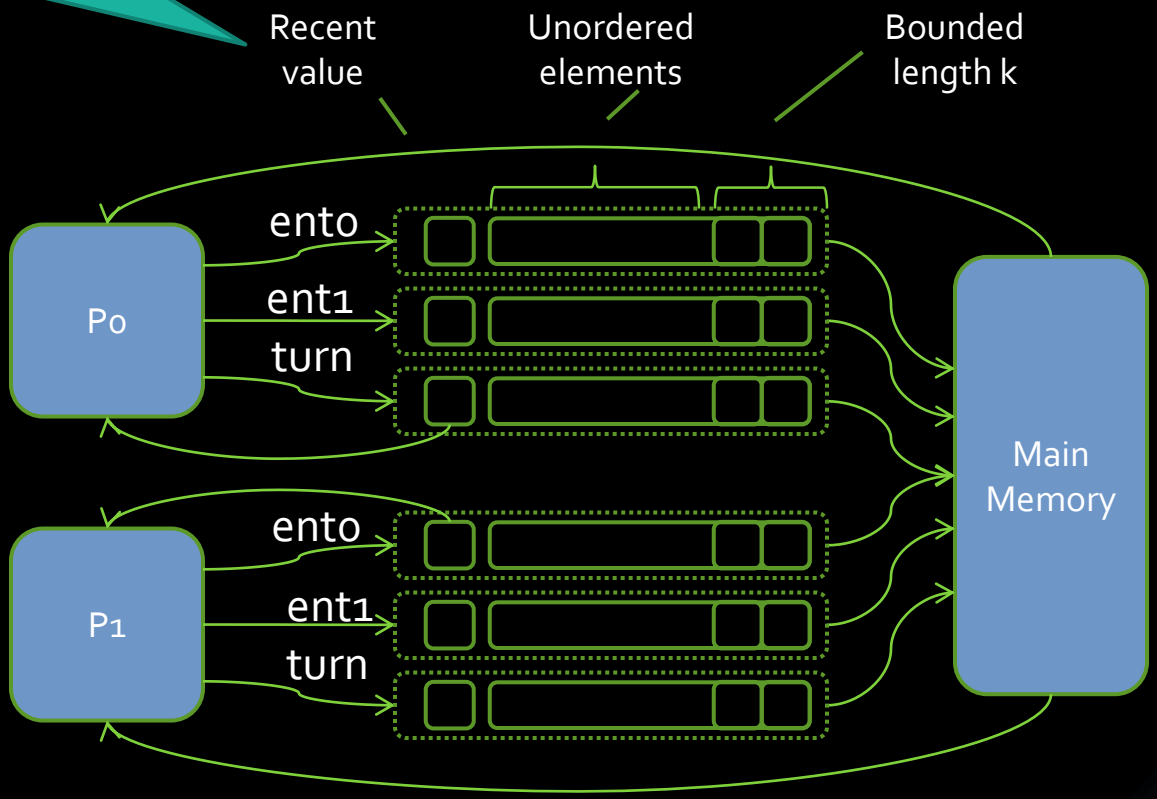
Partial Coherence Abstractions



Partial Coherence Abstractions

Allows precise fence semantics

Allows precise loads from buffer

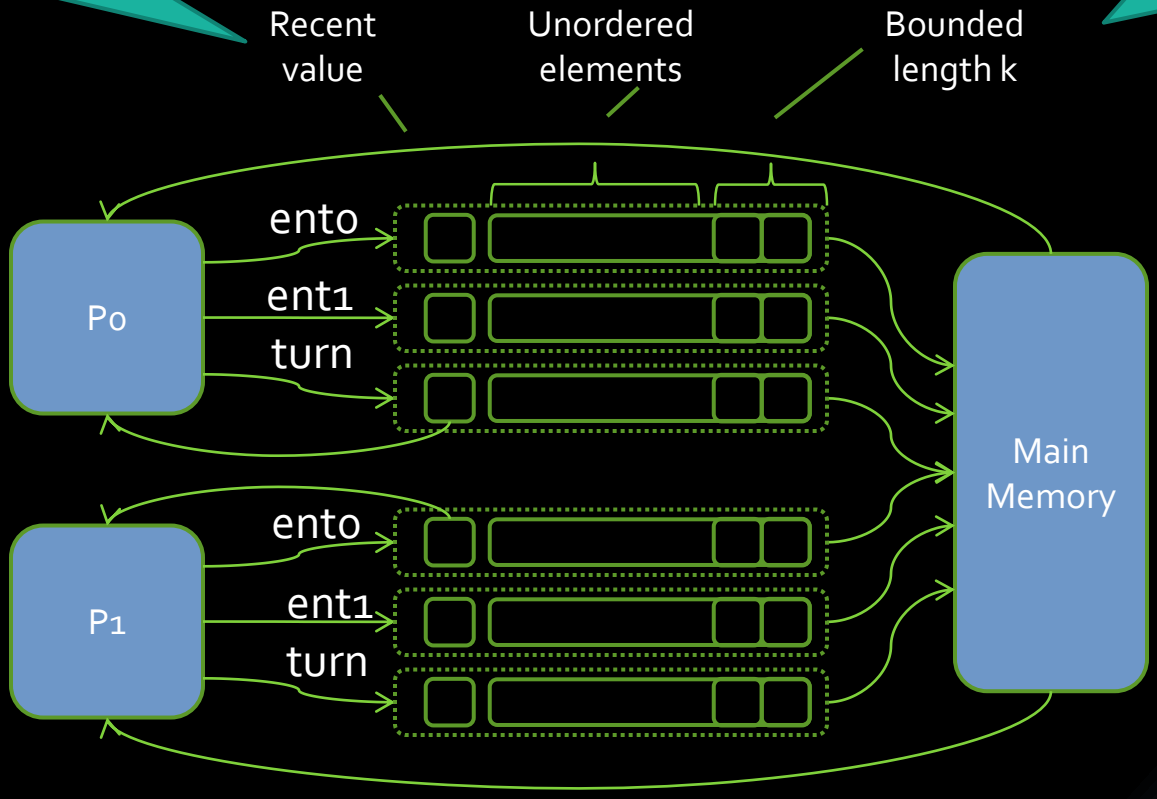


Partial Coherence Abstractions

Allows precise fence semantics

Allows precise loads from buffer

Keeps the analysis precise for "normal" programs



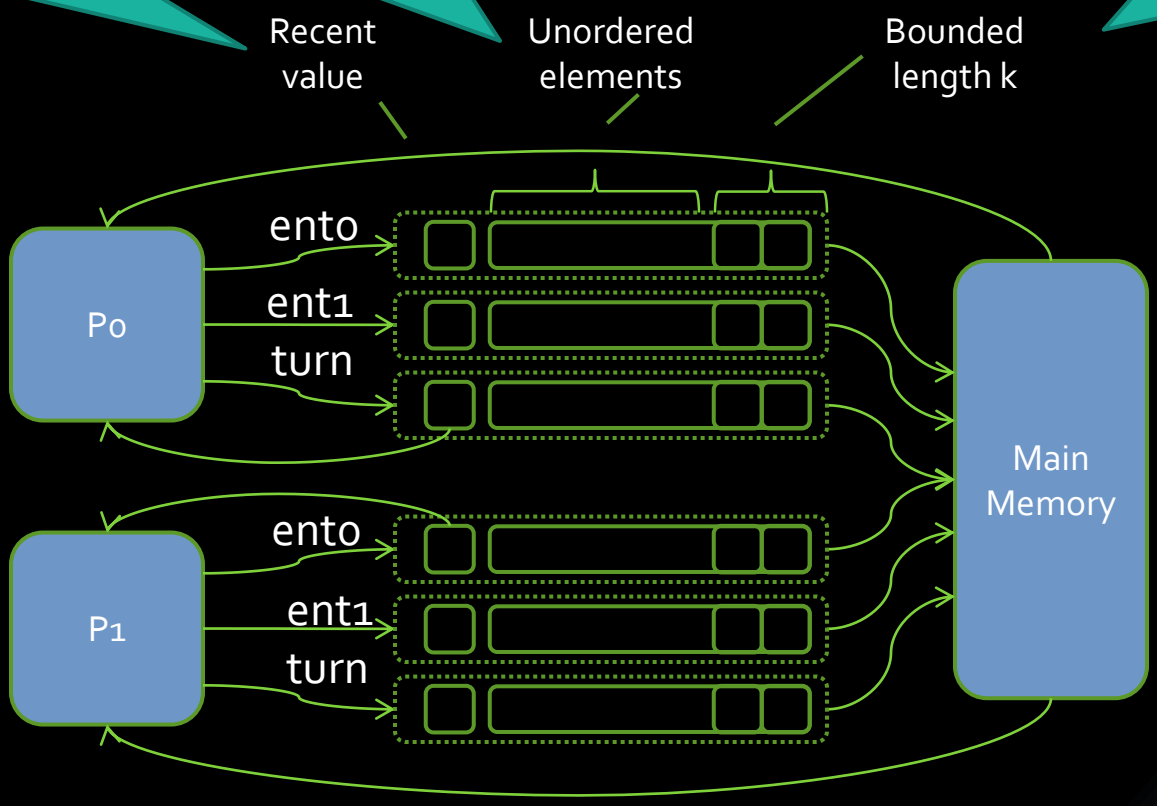
Partial Coherence Abstractions

Allows precise fence semantics

Allows precise loads from buffer

Fallback for soundness

Keeps the analysis precise for "normal" programs



Intuition

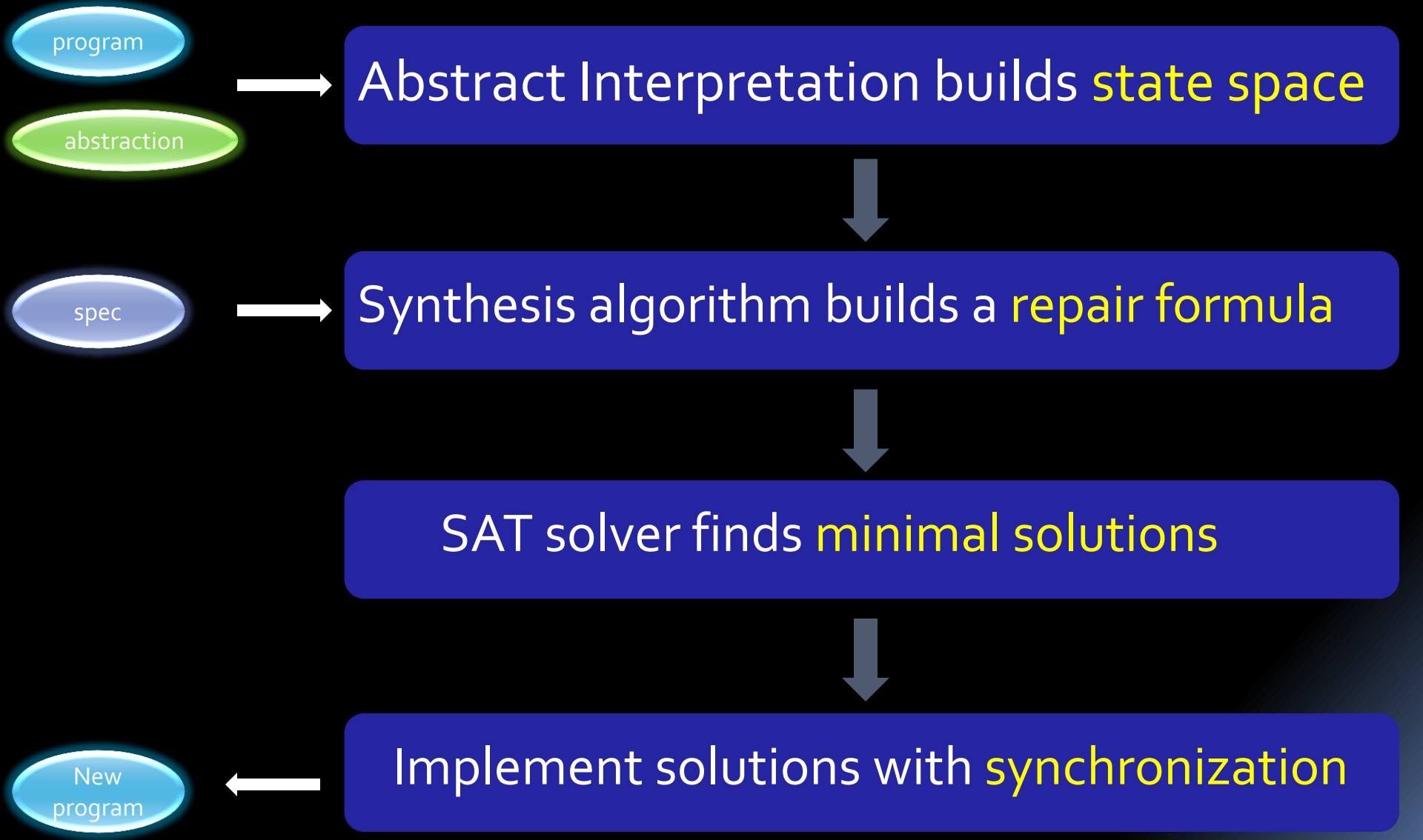
- Making unbounded number of writes to a memory location without a flush?
  - Seems very esoteric
  - Your program is suspicious

```
flag := true
while other_flag = true {
  flag := false
  //Do something
  flag := true
}
```

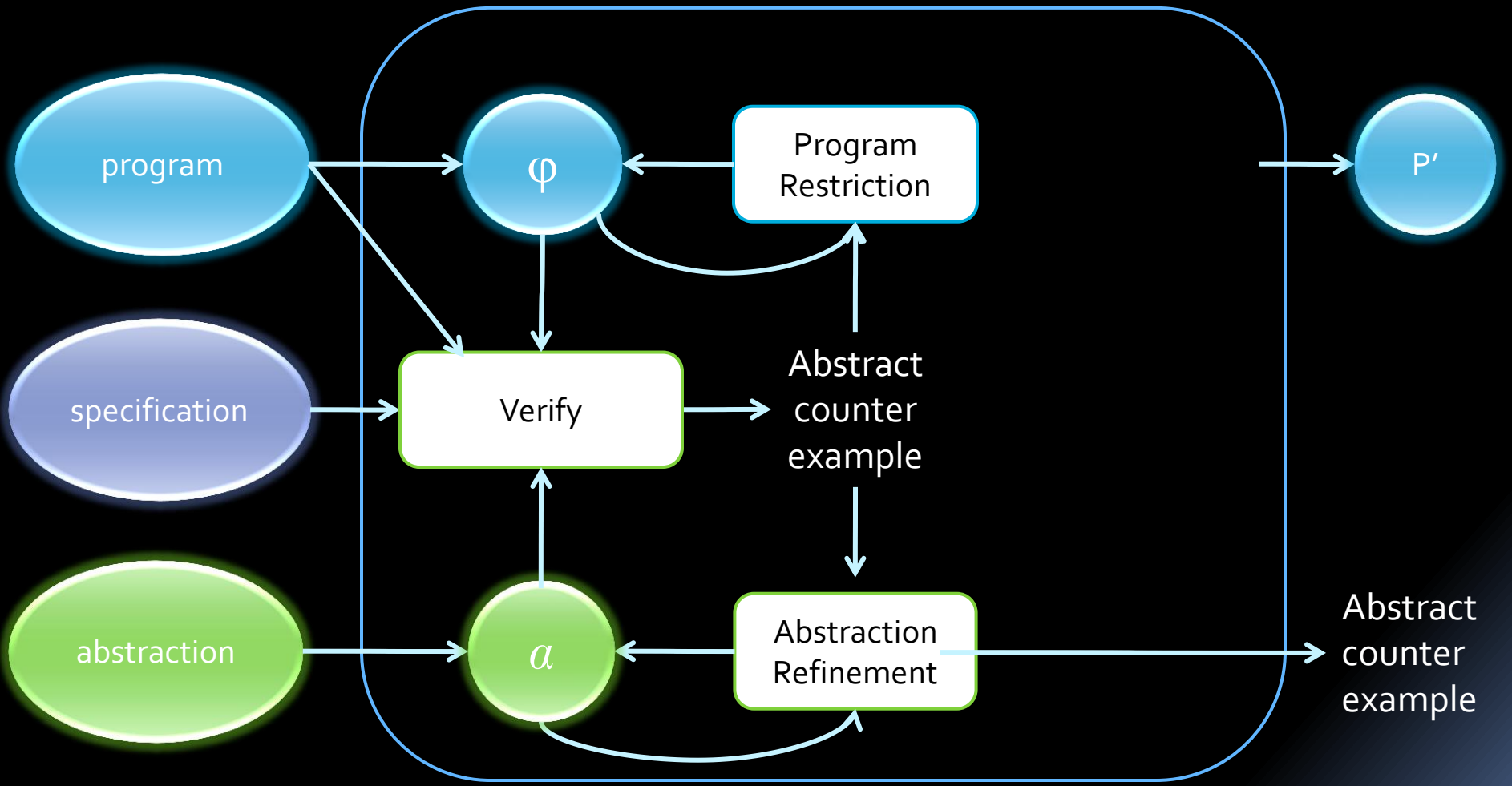
Adjusted Recipe

- Compute (abstract) reachable states for the program using partial-coherence abstraction
- Compute constraints on execution that guarantee that all "bad states" are avoided
- Implement the constraints with fences

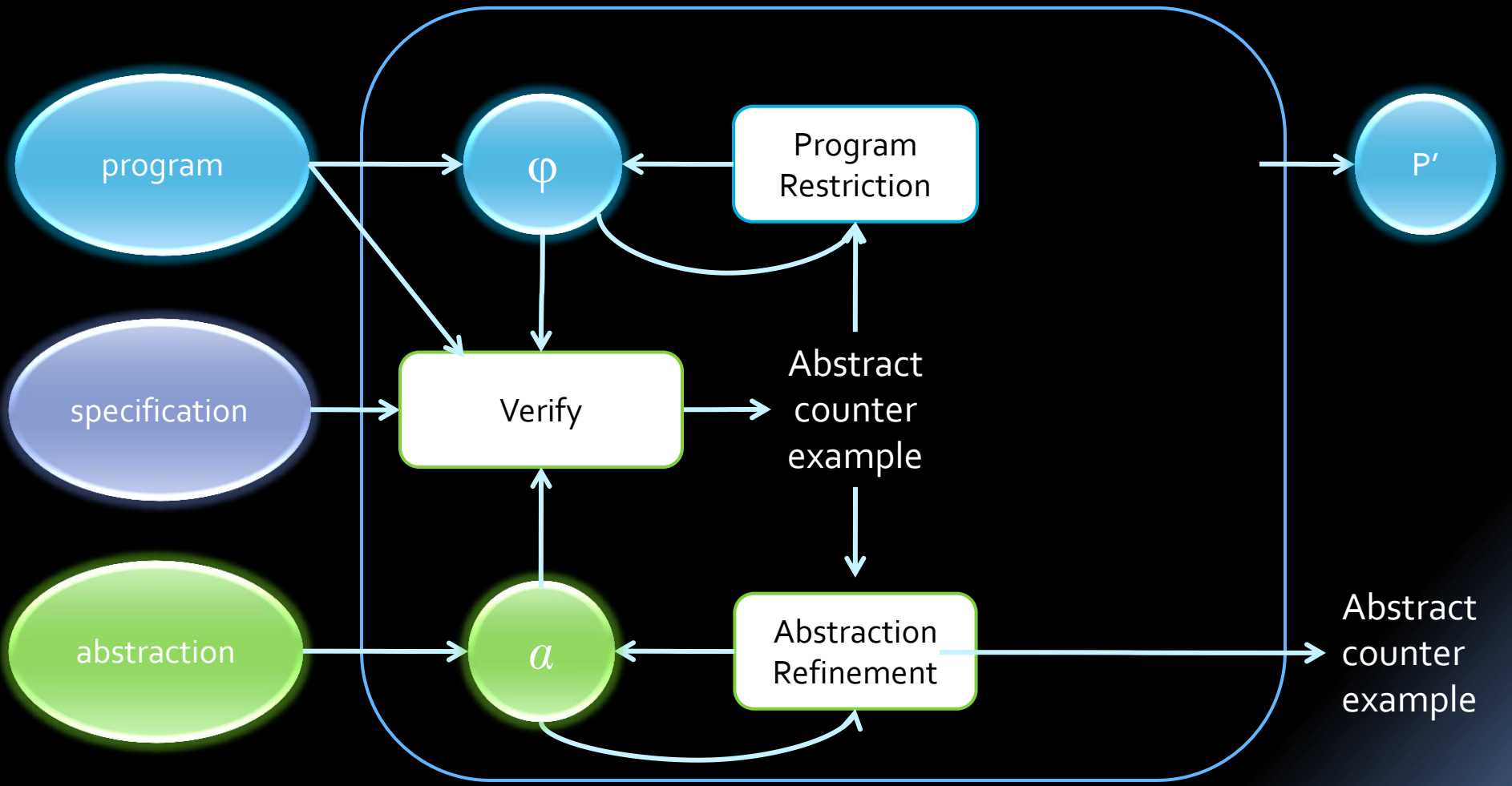
Flow of Synthesis



Thinking Roadmap

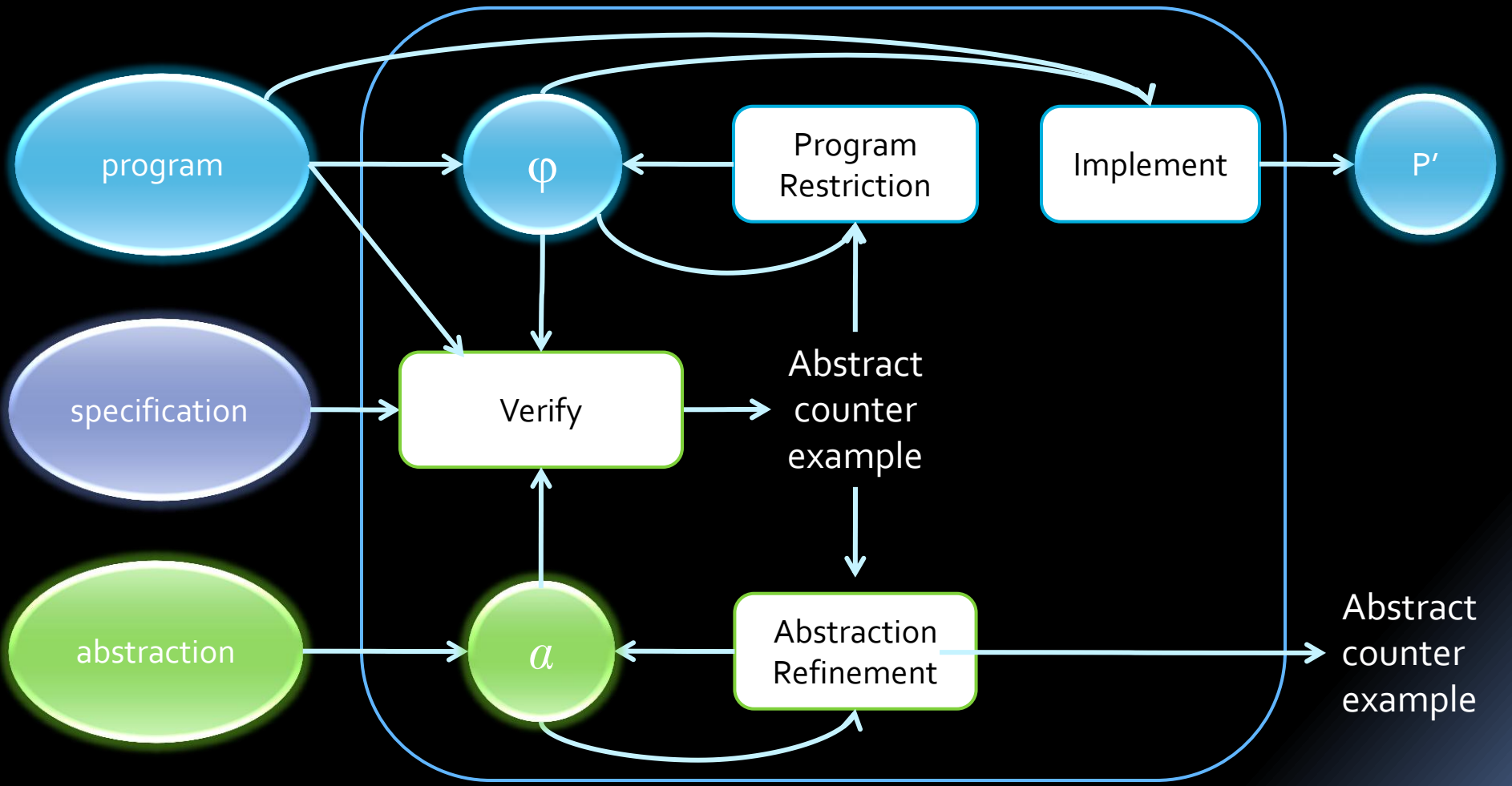


Thinking Roadmap



Change the **program** to match the **abstraction**

Thinking Roadmap



Change the **program** to match the **abstraction**

More information

- My group @ ETH Zurich: http://www.srl.inf.ethz.ch/
- Fender: http://practicalsynthesis.org/fender/
- Workshops:
  - PSY: http://practicalsynthesis.org/psy2012/
  - Dagstuhl: http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=12152

thanks for your attention

merci pour votre attention ☺