Software Reliability

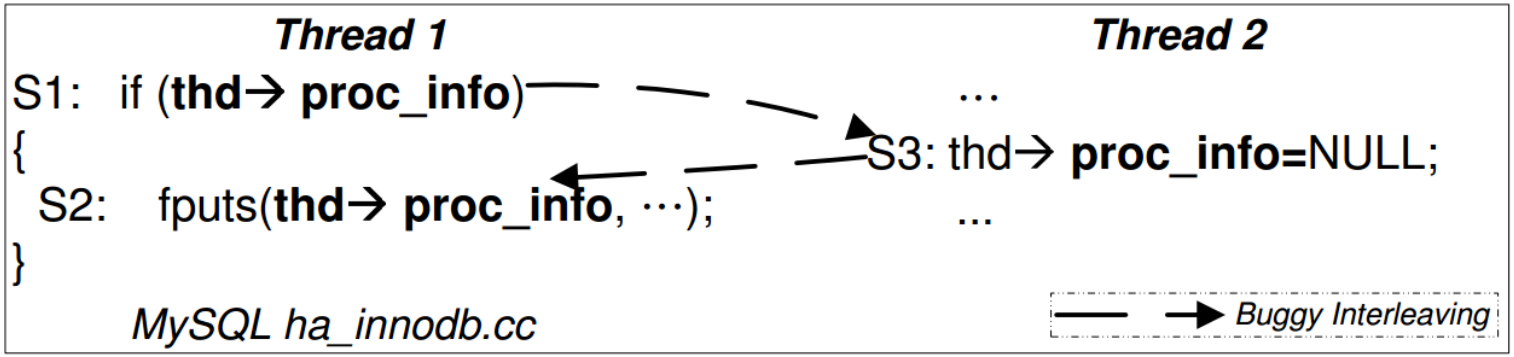
Lecture 10

Systematic Concurrency Testing

Alastair Donaldson
www.doc.ic.ac.uk/~afd

Thanks to **Paul Thomson** for the original lectures slides on which this is based

Motivation



An atomicity violation bug from MySQL.

- Concurrency bugs are:
  - crashes
  - assertion failures

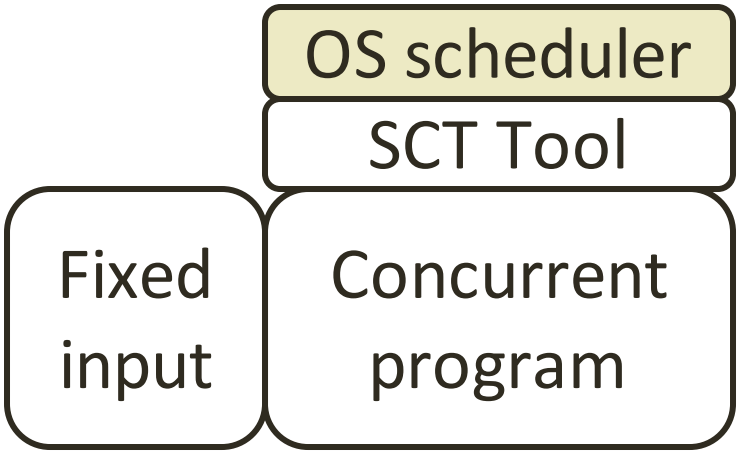
  that only manifest in a concurrent context, depending on **thread schedule**

Schedule [t1, t2, t1] exposes a concurrency bug here

Concurrency bugs are *horrible*

- May manifest rarely
- Hard to reproduce
- **Non-deterministic: “Heisenbugs”**

Systematic concurrency testing (SCT)



- Repeatedly execute the program to explore as many thread schedules as possible (all schedules, in the limit)

Required reading paper: Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, Iulian Neamtiu: *Finding and Reproducing Heisenbugs in Concurrent Programs*. OSDI 2008.

Systematic concurrency testing (SCT)

- Easy to apply to real programs
- No false-alarms
- Bugs are reproducible – they can be deterministically replayed

Simple example

- We shall illustrate the idea with a simple example, with two threads
- Class **Info** holds a reference to a **ProcInfo** class
- Method **`updateProcSize`** manipulates the **ProcInfo** object
- Method **`removeProcInfo`** sets the **ProcInfo** reference to null

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    if(procInfo != null) {
      int s = procInfo.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      procInfo.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Original code

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    if(procInfo != null) {
      int s = procInfo.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      procInfo.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Closer to what JVM executes

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

An interleaving

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;
pi = procInfo;
assert pi != null;
pi.size = s;
```

Thread 2 (removeProcInfo)

```
procInfo = null;
```

Another interleaving

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;
pi = procInfo;
assert pi != null;

pi.size = s;
```

Thread 2 (removeProcInfo)

```
procInfo = null;
```

And another

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Thread 1 (`updateProcSize`)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;
pi = procInfo;

assert pi != null;
pi.size = s;
```

Thread 2 (`removeProcInfo`)

```
procInfo = null;
```

A bad interleaving!

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;

pi = procInfo;
assert pi != null;
```

ERROR!

Thread 2 (removeProcInfo)

```
procInfo = null;
```

Another bad interleaving

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;

s = s / 2;
pi = procInfo;
assert pi != null;
```

ERROR!

Thread 2 (removeProcInfo)

```
procInfo = null;
```

A shorter, bad interleaving

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)

pi = procInfo;
assert pi != null;
```

ERROR!

Thread 2 (removeProcInfo)

```
procInfo = null;
```

A very short, good interleaving

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
```

Thread 2 (removeProcInfo)

```
procInfo = null;
```

SCT tool: implementation

- Insert callbacks into our multithreaded program.
  - Bytecode/binary instrumentation (runtime or offline)
  - Compile time instrumentation (clang pass)
  - Source code instrumentation

The method of one of our threads

```
public void updateProcSize() {
  ProcInfo pi = procInfo;
  if(pi != null) {
    pi = procInfo;
    assert pi != null;
    int s = pi.size;
    s = s * 2;
    s = s + 3;
    s = s / 2;
    pi = procInfo;
    assert pi != null;
    pi.size = s;
  }
}
```

The method after instrumentation

```
public void updateProcSize() {
  schedule();
  ProcInfo pi = procInfo;
  schedule();
  if(pi != null) {
    schedule();
    pi = procInfo;
    schedule();
    assert pi != null;
    schedule();
    int s = pi.size;
    schedule();
    s = s * 2;
    schedule();
    s = s + 3;
    schedule();
    s = s / 2;
    schedule();
    pi = procInfo;
    schedule();
    assert pi != null;
    schedule();
    pi.size = s;
  }
}
```

A call to **schedule** ensures that systematic search will consider each enabled thread at every scheduling point

A simple test harness

```
void main() {
  Info info = new Info();
  Thread t1 = new Thread( { info.updateProcSize(); } );
  Thread t2 = new Thread( { info.removeProcInfo(); } );

  t1.start();
  t2.start();

  t1.join();
  t2.join();
}
```

Systematic concurrency testing will explore every interleaving that can arise for this test case, considering thread switches at each **schedule** point

Improvements

- Partial-order reduction:
  - Skip many schedules without missing bugs
  - All terminal states will be explored
  - **Sound**
- Schedule bounding:
  - Explore only a subset of schedules so that many bugs will still be found
  - Bugs may be missed
  - **Unsound**

Simple Partial Order Reduction (POR): motivation

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;



i = procInfo;
assert pi != null;
(terminate)
```

Thread 2 (removeProcInfo)

```
procInfo = null;
int x = 0;
x = x + 1;
```

This schedule….

Simple Partial Order Reduction (POR): motivation

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;

s = s / 2;


i = procInfo;
assert pi != null;
(terminate)
```

Thread 2 (removeProcInfo)

```
procInfo = null;

int x = 0;
x = x + 1;
```

…is equivalent to this schedule

Simple Partial Order Reduction (POR): motivation

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;

s = s / 2;

i = procInfo;

assert pi != null;
(terminate)
```

Thread 2 (removeProcInfo)

```
procInfo = null;

int x = 0;

x = x + 1;
```

…and to this one

Simple Partial Order Reduction (POR): motivation

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;

s = s / 2;
i = procInfo;


assert pi != null;
(terminate)
```

Thread 2 (removeProcInfo)

```
procInfo = null;


int x = 0;
x = x + 1;
```

…and to this one…

Simple Partial Order Reduction (POR): motivation

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;

i = procInfo;


assert pi != null;
(terminate)
```

Thread 2 (removeProcInfo)

```
procInfo = null;

int x = 0;
x = x + 1;
```

…and also this one!

But…

Simple Partial Order Reduction (POR): motivation

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;
i = procInfo;



assert pi != null;
pi.size = s;
```

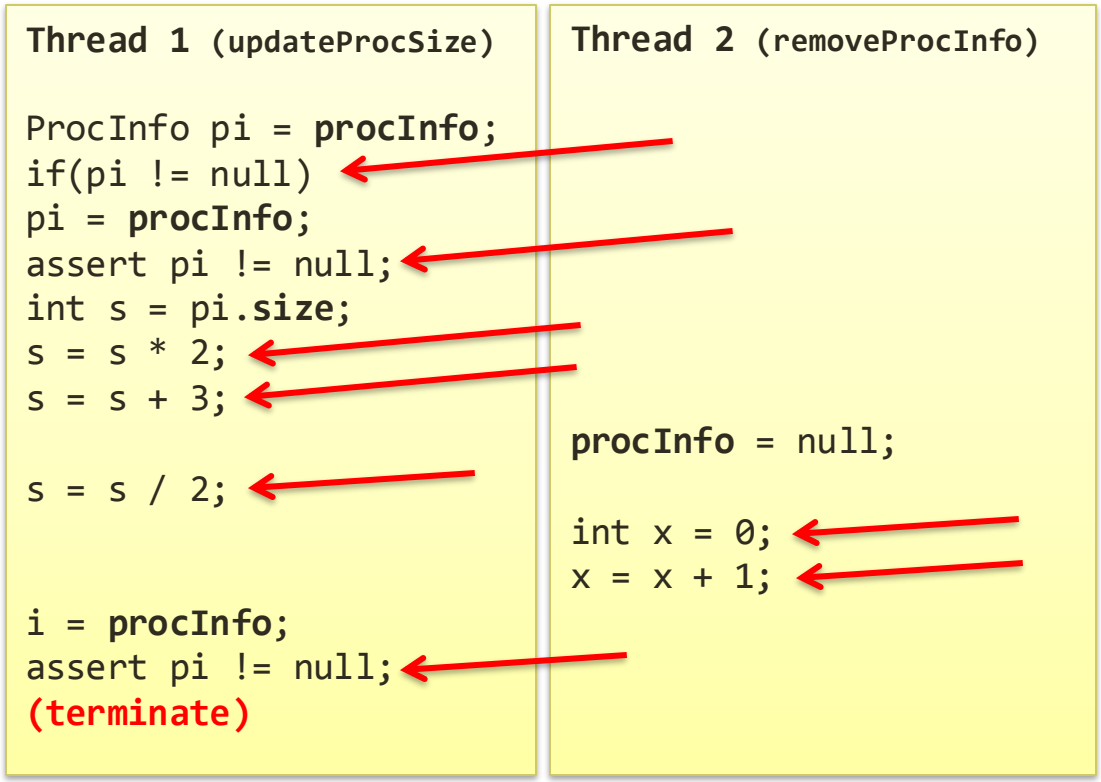
Thread 2 (removeProcInfo)

```
procInfo = null;
int x = 0;
x = x + 1;
```

…this schedule is not equivalent, because **procInfo** is a **shared variable**

Invisible operations

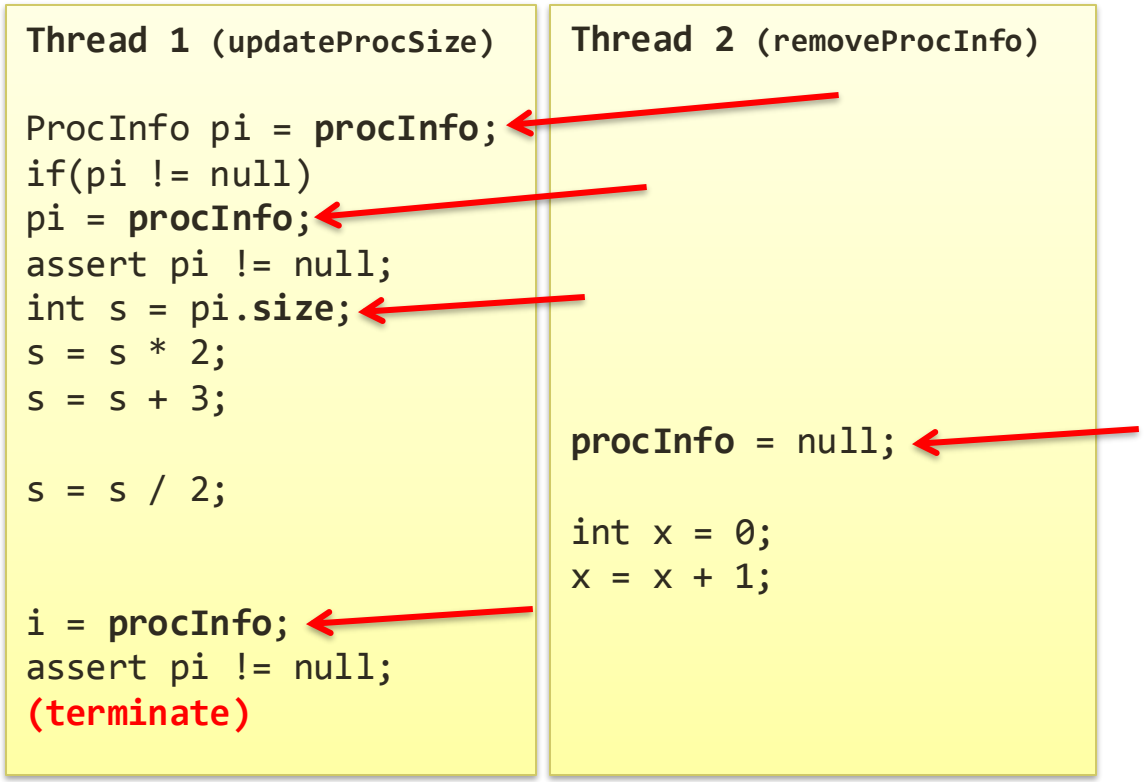
```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```



These are **invisible** operations: they only access thread-private state

Visible operations

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```



These are **visible** operations: they update shared state

Simple partial order reduction

Rule:

- If an adjacent pair of operations in different threads are swapped, where at least one is **invisible**, the resulting schedule is equivalent to the original

This is because the operations must access **disjoint data**

The rule can be applied many times to identify many redundant schedules

We don't want to explore these schedules

Merging visible and invisible operations

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;
```

```
i = procInfo;
assert pi != null;
```

(terminate)

Thread 2 (removeProcInfo)

```
procInfo = null;
int x = 0;
x = x + 1;
```

We treat a visible operation followed by a series of invisible operations like a **single** operation

Interleavings now considered at coarser level of granularity

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (`updateProcSize`)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;
i = procInfo;
assert pi != null;
pi.size = s;
```

Thread 2 (`removeProcInfo`)

```
procInfo = null;
int x = 0;
x = x + 1;
```

A schedule now describes a sequence of chunks

```
Schedule = [

t1,

t1,

t1,

t1,

t2,

t1

]
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
```

```
pi = procInfo;
assert pi != null;
```

```
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;
```

```
i = procInfo;
assert pi != null;
```

```
pi.size = s;
```

Thread 2 (removeProcInfo)

```
procInfo = null;
int x = 0;
x = x + 1;
```

To implement simple POR:

```
public void updateProcSize() {
  schedule();
  ProcInfo pi = procInfo;
  schedule();
  if(pi != null) {
    schedule();
    pi = procInfo;
    schedule();
    assert pi != null;
    schedule();
    int s = pi.size;
    schedule();
    s = s * 2;
    schedule();
    s = s + 3;
    schedule();
    s = s / 2;
    schedule();
    pi = procInfo;
    schedule();
    assert pi != null;
    schedule();
    pi.size = s;
  }
}
```

Instead of inserting a scheduling point before each operation…

To implement simple POR:

```
public void updateProcSize() {
  schedule();
  ProcInfo pi = procInfo;

  if(pi != null) {
    schedule();
    pi = procInfo;

    assert pi != null;
    schedule();
    int s = pi.size;

    s = s * 2;

    s = s + 3;

    s = s / 2;
    schedule();
    pi = procInfo;

    assert pi != null;
    schedule();
    pi.size = s;
  }
}
```

Only insert a scheduling point before **visible** operations

This simple POR is not optimal

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```

Thread 1 (updateProcSize)

```
ProcInfo pi = procInfo;
if(pi != null)
pi = procInfo;
assert pi != null;
int s = pi.size;
s = s * 2;
s = s + 3;
s = s / 2;
i = procInfo;
assert pi != null;
pi.size = s;
```

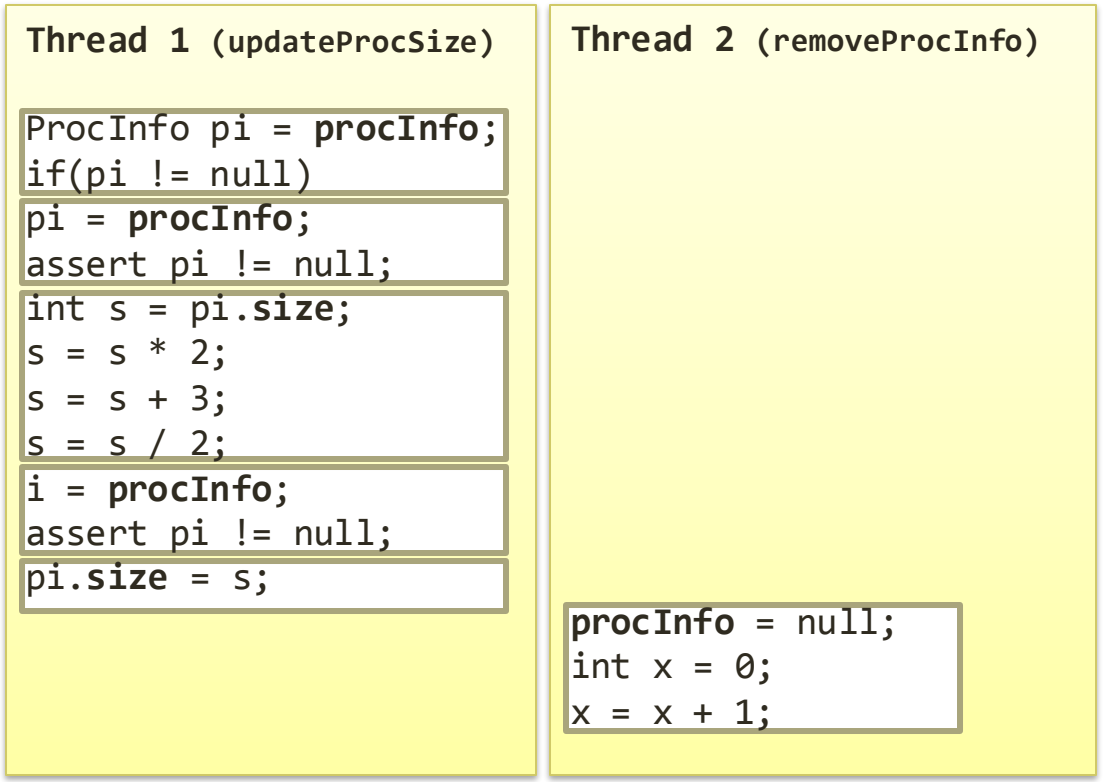
Thread 2 (removeProcInfo)

```
procInfo = null;
int x = 0;
x = x + 1;
```

This schedule is clearly equivalent to…

This simple POR is not optimal

```
public class Info {

  private ProcInfo procInfo;

  …

  public void updateProcSize() {
    ProcInfo pi = procInfo;
    if(pi != null) {
      pi = procInfo;
      assert pi != null;
      int s = pi.size;
      s = s * 2;
      s = s + 3;
      s = s / 2;
      pi = procInfo;
      assert pi != null;
      pi.size = s;
    }
  }

  public void removeProcInfo() {
    procInfo = null;
    int x = 0;
    x = x + 1;
  }

}
```



…this schedule!

Dynamic partial order reduction provides a more advanced algorithm (not covered here)

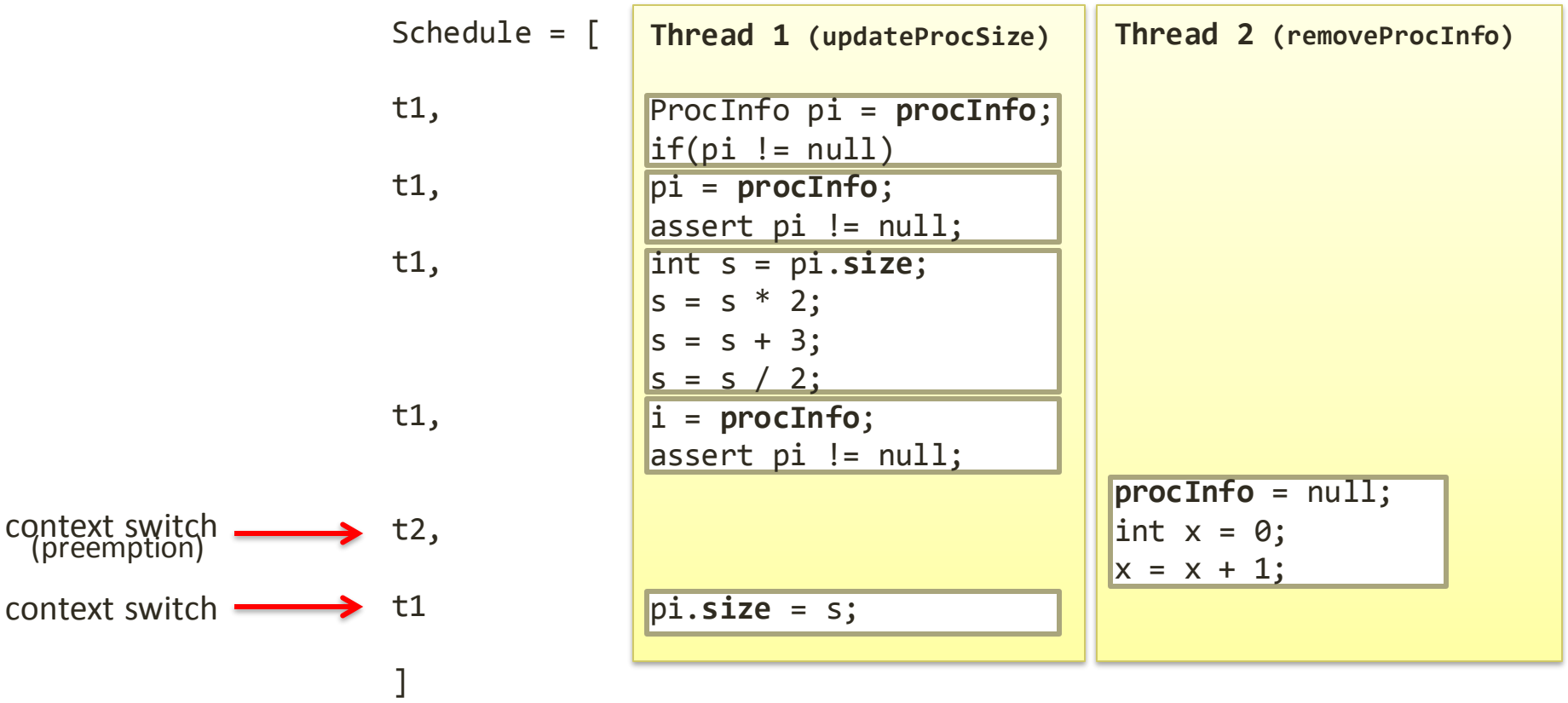
Schedule bounding

- Explore only a subset of schedules so that many bugs will still be found
- Bugs may be missed
- **Unsound**, but pragmatic

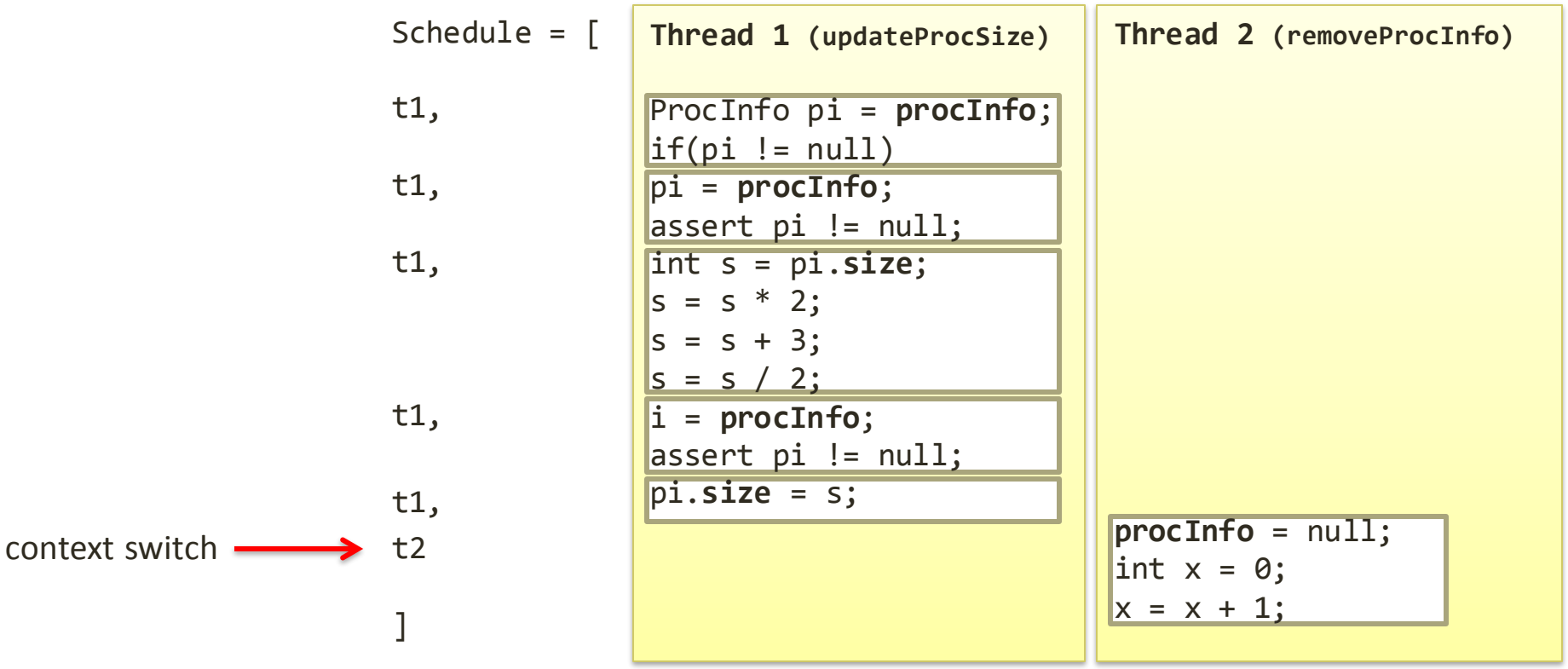
Context switches and preemptions

- A **context switch** occurs in a schedule whenever the executing thread changes
- A context switch is a **preemption** if the previously executing thread could have *continued* executing

A preemptive and non-preemptive context switch



This context swith is not preemptive…



…because Thread 1 had no more instructions

Preemption bounding

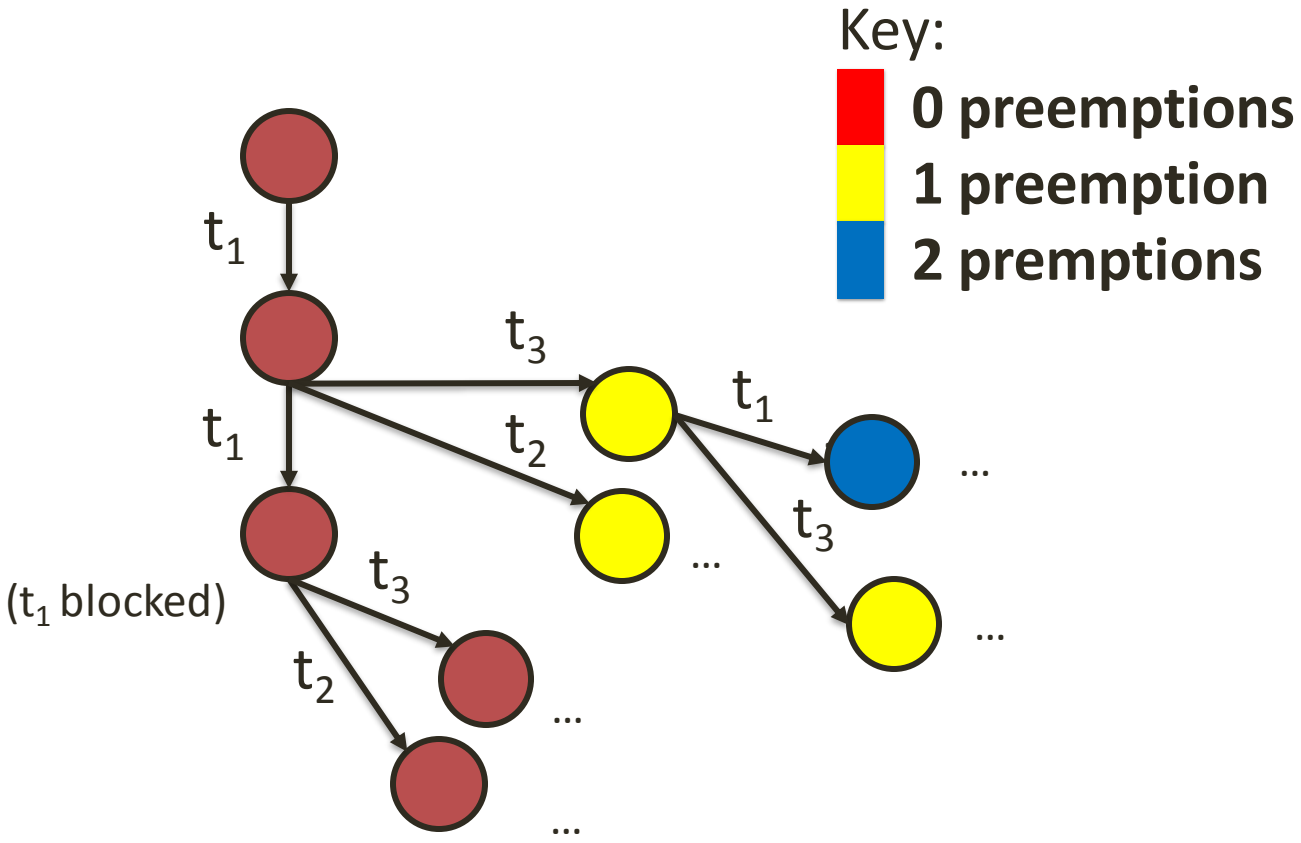
Explore all schedules that exhibit *d* preemptions or fewer, for some small *d*

Empirical evidence suggests that real-world concurrency bugs usually manifest under small preemption bounds

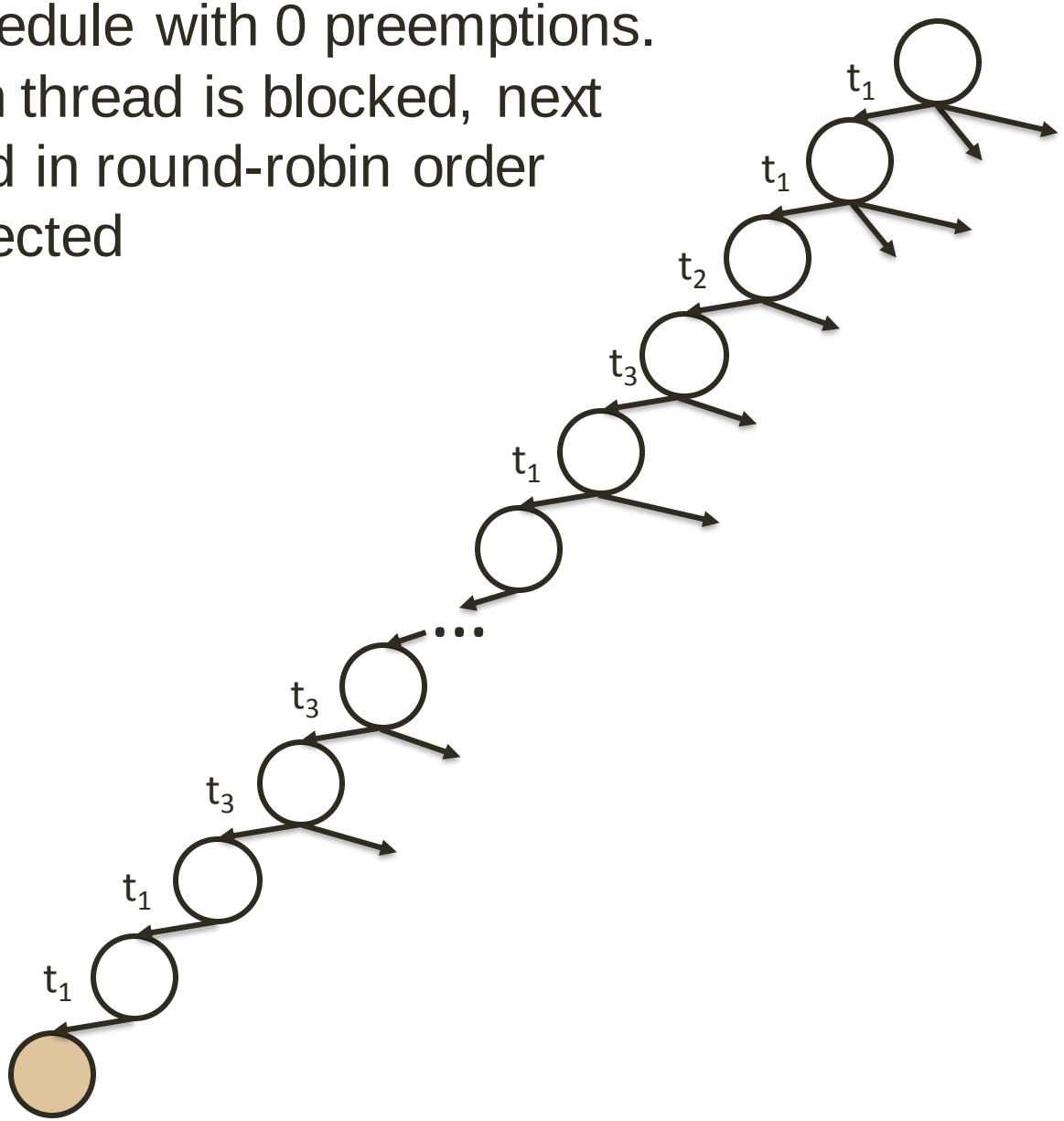
Easy to contrive a bug that **requires** e.g. 17 preemptions to occur…

…but most concurrency bugs that can manifest with many preemptions can **also** manifest with few preemptions

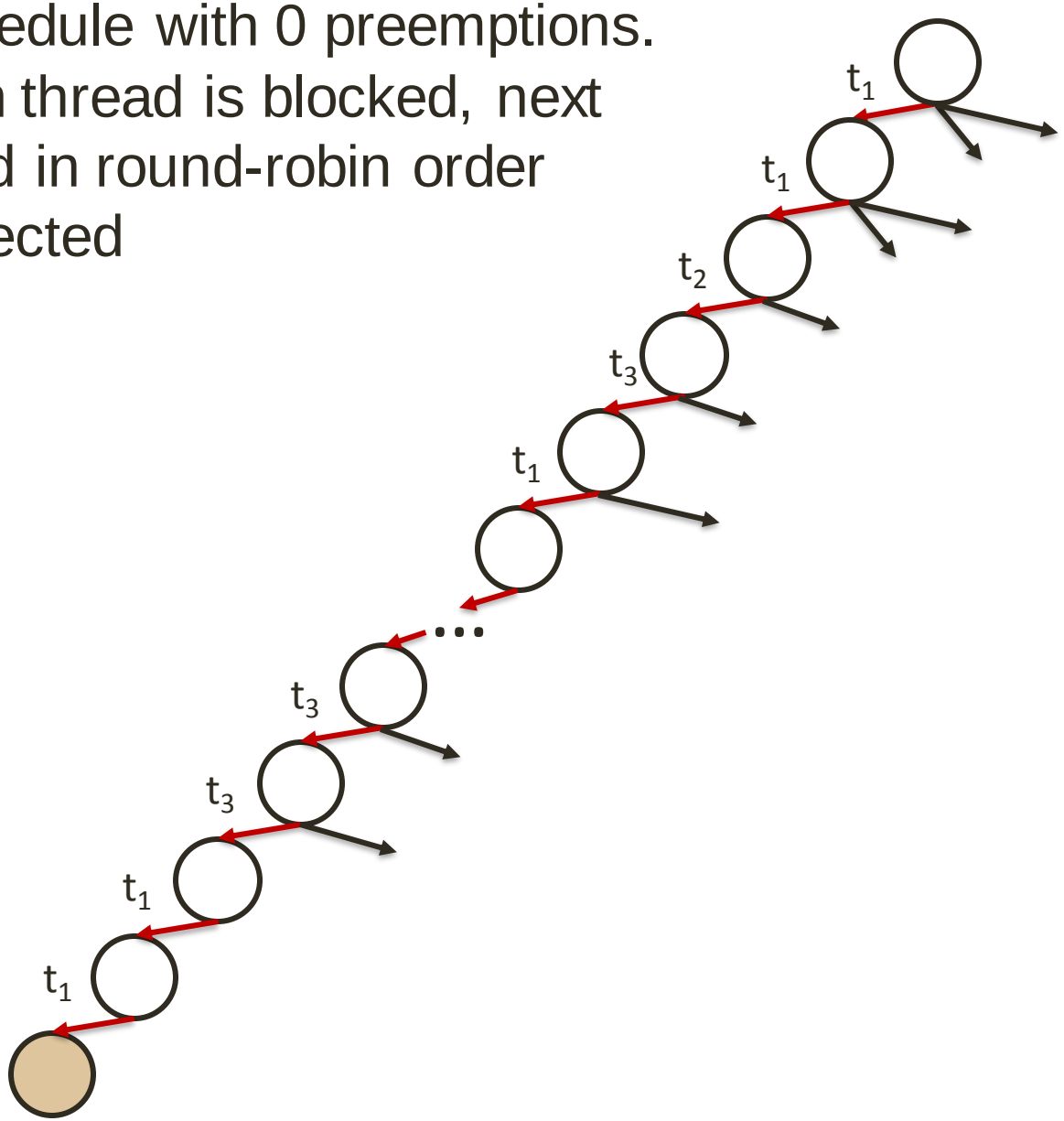
Preemption bounding



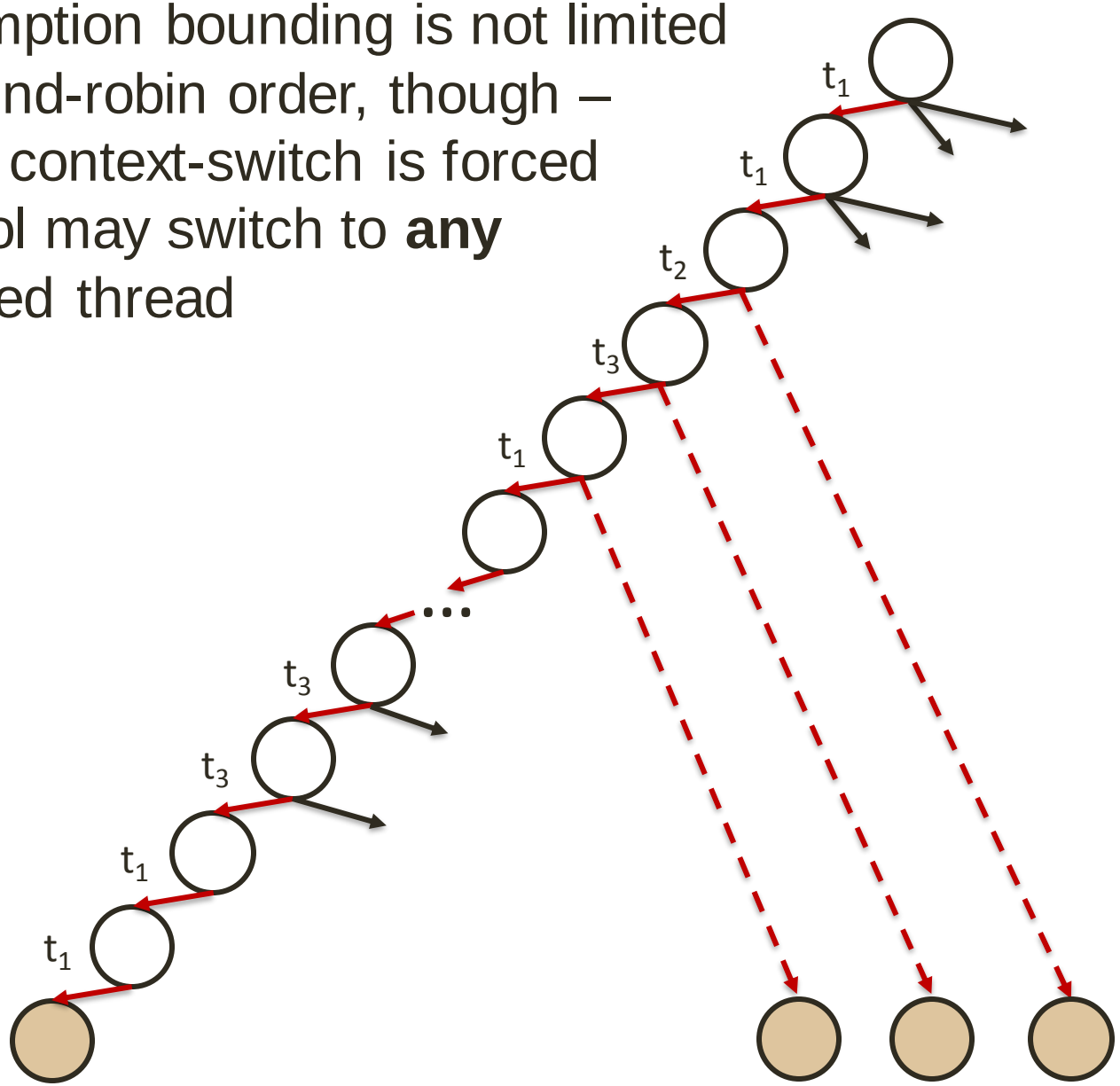
A schedule with 0 preemptions. When thread is blocked, next thread in round-robin order is selected



A schedule with 0 preemptions. When thread is blocked, next thread in round-robin order is selected

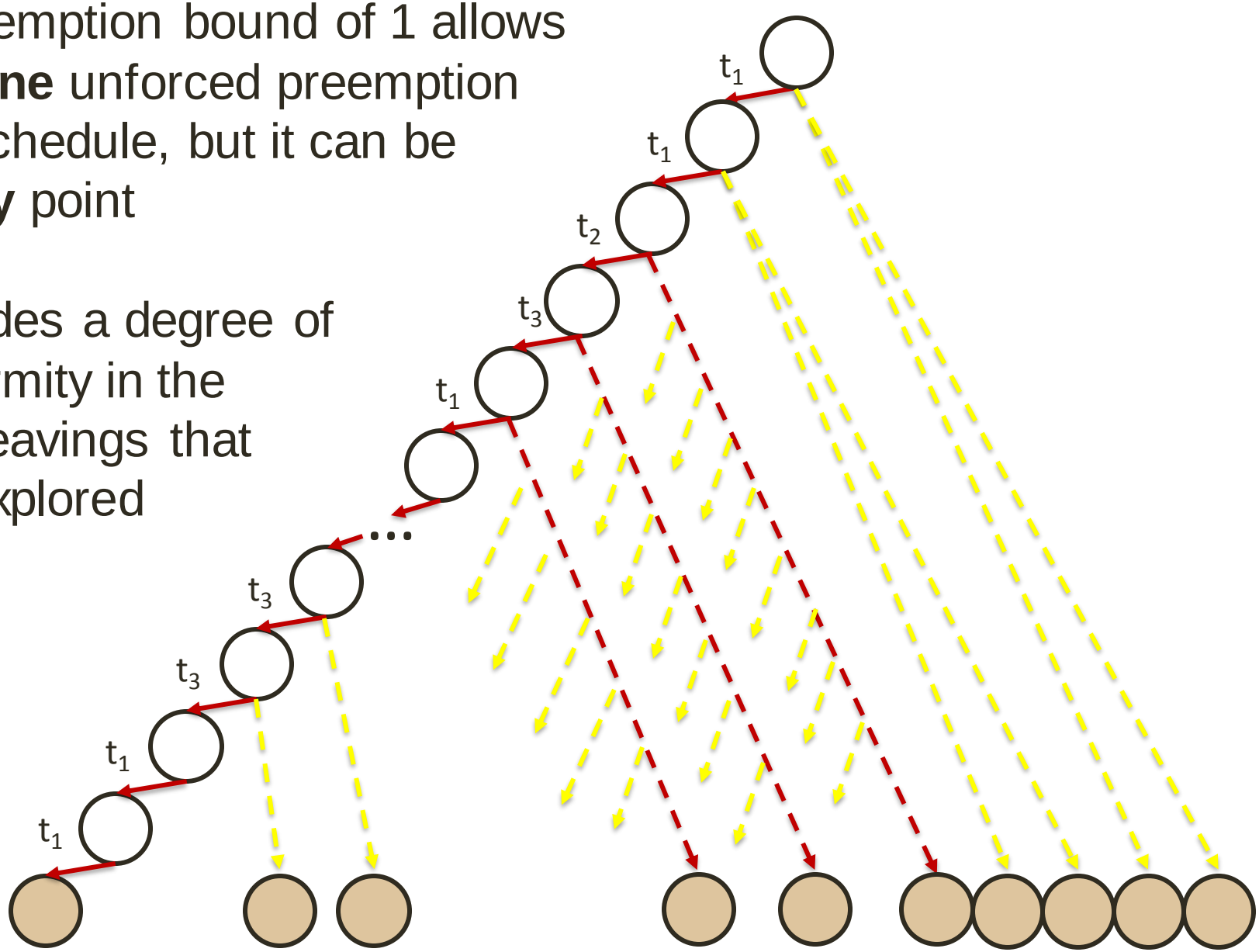


Preemption bounding is not limited to round-robin order, though – when context-switch is forced control may switch to **any** enabled thread

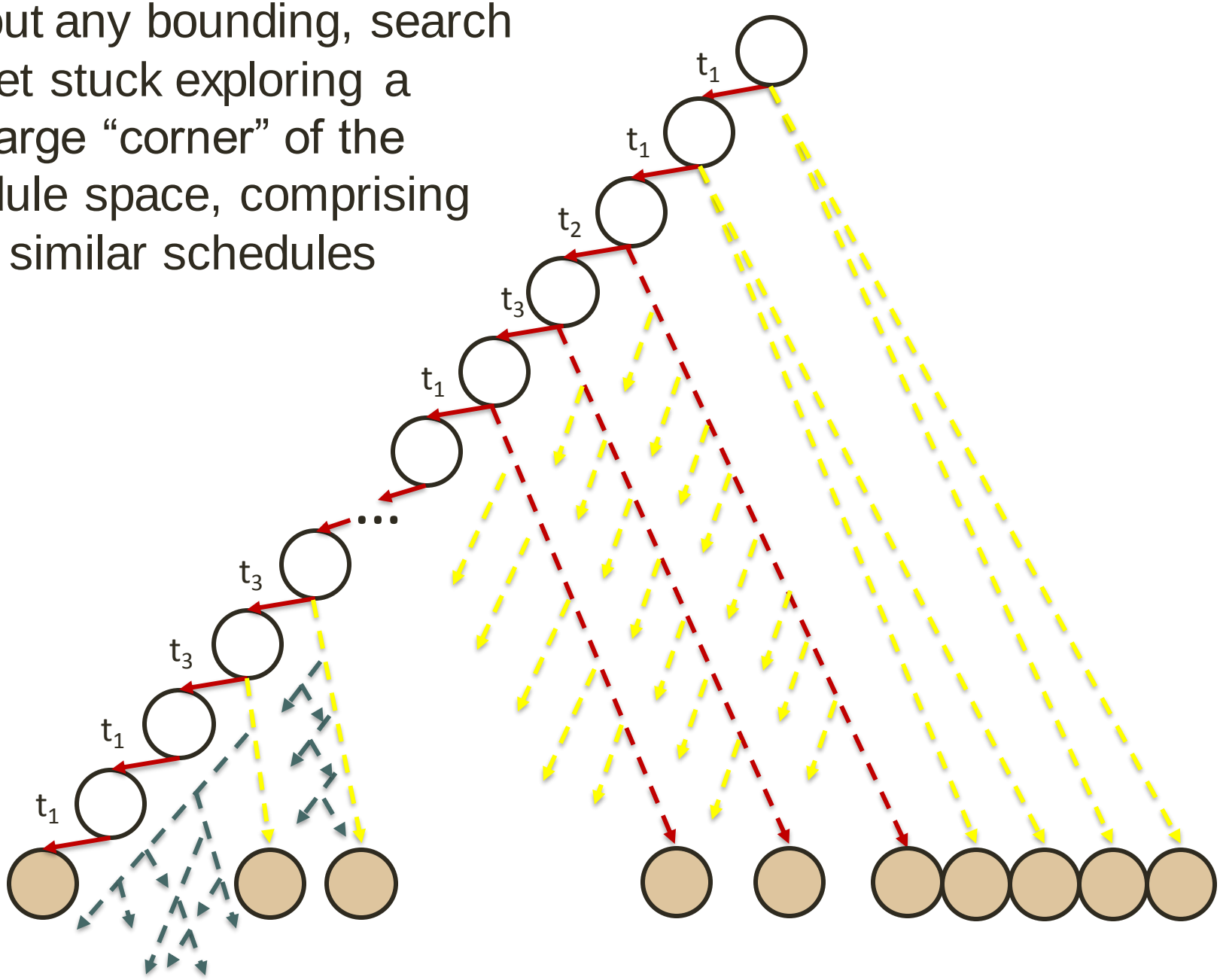


A preemption bound of 1 allows just **one** unforced preemption per schedule, but it can be at **any** point

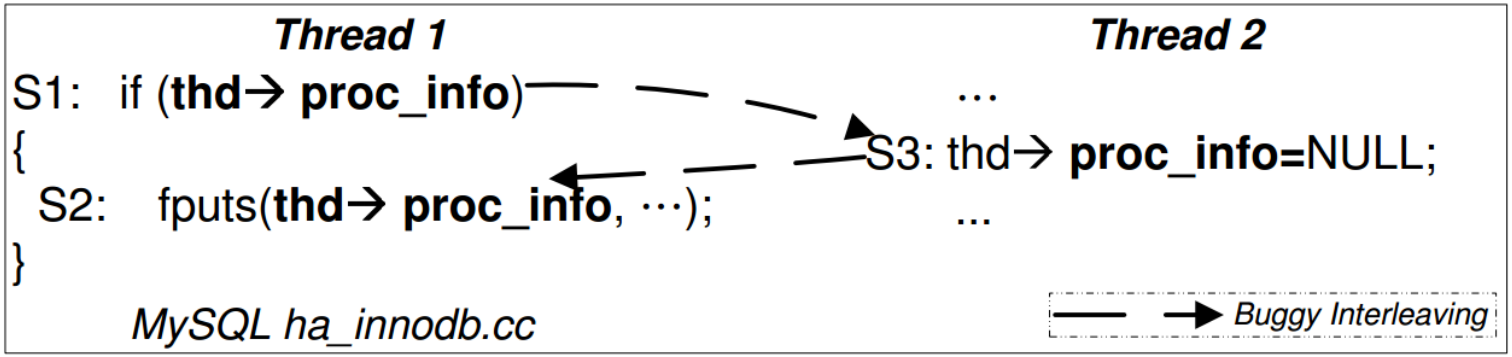
Provides a degree of uniformity in the interleavings that are explored



Without any bounding, search can get stuck exploring a very large “corner” of the schedule space, comprising many similar schedules



How many preemptions does this bug require?



An atomicity violation bug from MySQL.

Preemption bounding properties

1. Scales well (with more execution steps)
   - Does not scale well with more threads
2. Simple counter-example schedules
3. Bounded guarantees
   1. Missed bugs require more than k preemptions
   2. Missed bugs may be less likely to occur
4. **Low preemption bound => Many bugs found**
   - **(compared with depth-first search)**

Summary (but not the last slide!)

Systematic concurrency testing finds concurrency bugs automatically and allows deterministic replay

A concurrency test case is required

The schedule space can be enormous

Partial order reduction enables sound pruning

Schedule bounding restricts search – may cause bugs to be missed, but can find typical concurrency bugs effectively

However…

Concurrency testing using schedule bounding: an empirical study

Paper by Paul Thomson, me, and Adam Betts, at Principles and Practice of Parallel Programming 2014

- 52 open source multithreaded test cases
- Baseline was naïve depth-first search
- Schedule limit was 10,000
- DFS: found 33 bugs
- Preemption bounding: found 45 bugs
- **Random scheduler:** 45 bugs!

Open question: were the test cases representative?