CS 636: Testing Concurrent Programs

Swarnendu Biswas

Semester 2020-2021-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Evaluating your Concurrent Program

Check for correctness

- Atomicity violations, order violations, sequential consistency violations
- Deadlocks and livelocks

Check for performance and scalability

- Check whether all real-time requirements are met
- Check for any performance regressions

Possible Ideas to Ensure Correctness of Concurrent Programs

Programming language features

- Language ensures bad things cannot happen by design (e.g., DPJ)
- Restricts the power and expressiveness of the language

Resilient algorithms

- Design algorithms that are resilient to errors
- Limits the kind of data structures that you can use

Comprehensive testing

- Cannot guarantee correctness, usually a “best effort” strategy
- Places no restrictions on the application

Testing Concurrent Programs is Hard!

Nondeterminism is everywhere

- May be inherent in the application
- Can be due to inputs or interleavings
- Large space of all possible thread interleavings

Only specific thread interleavings may expose subtle errors – a concurrency bug

- Random or naïve testing can often miss such errors
- Often called "Heisenbugs"

Testing Concurrent Programs is Hard!

Even when found, errors are hard to debug

- Usually no repeatable trace, just retrying the execution may not reproduce the error if it is rare
- Debugging with `print()` statements may actually change the desired buggy interleaving
- Source of the bug may be far away from where it manifests

Huge productivity problem

- Developers and testers often spend weeks chasing after a single Heisenbug!

Testing Concurrent Programs

High-level steps

- Test code, test inputs, and test oracles – a test harness
- A deterministic schedule may be needed to validate with the oracles
- Associated notion of coverage – test as many interleavings as possible

Exhaustively explore all possible interleavings

Deterministic testing

- Controls thread scheduling decisions during execution and systematically explores interleavings
- Depends on a deterministic scheduler
- Nondeterminism could still be there due to inputs

Testing Concurrent Programs

Nondeterministic “best effort” testing

- Run the program for some time and hope for the best
- Naïve and inefficient

Stress testing

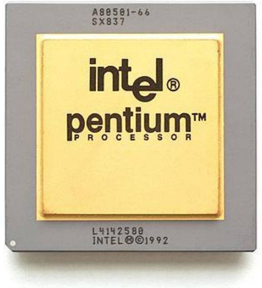
- Launch more threads than processors so that only a few threads are running at a time
- Try to decrease predictability in thread interleavings

Noise injection

- Introduce random perturbations during execution
- Should not introduce false positives

Alternatives to Testing

- Reason about correctness without running the program
  - Static analysis
  - Theorem proving
  - Model checking
- Try to prove programs correct
  - Requires a formal or mathematical characterization of the programs behavior
  - Very difficult for large systems since there are a lot of unknowns
    - For example, how do you model VM behavior like JIT compilation and GC?
  - Use is often limited to safety-critical software like integrated circuit design



Possible Approaches to Testing

- Model checking – Check whether a system model satisfies the given specification
  - Suffers from state explosion problem
  - Use partial order reduction to deal with the state space problem
  - Use is limited to only critical portions of the program
- Sophisticated static analysis and model checking do not scale well
- Dynamic analysis
  - User-defined events and properties that need to hold
  - Only verifies the current schedule that is being executed

Address Nondeterminism

- Enforce the correct schedule that needs to be executed
  - Deterministic execution: record and replay
- Explore all possible schedules
  - Stateful exploration
    - Model the program state at each step and use backtrack and state comparison to explore new schedules
    - Advantage is it can merge same states, alleviating the state space explosion problem
    - Java PathFinder is the state-of-art tool
  - Stateless exploration
    - Does not maintain program state
    - Each schedule maintains all the choices made during execution
    - Need to start from the beginning to execute other schedules
    - Each run is faster than stateful exploration, but possibly has more schedules to explore

Software Testing vs Concurrency Testing

Software Testing	Concurrency Testing

• Broad area of work which considers the overall quality of the software along with the integrated engineering processes
  • Lots of paradigms, processes, testing levels
	• The context that we will be discussing has more narrow focus
	  • Try to improve bug detection coverage of concurrent programs
	  • Mostly carried out by the developers themselves during unit testing

Software Testing vs Concurrency Testing

Software Testing	Concurrency Testing

- Bro… cor… the… int… pro…
  - Lots of paradigms, processes, testing levels

- …be …ow
- …tion …rograms
  - Mostly carried out by the developers themselves during unit testing

- A concurrency bug manifests on a strict subset of possible schedules
  - Bugs that manifest in all schedules are not concurrency bugs
- The problem of concurrency testing is to find those schedules that can trigger these bugs

Concurrency Testing Tools

- Java PathFinder (JPF) by NASA Ames Research Center
  - Model checking of concurrent programs
- Concutest – concurrency aware version of JUnit (concJUnit)
- ConTest – test concurrent Java programs by IBM Research Labs Haifa
- FindBugs – static analysis tool for Java
- Chess – Microsoft Research

Current Practice

- Concurrency testing is delegated to Random testing and Stress testing
- Example: Test a concurrent queue implementation
  - Create 100 threads performing queue operations
  - Run for days
  - Randomly perturb the execution
- Stress increases the likelihood of rare interleavings
  - Makes any error found hard to debug

Performance Testing

- No good tools for predicting system performance
  - Check for latency, resource consumption
- Other considerations
  - Garbage Collection (GC) may take arbitrarily long and may be triggered at random points
    - Either turn off GC or design tests that invoke multiple GCs so that it can be averaged out
  - Dynamic compilation with JIT compiler
    - Methods compiled and time taken impacts the measured time of the program
    - Mixing interpretation and JIT is random
    - Fix which methods are going to be compiled beforehand and only compile those at runtime

Directions

- Techniques to expose concurrency bugs
- Techniques to generate test cases (inputs) to trigger concurrency bugs
- Technique to automatically fix concurrency bugs
- …

Find Concurrency Bugs in Java based on Code Patterns

D. Hovemeyer and W. Pugh. Finding Concurrency Bugs in Java. PODC Workshop on Concurrency and Synchronization in Java Programs, 2004.

Insights Related to Concurrency Bugs

Programmers tend to make simple mistakes

- Tend to think sequentially
- Misconceptions about shared-memory synchronization

Synchronization is slow

- This is a myth
- Lots of research to optimize the common case of low contention
- Natural tendency is to under-synchronize

Indirect influence of the language

- Writing threaded code with Java is easy
- Java gives some guarantees with improperly synchronized code
  - You get type and memory safety, so why bother!!!

Overview of FindBugs

- Goal is to use simple analysis to find common patterns that indicate errors
  - Similar in spirit to automated code reviews
- As such there can be both false negatives and false positives
- Tries to minimize false positives and not to eliminate them completely
  - Uses heuristics to prune false positives
- New version of FindBugs is called SpotBugs

https://spotbugs.github.io/

Design of FindBugs

- Static, open-source Java bytecode analyzer
  - Implemented using BCEL and ASM
- Error reports
  - Potential errors are classified into levels depending on estimated impact
    - scariest - 1-4
    - scary - 5-9
    - troubling - 10-14
    - of concern - 15-20
  - There is also a notion of confidence along with each reported error
- Lot of plugins are available for tools like Eclipse, IntelliJ, Ant, and Maven

Patterns Used in FindBugs

- All accesses to fields of a thread-safe class should be guarded with locks
  - Otherwise reported as bug
  - Reduce false positives – ignore accesses in constructors and finalizers
  - Ignore volatiles, final fields, non-final public fields
- Rank reports based on access frequency
  - 25% or fewer unsynchronized accesses is classified as medium to high priority
  - 25-50% unsynchronized accesses are classified as low priority

Patterns Used in FindBugs

Synchronized set method, unsynchronized get method

Finalizer method only nulling out fields

Object pair operations with lock on only one object

- `equals()` method

Double-checked locking

- `ifnull` → `monitorenter` → `ifnull`

```
static SomeClass field;

static SomeClass createSingleton() {
  if (field == null)
    synchronized (lock) {
      if (field == null) {
        SomeClass obj = new SomeClass();
        // initialize obj
        field = obj;
      }
    }
  return field;
}
```

Patterns Used in FindBugs

Unconditional wait

Wait and notify without holding lock on the object

- Intraprocedural analysis to identify lock scopes

Two locks held while waiting

- Intraprocedural analysis to identify lock scopes

Spin wait on non-volatile data

If overriding `equals()`, then `hashcode()` should be overridden too

```
if (!book.isReady()) {
  synchronized (book) {
    book.wait();
  }
}
```

```
while (listLock) {}
listLock = true;
```

Relevance of FindBugs

- An early work (~2004) that was very effective in pointing out errors in real applications like the Java libraries
  - Implementation is still being actively maintained

From Eclipse 3.5RC3:

```
org.eclipse.update.internal.ui.views.FeatureStateAction:

if (adapters == null && adapters.length == 0)
  return;
```

- First seen in Eclipse 3.2
- In practice, `adapters` is probably never null

Relevance of FindBugs

- An early work (~2004) that was very effective in pointing out errors in real applications like the Java libraries
  - Implementation is still being actively maintained

```
if (listeners == null)
  listeners.remove(listener);
```

- JDK1.6.0 b105: `sun.awt.x11.XMSelection`

```
public WebSpider() {
  WebSpider w = new WebSpider();
}
```

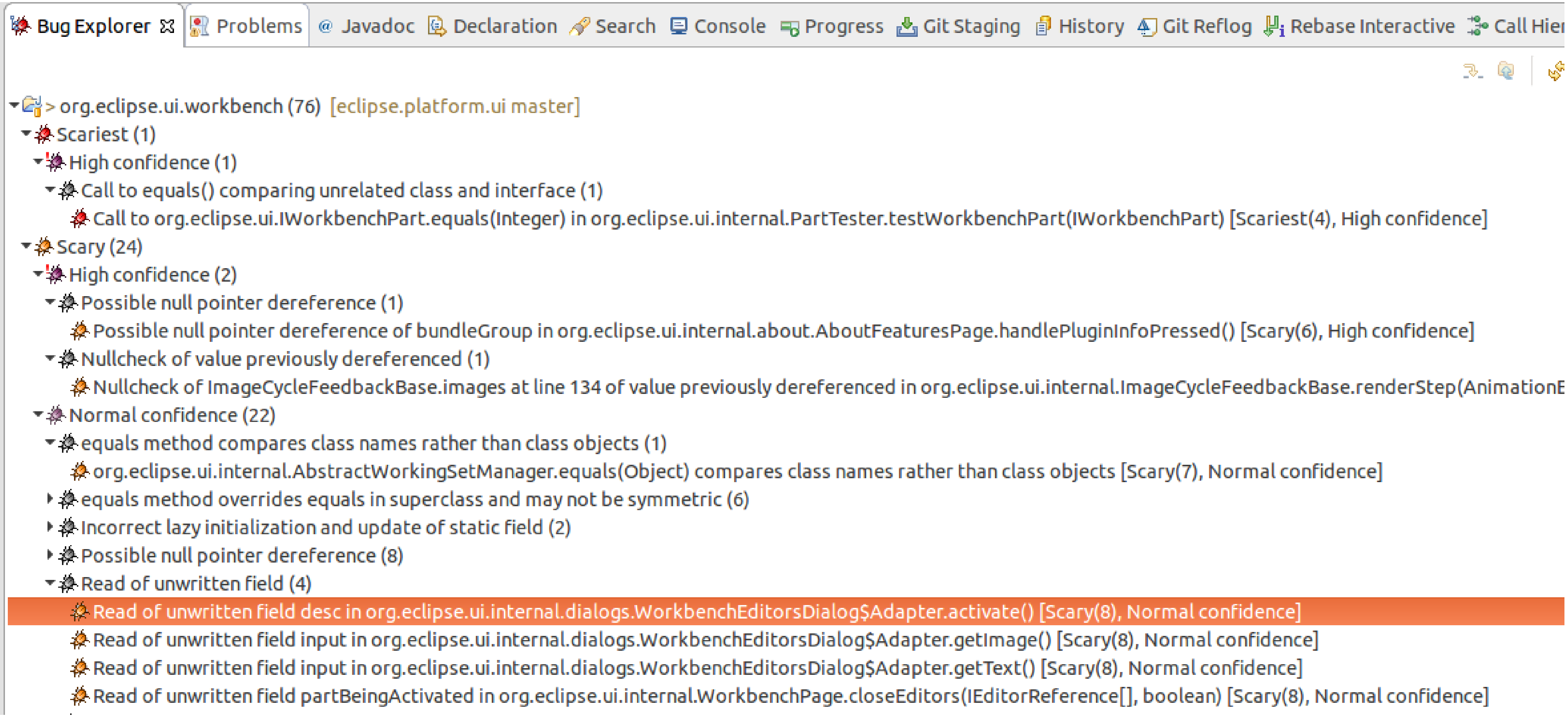
```
if (name != null || name.length > 0)
```

Design of FindBugs

- Over 400 bug patterns divided into different categories
  - **Correctness**
    - Infinite recursive loop, reads a field that is never written
  - **Multithreaded correctness**
  - **Bad practice**
    - Code that drops exceptions or fails to close file
  - **Performance**
    - Finalizers that set fields to null
  - **Dodgy** - code can lead to errors
    - Unused local variables or unchecked casts
  - …

https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html

FindBugs at work – Eclipse Plugin

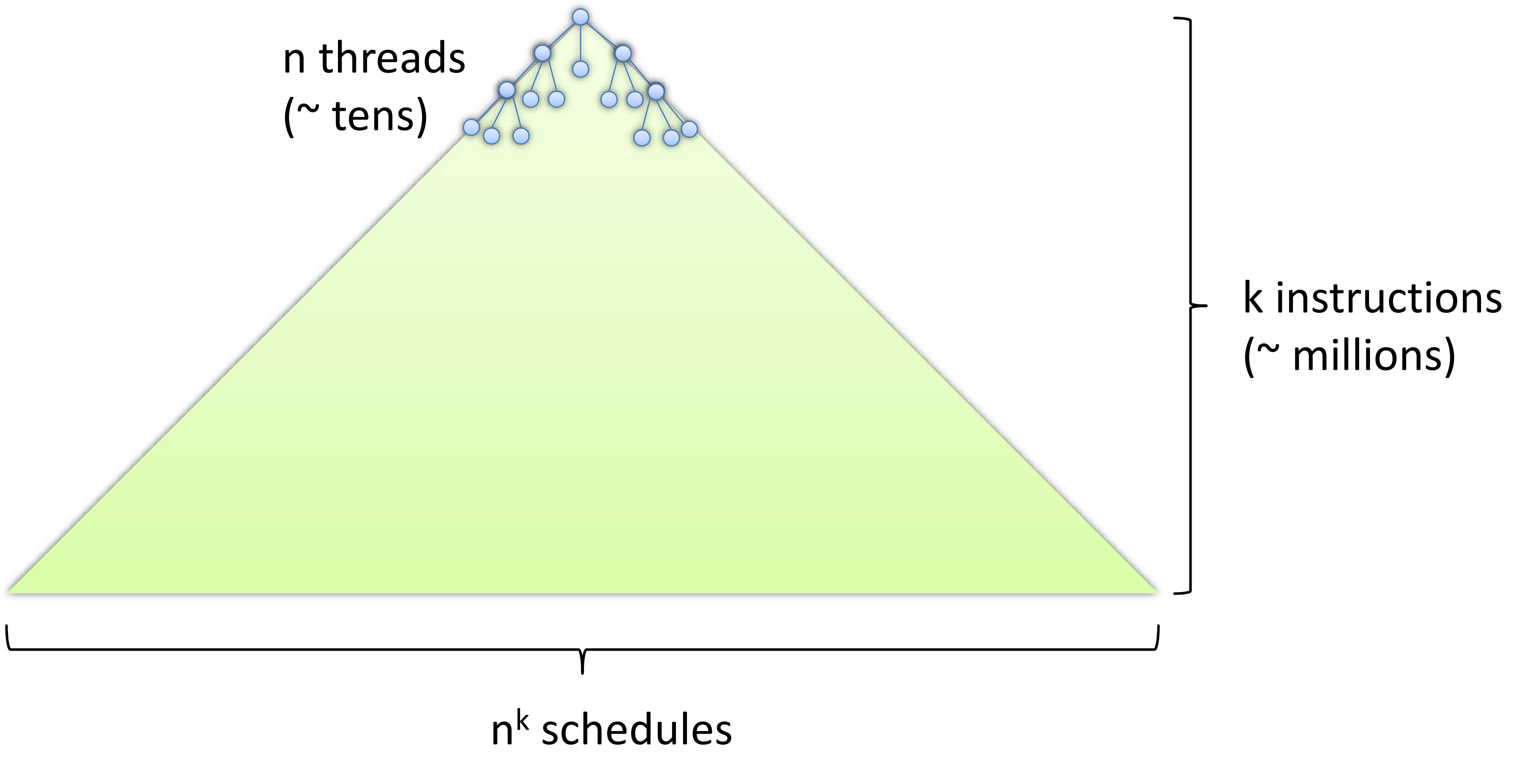


Probabilistic Concurrency Testing

S. Burckhardt et al. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS, 2010.
S. Nagarkatte et al. Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection. PLDI, 2012.

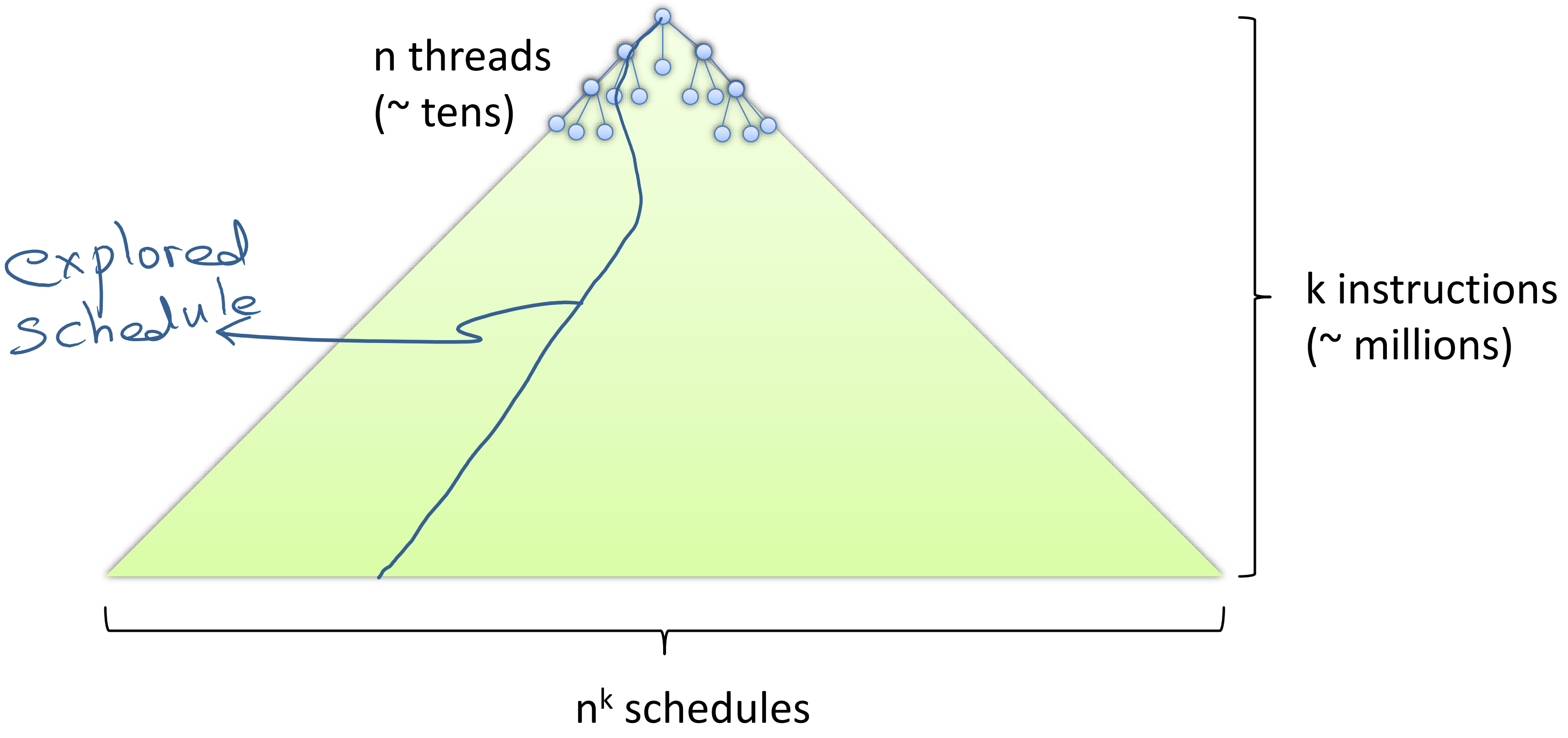
What is a “Bug” – first attempt

- Bug is defined as a particular buggy interleaving
- No algorithm can find the bug with a probably greater than $1/n^k$



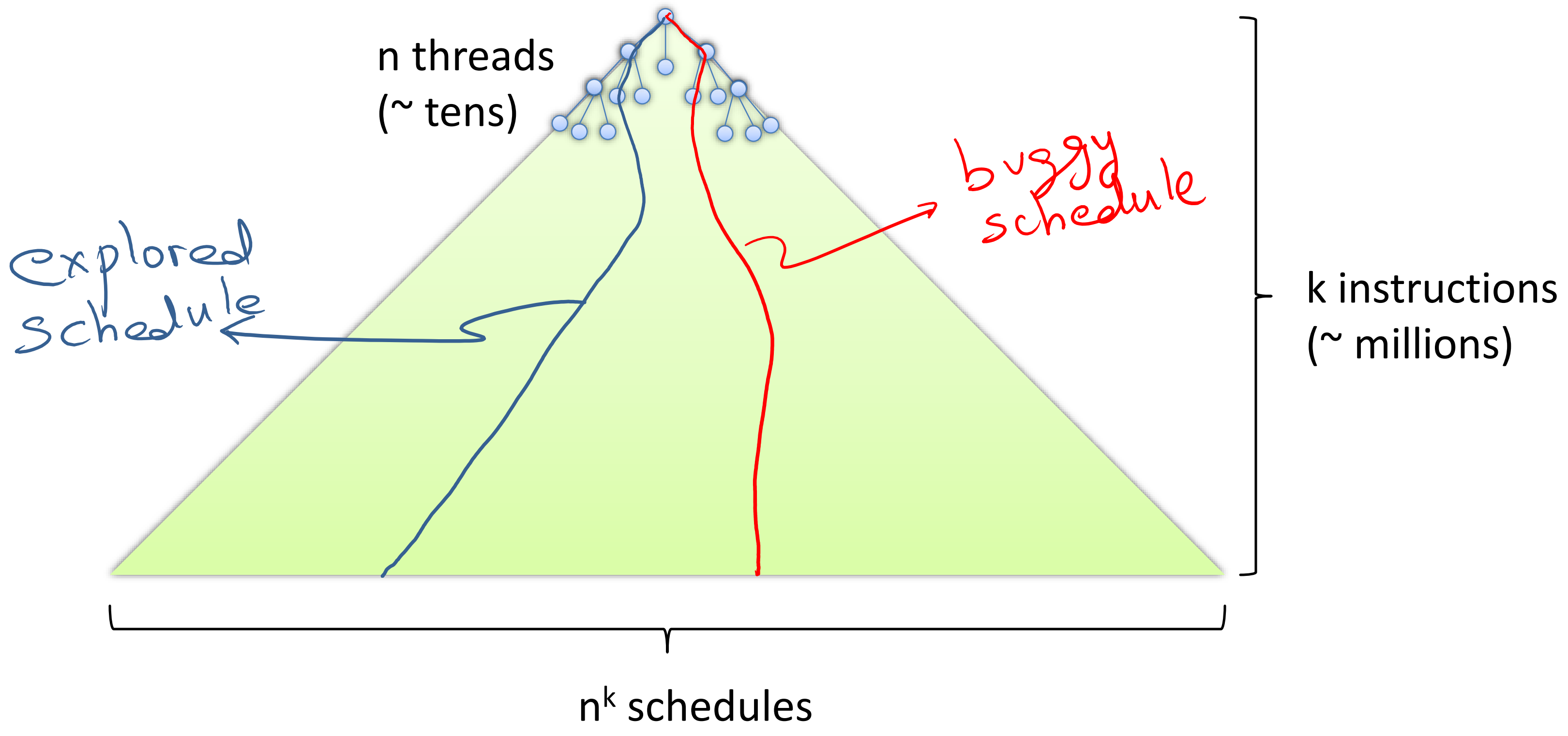
A Deterministic Algorithm

- Provides no guarantees



A Deterministic Algorithm

- Provides no guarantees



Effectiveness of Random Testing

- Suppose you have a system with n threads and at most k instructions are executed
  - Number of possible schedules is approximately n^k
- Say a concurrency bug is exposed by one particular interleaving among all these
- Probability of hitting that schedule is $\frac{1}{n^k}$

Debugging with Randomized Scheduling

Thread 1

```
assert(b != 0);
step(1);
step(2);
…
step(m);
a = 0;
```

Thread 2

```
assert(a != 0);
step(1);
step(2);
…
…
step(n);
b = 0;
```

Order Violation

Thread 1

```
void init(…) {
  …
  mThread=
      PR_CreateThread(mMain, …);
  …
}
```

Thread 2

```
void mMain() {
  mState=mThread->State;
}
```

Mozilla
nsthread.cpp

Atomicity Violation

Thread 1 | **Thread 2**

```
if (thd->proc_info)
                              thd->proc_info = NULL;
  fputs(thd->proc_info, …)
```

MySQL
ha_innodb.cc

Classifying Concurrency Bugs

- Root cause of a bug is characterized by the set of ordering constraints required to trigger the bug

Bug depth – Size of the minimum such set

Thread 1

```
void init(…) {
  …

  mThread=
      PR_CreateThread(mMain, …);
  …
}
```

Thread 2

```
void mMain() {
  mState=mThread->State;
}
```

A Bug of Depth 1

Parent

```
A:  …
B:  fork (child);
C:  p = malloc();
D:  …
E:  …
```

Child

```
F:   ….
G:  do_init();
H:  p->f++;
I:   …
J:   …
```

Possible schedules

A B C D E F G H I J ✓

A B F G H C D E I J ✗

A B F G C D E H I J ✓

A B F G C H D E I J ✓

A B F G H I J C D E ✗

…

Bug depth - number of ordering constraints sufficient to find the bug

A Bug of Depth 2

Parent

```
A: …
B: p = malloc();
C: fork (child);
D: …
E: if (p != NULL)
F:     p->f++;
G:
```

Child

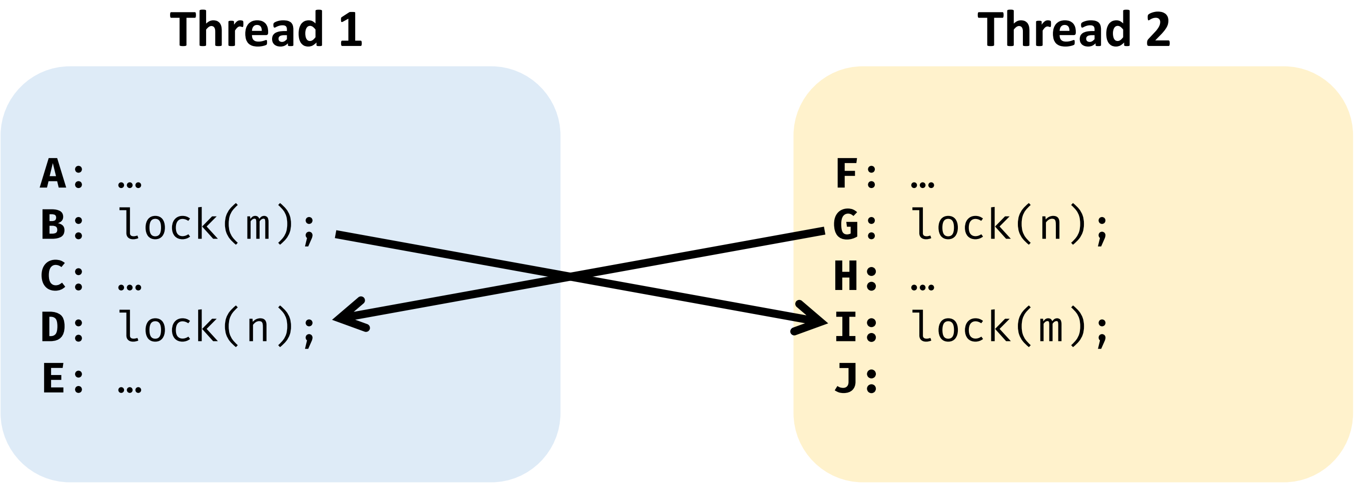
```
H: …
I: p = NULL;
J :   ….
```

Possible schedules

A B C D E F G H I J ✓
A B C D E H I J F G ✗
A B C H I D E G J ✓
A B C D H E F I J G ✓
A B C H D E I J F G ✗
…

Bug depth - number of ordering constraints sufficient to find the bug

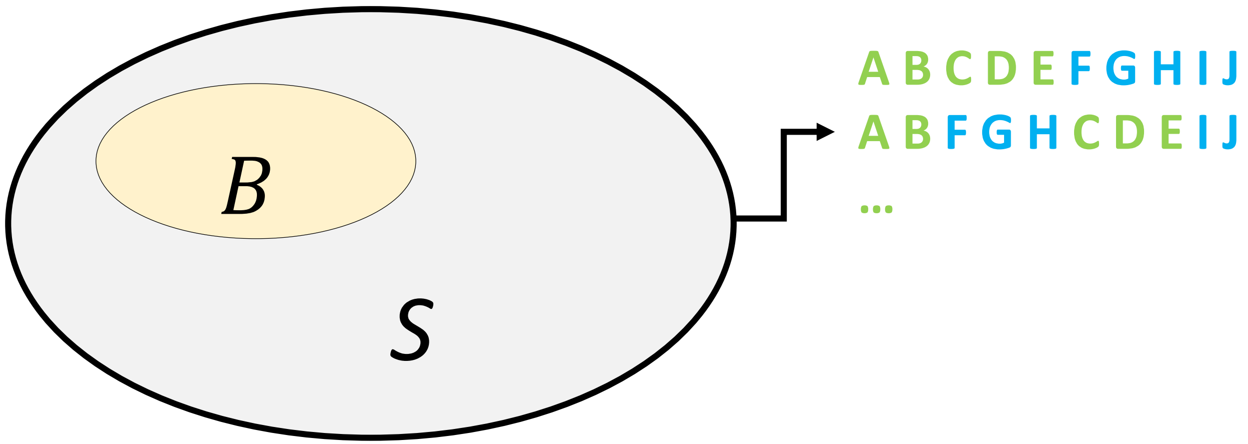
Another Bug of Depth 2



Bug depth - number of ordering constraints sufficient to find the bug

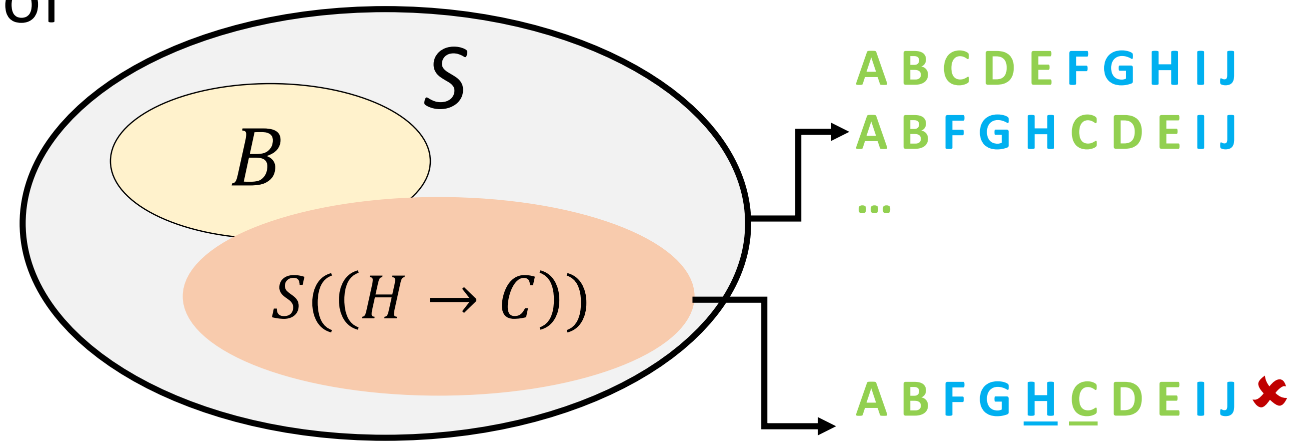
What is Bug Depth?

- A system is defined by its set of executions S
  - Each execution is a sequence of labelled events
- A concurrency bug B is some **strict** subset of S



What is Bug Depth?

- An ordering constraint c is a pair of events $c = (a \rightarrow b)$
- A schedule s satisfies $(a \rightarrow b)$ if a occurs before b in s
- $S(c_1, c_2, \ldots, c_d)$ – set of schedules that satisfy constraints $c_1, c_2, \ldots c_d$

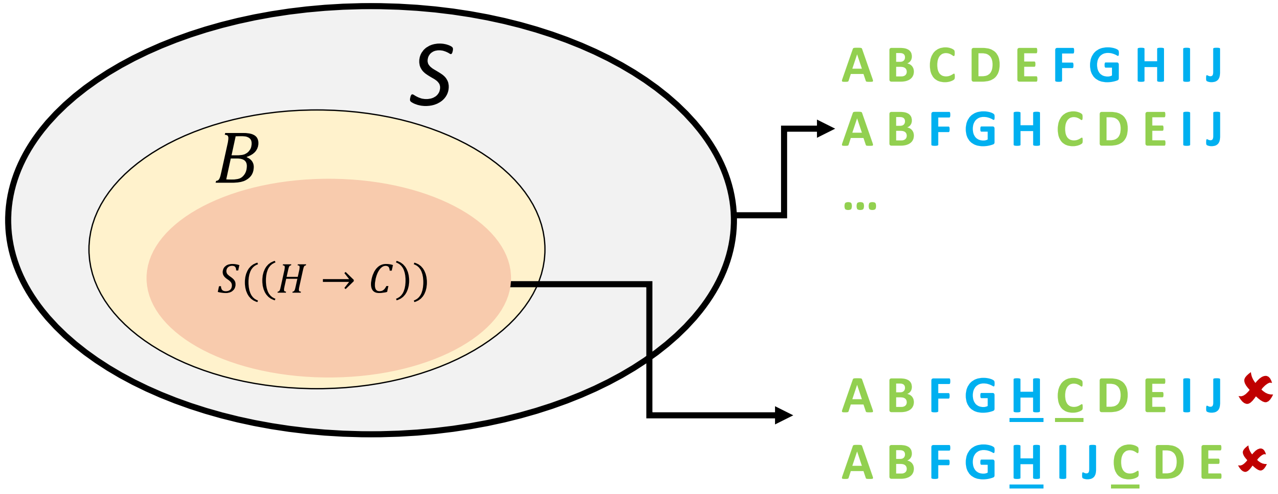


What is Bug Depth?

- A bug B is of depth d if there exists $c_1, c_2, \ldots, c_d$ such that

$$S(c_1, c_2, \ldots, c_d) \subseteq B$$

and d is the smallest such number for B

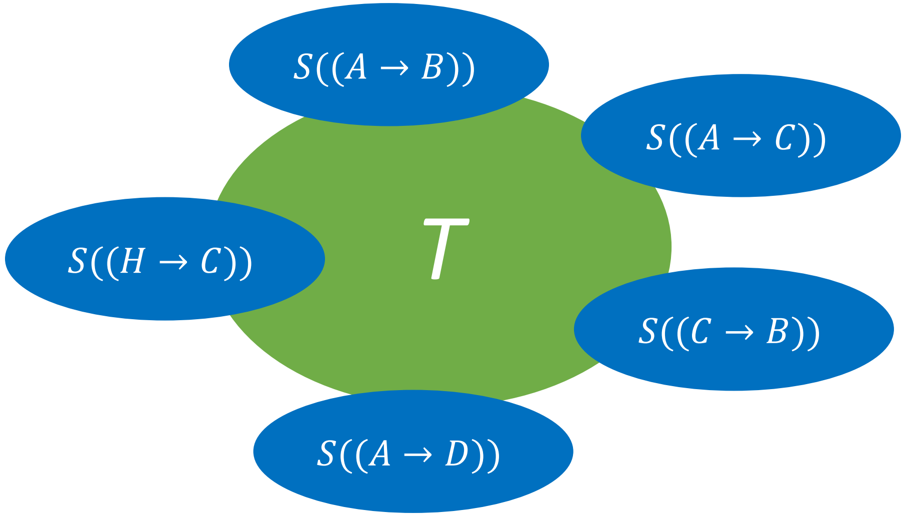


Finding All Bugs of Depth d

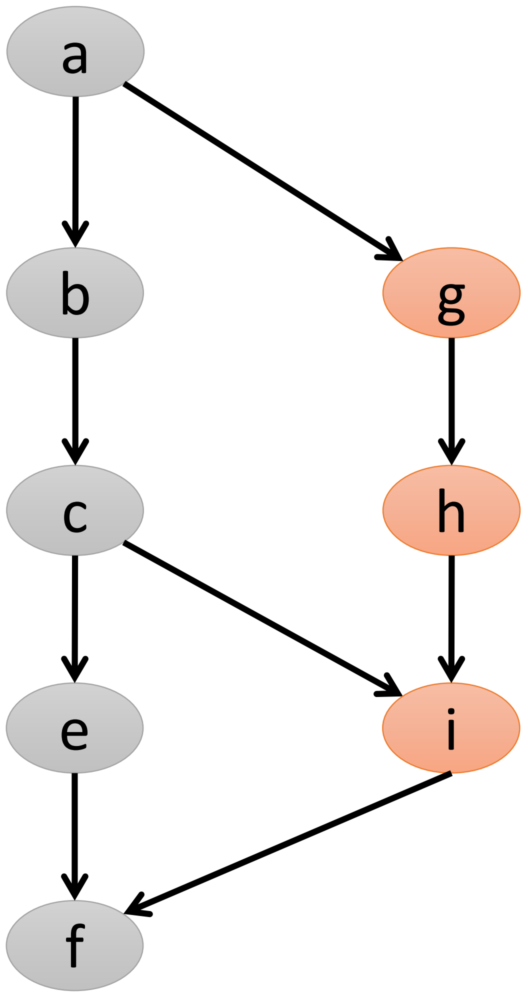
- A set of schedules T covers all bugs of depth d if

$$\forall\, c_1, \ldots, c_d : S(c_1, \ldots, c_d) \cap T \neq \phi$$

- Coverage problem: find the smallest such T

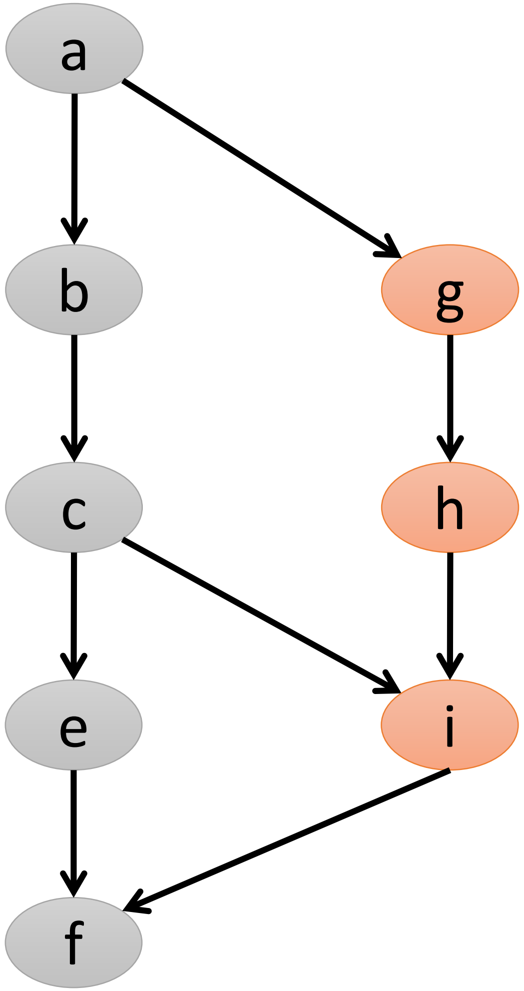


Let's study when $d = 1$

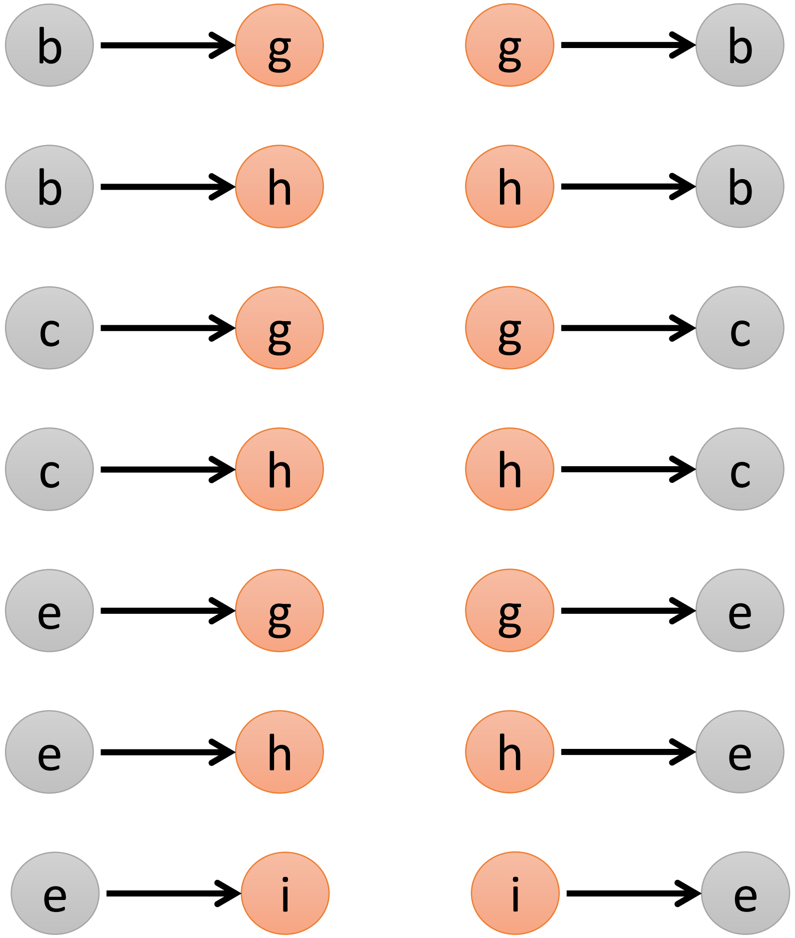


Which all pair of operations are concurrent?

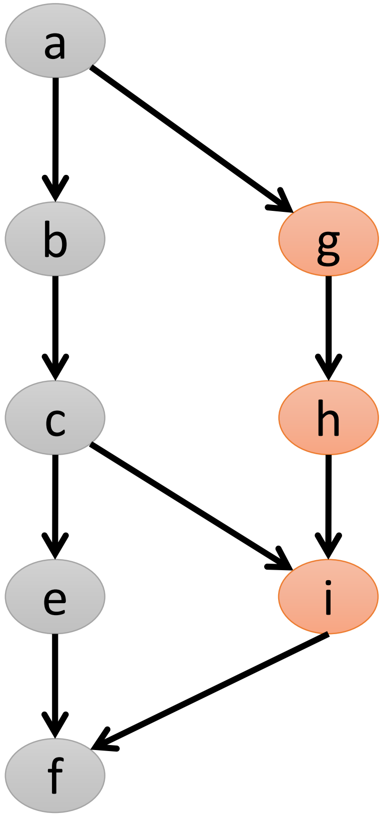
Let's study when $d = 1$



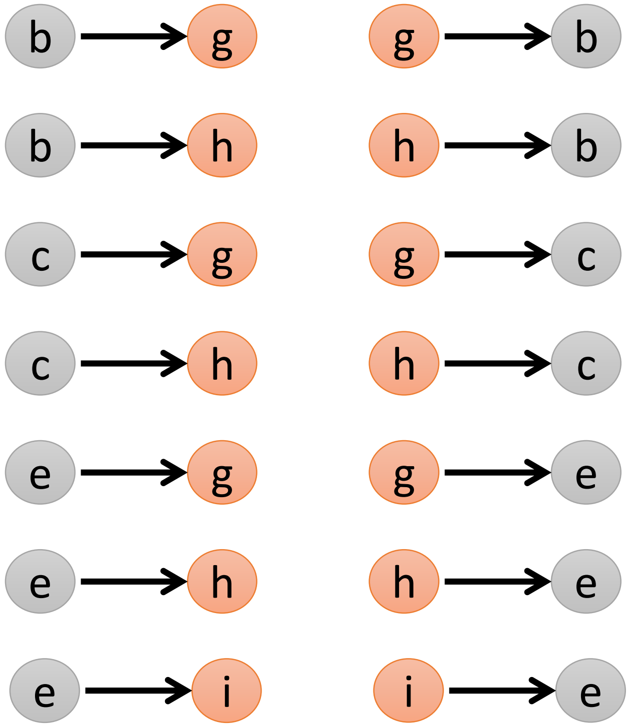
Need to cover all of:



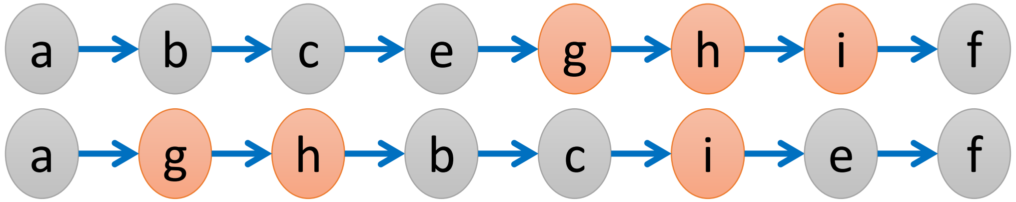
Let's study when $d = 1$



Need to cover all of:



Two interleavings are sufficient!



Concurrency Bugs and Bug Depth

- Most concurrency bugs are usually of **low depth**
  - Order violations – depth 1 (or 2 in presence of control flow)
  - Atomicity violations – depth 2
  - Deadlocks – depth 2 if 2 threads are involved, depth n if n threads are involved

- Bugs with greater depth are more subtle

A Bug of Depth 2

Main Thread	Filewriter Thread
…	…
`free(mutex);`	
	`mutex.unlock();`
`exit(0);`	

S. Burckhardt et al. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS, 2010.

An Ordering Bug of Depth 2

Main Thread	Filewriter Thread
…	…
`init = true;`	…
`t = new T();`	`if (init)`
…	`  t->state = 1;`
…	…

Presence of control dependence may complicate the interleaving

PCT: Probabilistic Concurrency Testing

- An intelligent **randomized** scheduler for finding concurrency bugs

- Provides probabilistic guarantees to expose bugs
  - Every run finds every bug with nontrivial probability
  - Repeated test runs increases the chance of finding a bug

S. Burckhardt et al. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS 2010.

PCT's Randomized Scheduler

- User-level scheduler is randomized priority-based
  - Every thread has a priority, lower number indicate lower priorities
- Only one thread is scheduled to execute at each step
- Low priority threads are scheduled only when higher-priority threads are blocked

- A dynamic execution has a few **priority change** points
  - Priority change points have fixed priorities assigned
  - A thread that reaches a change point will inherit the priority of the change point

PCT Algorithm

- INPUT: *n* threads, *k* instructions, and *d* priority change points
- STEPS:
  1. Assign *n* priority values *d*, *d+1*, …, *d+n-1* randomly to the *n* threads
  2. Pick *d-1* random priority change points from the *k* instructions. Each change point k_i, $1 \leq i < d$ has an associated priority of *i*
  3. Schedule threads based on their priorities
  4. When a thread reaches change point k_i, change the priority of that thread to *i*

Assumptions in PCT

Higher priority threads run faster

An ordering constraint ($a \rightarrow b$) will be met if a is executed by a higher priority thread

How PCT Works?

Initial thread priority

Thread 1 1

…

```
t = new T();
```

…

…

…

2 **Thread 2**

…

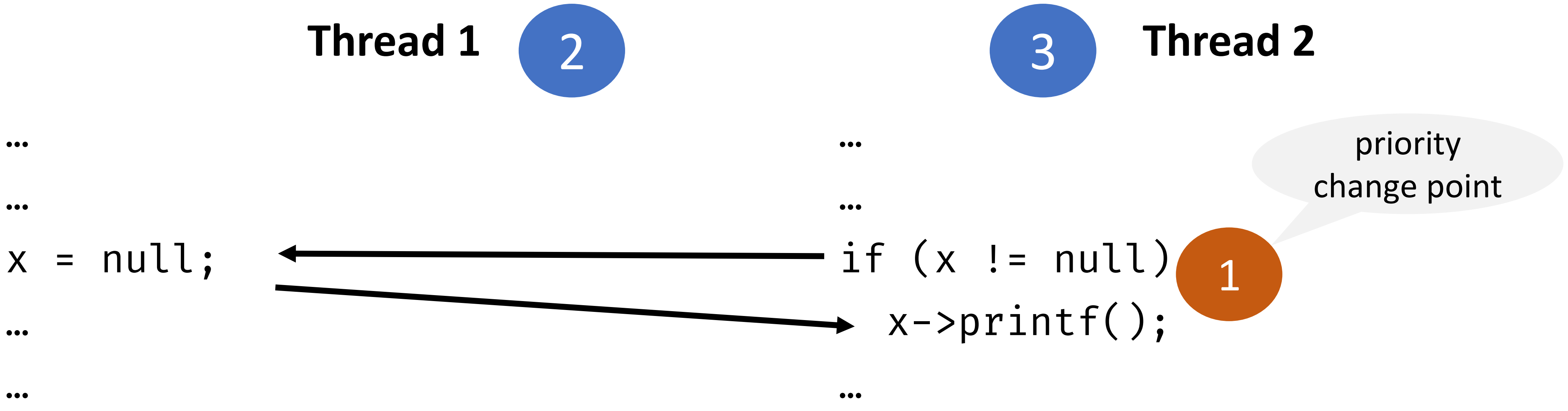
…

```
if (t->state == 1)
```

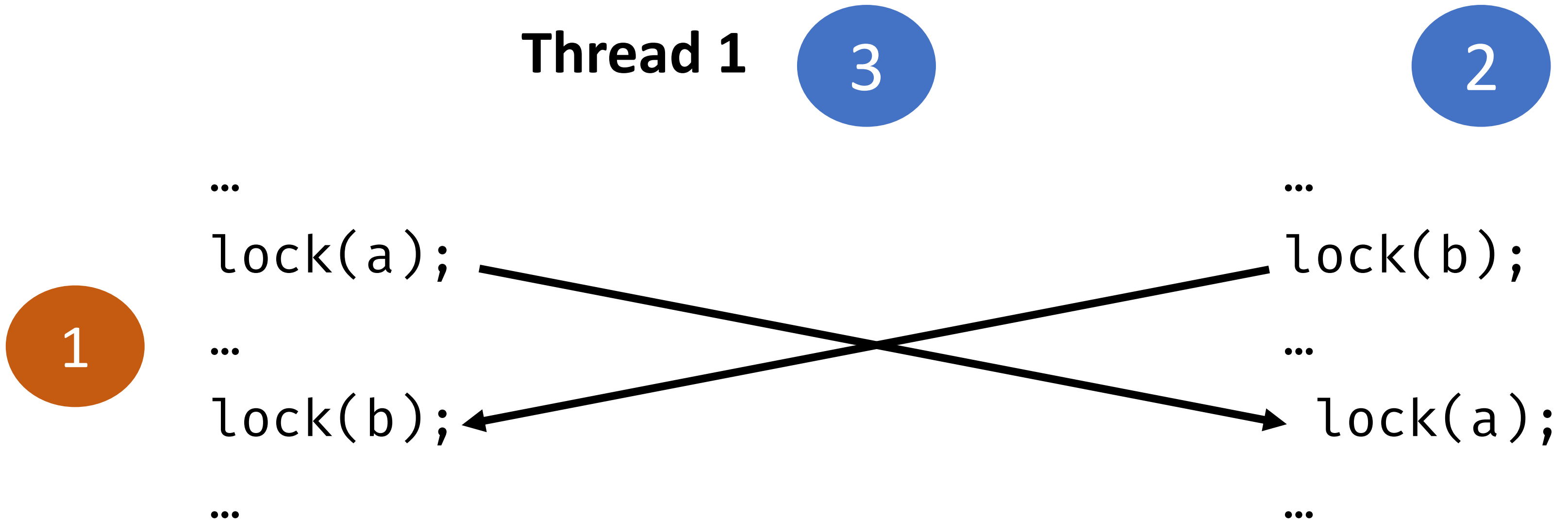
…

…

How PCT Works?



How PCT Works?



Issues to Consider in PCT

- Does not reuse OS thread priorities
  - Needs to force higher priority threads to run faster
  - PCT implements an **user-level scheduler** instead
- Consider priority inversion
  - Higher priority thread may be blocked for a resource owned by a lower priority thread
  - But there will be other schedules where the priorities will be in the correct order (probability $\frac{1}{n}$)
- Ensure starvation freedom
  - Higher priority threads may wait in a spin loop for a lower priority thread
  - Uses heuristics to identify and resolve such situations

Effectiveness of PCT

- Probability of finding any bug with depth d in PCT is $\frac{1}{nk^{(d-1)}}$
  - Compare the probability with naïve random testing which is $\frac{1}{n^k}$
- If d = 1 or d = 2 (common cases), then probabilities of finding a bug is $\frac{1}{n}$ and $\frac{1}{nk}$ respectively
- PCT is expected to do better than the worst-case bound

Why?

Measured Probability of Finding a Bug

Program	Stress Testing	Random Testing with Sleeps	PCT	
			Empirical	Worst-case Bound
Splash-FFT	0.06	0.27	0.50	0.50
Splash-LU	0.07	0.39	0.50	0.50
Splash-Barnes	0.0074	0.0101	0.4916	0.5
Pbzip2	0	0	0.701	0.0001
Work Steal Queue	0	0.001	0.002	0.0003
Dryad	0	0	0.164	$2*10^{-5}$

Effectiveness of PCT

- Probability of finding any bug with depth d in PCT is $\frac{1}{nk^{(d-1)}}$
  - Compare the probability with naïve random testing which is $\frac{1}{n^k}$
- If d = … g a bug is $\frac{1}{n}$ an…
- PCT …

- Good enough to have the priority change point on one from a set of instructions, need not be exact
- Multiple ways to trigger a bug (symmetric case in deadlocks)
- Buggy code can be repeated multiple times in a program, more chances of being exposed

Extensions of PCT

- PCT runs only a single thread at a time
  - Does not utilize multicore hardware, incurs large slowdowns
- PPCT: Parallel PCT
  - Insight: Need to control the schedule of only *d* threads to expose a bug of depth *d*
  - Partitions threads into high (> *d*) and low priority
  - Runs threads with higher priority parallelly, size of the lower priority set is bounded by *d*
- PCT serializes all threads, PPCT serializes only the low priority threads

S. Nagarakatte et al. Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection. PLDI 2012.

PPCT Algorithm

- INPUT: *n* threads, *k* instructions, and *d* priority change points
- STEPS:
  1. Pick a random thread and assign it a priority *d*. Insert the thread in a low priority set *L*. Insert all other threads into a high priority set *H*.
  2. Pick *d-1* random priority change points from the *k* instructions. Each change point k_i, $1 \leq i < d$ has an associated priority of *i.*
  3. At each scheduling step, schedule **any** non-blocked thread in *H*. If *H* is empty of all threads are blocked, then schedule the highest priority thread in *L*.
  4. When a thread reaches change point k_i, change the priority of that thread to *i* and insert in *L.*

CHESS: Systematic Schedule Exploration

M. Musuvathi et al. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI, 2008.

What have we learnt so far?

Systematic schedule exploration enumerates all possible thread interleavings

- Does not scale

PCT/PPCT argued in favor of intelligent randomized testing

What have we learnt so far?

PCT/PPCT argued in favor of intelligent randomized testing

CHESS performs systematic schedule exploration

S

thread interleavings

- Does not scale

M. Musuvathi et al. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.

Traditional Testing

Traditional Testing

```
testStartup();
while (true) {

  runTestScenario();

  if (*some condition*)
    break;

}
testShutdown();
```

What is required for systematic exploration?

- Suppose you have two threads contending on a lock
- Systematic exploration should explore both schedules – one where each thread wins the lock first

What is required for systematic exploration?

- Suppose you have two threads contending on a lock
- Sy here
  e

Basically capture all nondeterministic choices

Why Track Nondeterminism?

Capture all sources of nondeterminism

- Required for reliably reproducing errors

Ability to explore these nondeterministic choices

- Required for finding errors

Sources of Nondeterminism

- Input, environment
- Interleaving
- Other sources like compiler and hardware reordering

Input Nondeterminism

- Environment data can affect program execution
  - User can provide different inputs
  - Nondeterministic functions like `gettimeofday()`, `random()`

- **Idea**: Use "record and replay" techniques
  - Two phases – a record phase and a replay phase

Input Nondeterminism

- Environment data can affect program execution
  - User can provide different inputs
  - Nondeterministic functions like `gettimeofday()`, `random()`
- **Ide**
  -

Which phase is usually more expensive, record or replay?

Capturing Input Nondeterminism in CHESS

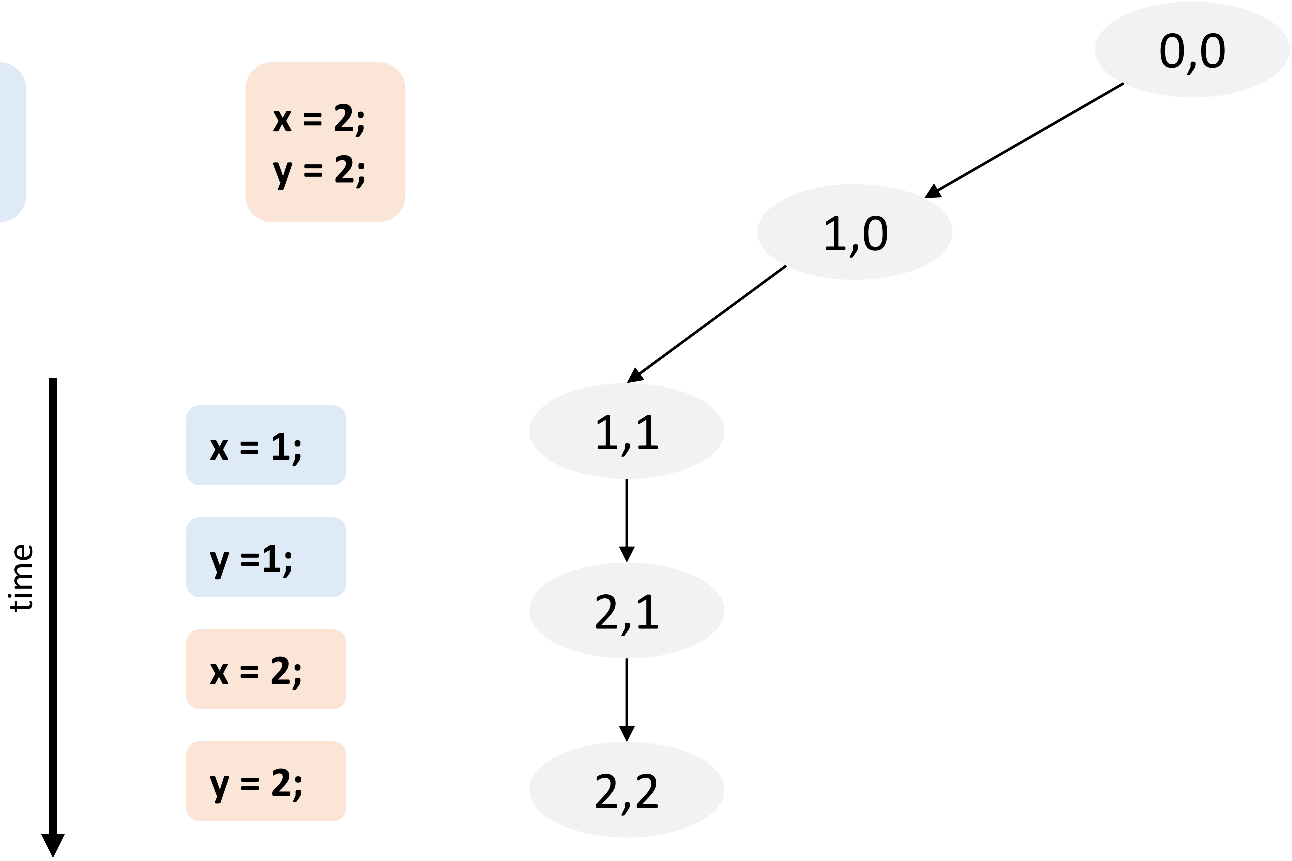
- CHESS is not a typical record-and-replay system
- Relies on the test setup to provide deterministic inputs
- Records a few nondeterministic events like current time, processor and thread id mapping, random numbers

Concurrent Executions are Nondeterministic

Thread 1	Thread 2
x = 1; **y = 1;**	**x = 2;** **y = 2;**

Concurrent Executions are Nondeterministic

Thread 1	Thread 2
x = 1; y = 1;	x = 2; y = 2;

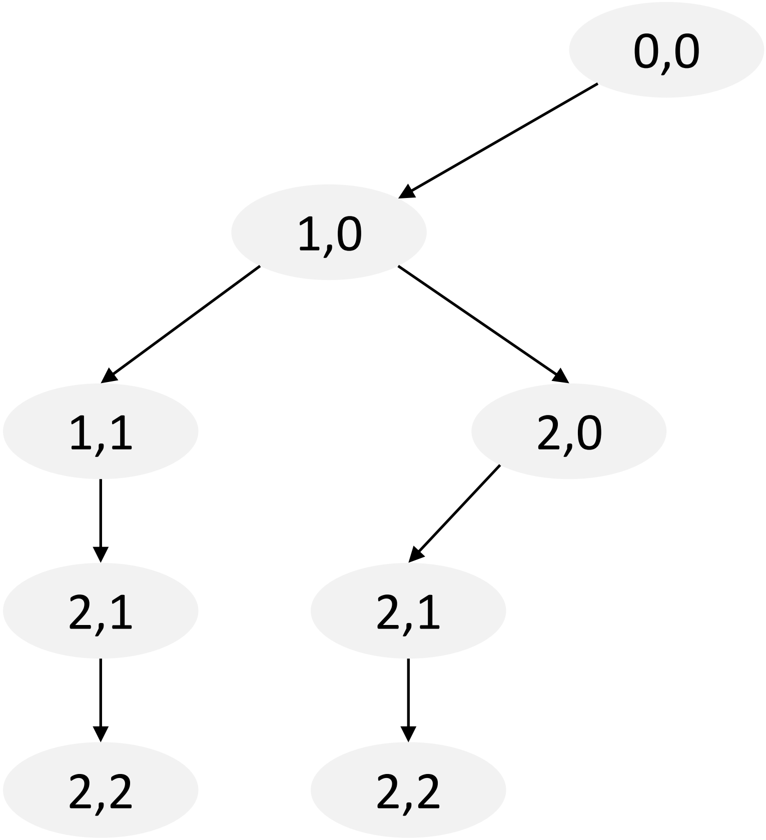


Concurrent Executions are Nondeterministic

Thread 1	Thread 2
`x = 1;`	`x = 2;`
`y = 1;`	`y = 2;`

time ↓

```
x = 1;
x = 2;
y = 1;
y = 2;
```



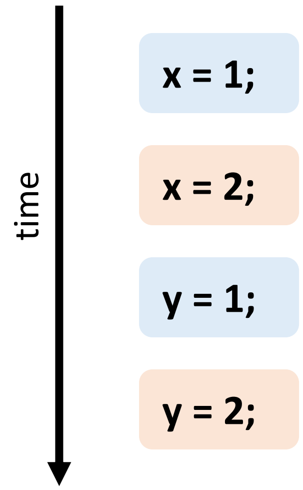
Concurrent Executions are Nondeterministic

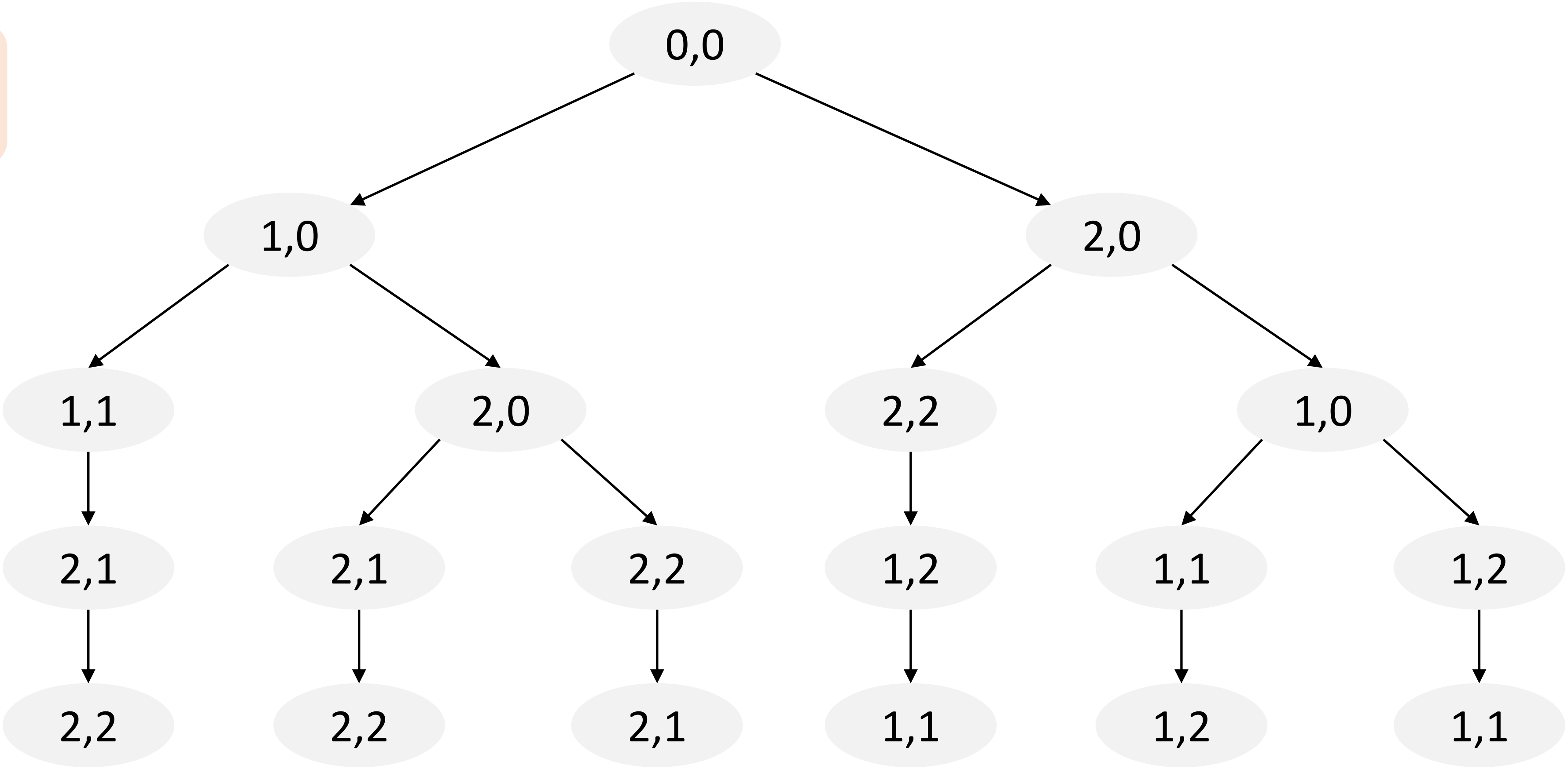
Thread 1

```
x = 1;
y = 1;
```

Thread 2

```
x = 2;
y = 2;
```





Scheduling Nondeterminism

Interleaving nondeterminism

- Threads can race to access shared variables or monitors
- OS can preempt threads at arbitrary points

Timing nondeterminism

- Timers can fire in different orders
- Sleeping threads wake up at arbitrary times in the future
- Asynchronous calls complete at arbitrary times in the future

CHESS in a nutshell

User-mode scheduler – controls all scheduler nondeterminism

Provides systematic overage of **all** thread interleavings

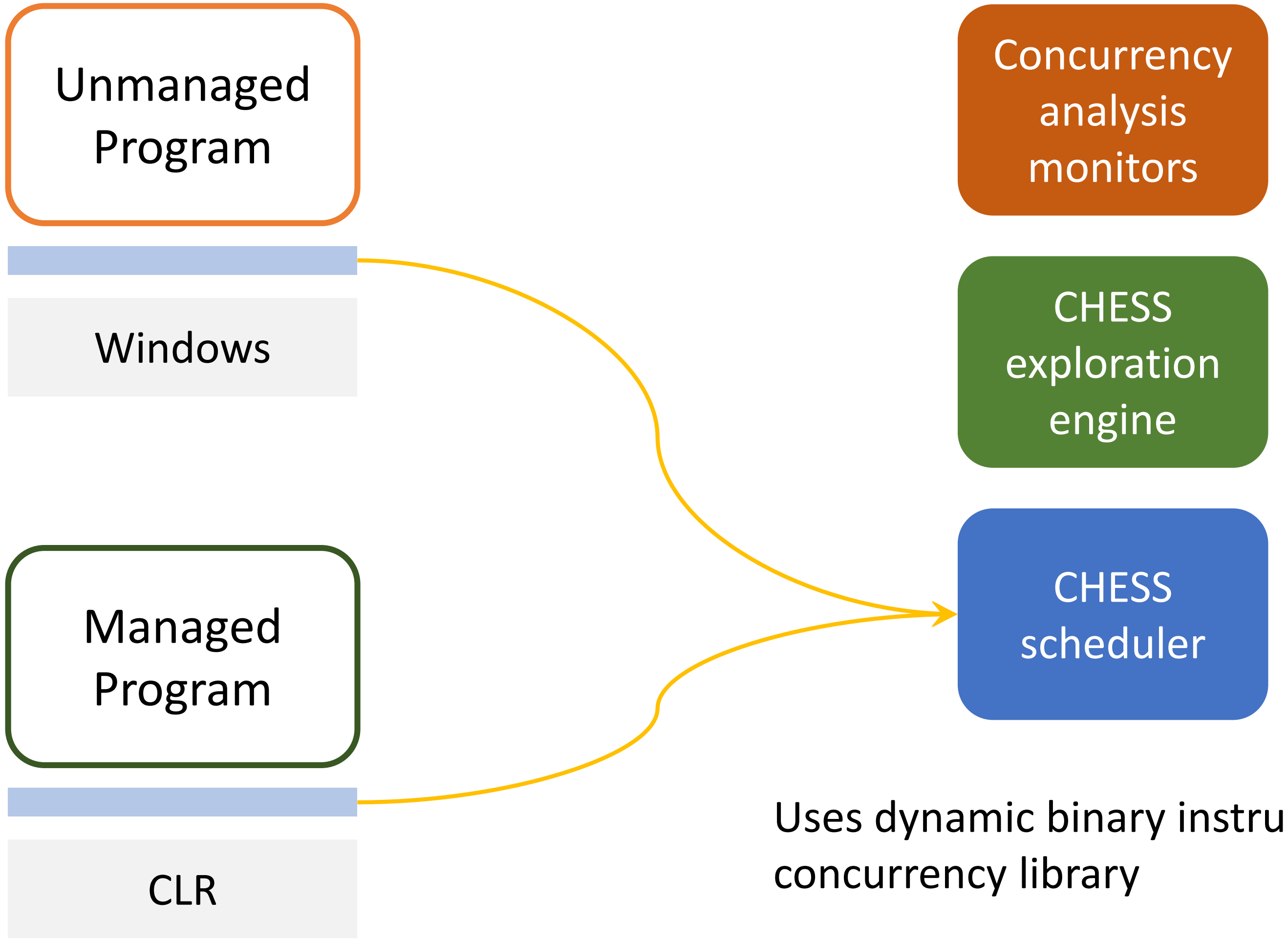
- Every program run takes a different thread interleaving

CHESS is precise, does not introduce **new behaviors**

Provides replay capability for easy debugging

- Reproduce the interleaving for every run

CHESS Architecture



Uses dynamic binary instrumentation to intercept calls to the concurrency library

Scheduler captures the happens-before graph of the execution

Interleaving Nondeterminism

```
init:
  balance = 100;
```

Deposit Thread

```
void Deposit100(){
  EnterCriticalSection(&cs);
  balance += 100;
  LeaveCriticalSection(&cs);
}
```

Withdraw Thread

```
void Withdraw100(){
  int t;

  EnterCriticalSection(&cs);
  t = balance;
  LeaveCriticalSection(&cs);

  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);

}
```

```
final:
  assert(balance = 100);
```

Invoke the Scheduler at Preemption Points

```
init:
  balance = 100;
```

Deposit Thread

```
void Deposit100(){
  CHESSSchedule();
  EnterCriticalSection(&cs);
  balance += 100;
  CHESSSchedule();
  LeaveCriticalSection(&cs);
}
```

Withdraw Thread

```
void Withdraw100(){
  int t;
  CHESSSchedule();
  EnterCriticalSection(&cs);
  t = balance;
  CHESSSchedule();
  LeaveCriticalSection(&cs);
  CHESSSchedule();
  EnterCriticalSection(&cs);
  balance = t - 100;
  CHESSSchedule();
  LeaveCriticalSection(&cs);
}
```

Each call is a potential preemption point

```
final:
  assert(balance = 100);
```

Insert Predictable Delays with Additional Synchronization

Deposit Thread

```
void Deposit100(){




  waitEvent(e1);
  EnterCriticalSection(&cs);
  balance += 100;
  LeaveCriticalSection(&cs);
  setEvent(e2);
}
```

Withdraw Thread

```
void Withdraw100(){
  int t;
  EnterCriticalSection(&cs);
  t = balance;
  LeaveCriticalSection(&cs);
  setEvent(e1);




  waitEvent(e2);
  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);
}
```

Blindly Inserting Delays can lead to Deadlocks!

Deposit Thread

```
void Deposit100(){




  EnterCriticalSection(&cs);
  balance += 100;
  waitEvent(e1);
  LeaveCriticalSection(&cs);
}
```

Withdraw Thread

```
void Withdraw100(){
  int t;
  EnterCriticalSection(&cs);
  t = balance;
  LeaveCriticalSection(&cs);
  setEvent(e1);




  EnterCriticalSection(&cs);
  balance = t - 100;
  LeaveCriticalSection(&cs);
}
```

CHESS Scheduler Basics

- CHESS is a non-preemptive, fair, round-robin and priority-based, starvation-free scheduler
  - Executes chunks of code atomically
- Scheduler basically captures the happens-before graph for the execution
- Each graph node tracks threads, synchronization resources, and the operations, and whether tasks are enabled or disabled

CHESS Scheduler Basics

- Introduces an event per thread, every thread blocks on its event
- The scheduler wakes one thread at a time by enabling the corresponding event
- The scheduler does not wake up a disabled thread
  - Need to know when a thread can make progress
  - Synchronization wrappers provide this information
- The scheduler has to pick one of the enabled threads
  - The exploration engine decides for the scheduler

CHESS Scheduler Basics

Three steps

- **Record**
  - Schedules a thread till the thread yields
- **Replay**
  - Replays a sequence of scheduling choices from a trace file
- **Search**
  - Uses the enabled information at each schedule point to determine the scheduler for the next iteration

Traditional Testing vs CHESS

Traditional Testing

```
testStartup();
while (true) {

  runTestScenario();

  if (*some condition*)
    break;

}
testShutdown();
```

CHESS

```
testStartup();
while (true) {

  runTestScenario();

  if (*some condition*)
    break;

}
testShutdown();
```

replay

record

search

Preemption bounding

- Systematically inserts a **small** number preemptions
  - Preemptions are context switches forced by the scheduler (e.g. timeslice expiration)
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end

```
x = 1;
if (p != 0) {
  x- = p->f;
}
```

```
p = 0;
```

Preemption bounding

- Systematically inserts a **small** number preemptions
  - Preemptions are context switches forced by the scheduler (e.g. timeslice expiration)
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end

```
x = 1;
if (p != 0) {
```

preempted

```
p = 0;
```

```
  x = p->f;
}
```

Preemption bounding

- Systematically inserts a **small** number preemptions
  - Preemptions are context switches forced by the scheduler (e.g. timeslice expiration)
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end

Helps alleviate the problem of state space explosion

```
p = 0;
```

```
  x = p->f;
}
```

Advantages of preemption bounding

Most errors are caused by few (<2) preemptions (similar to bug depth)

Generates an easy to understand error trace

- Preemption points almost always point to the root cause of the bug

Leads to good heuristics

- Insert more preemptions in code that needs to be tested
- Avoid preemptions in libraries
- Insert preemptions in recently modified code

A good coverage guarantee to the user

- When CHESS finishes exploration with 2 preemptions, any remaining bug requires 3 preemptions or more

Contributions of CHESS

Integrates stateless model checking ideas to testing concurrent programs with minimal perturbation

Ability to consistently reproduce erroneous interleavings

DTHREADS: Efficient and Deterministic Multithreading

T. Liu et al. DTHREADS: Efficient Deterministic Multithreading. SOSP, 2011.

Remember the Sources of Nondeterminism?

- Input, environment
- Interleaving
- Other sources like compiler and hardware reordering

Deterministic Multithreading

- Deterministic execution can **simplify** multithreading
  - Executing the same program with same inputs will always provide same results
- Would simplify
  - Testing and debugging
  - Record and replay mechanism
  - Fault tolerance mechanisms

Deterministic Execution Example

```
int a = 0;
int b = 0;
int main() {
  spawn(thread1);
  spawn(thread2);
  print(a, b);
}
```

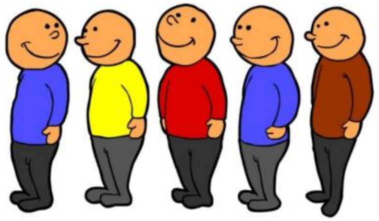
```
void thread1() {
  if (b == 0) {
    a = 1;
  }
}

void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```

How DTHREADS Provides Determinism

Isolation

Deterministic Time

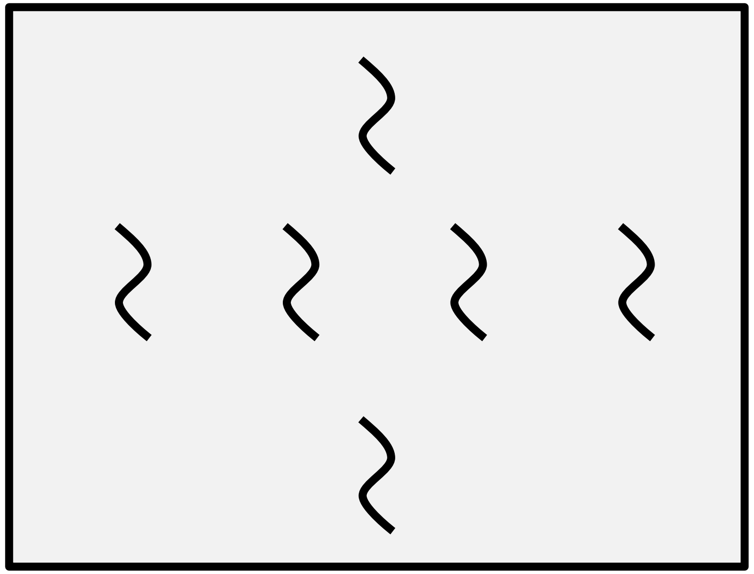


Deterministic Order

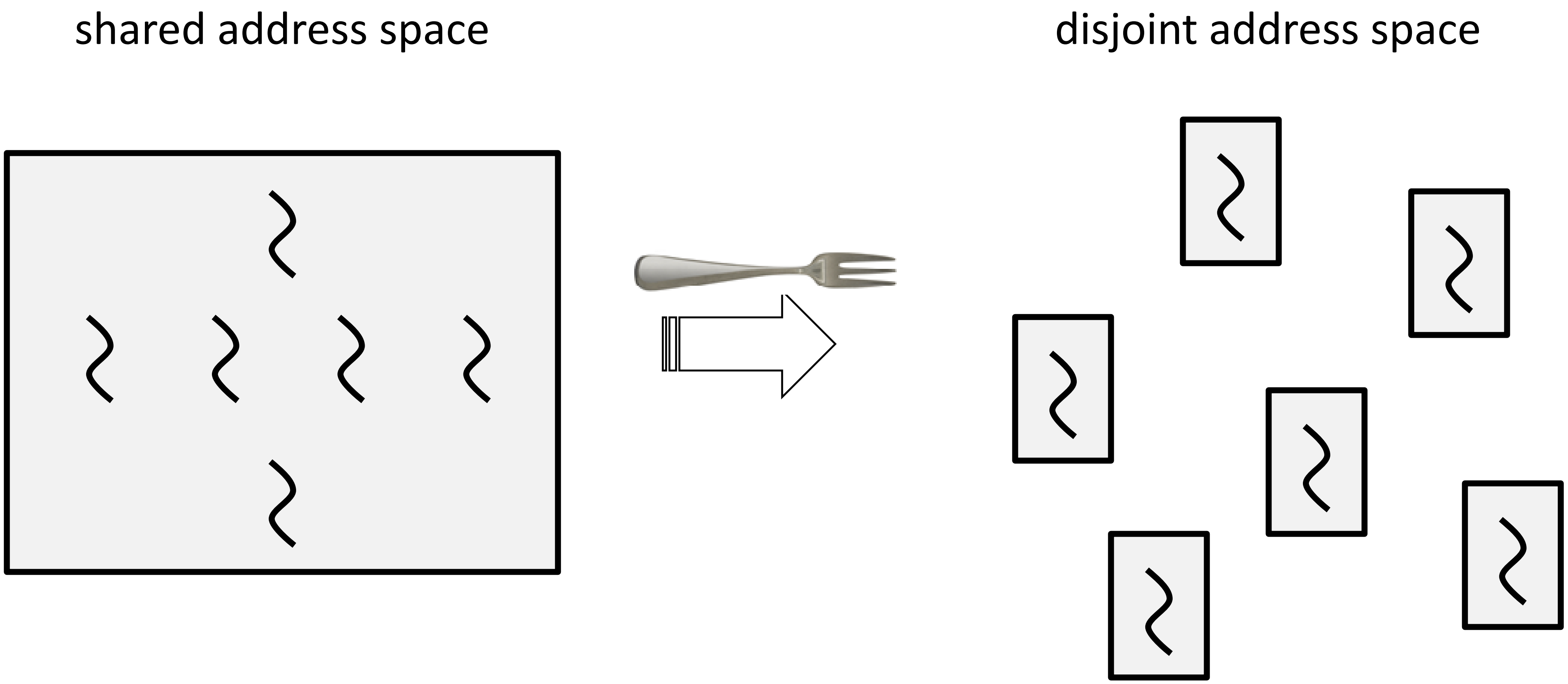
T Liu et al. DTHREADS: Efficient and Deterministic Multithreading. SOSP 2011.

Isolated Memory Access

shared address space



Isolated Memory Access



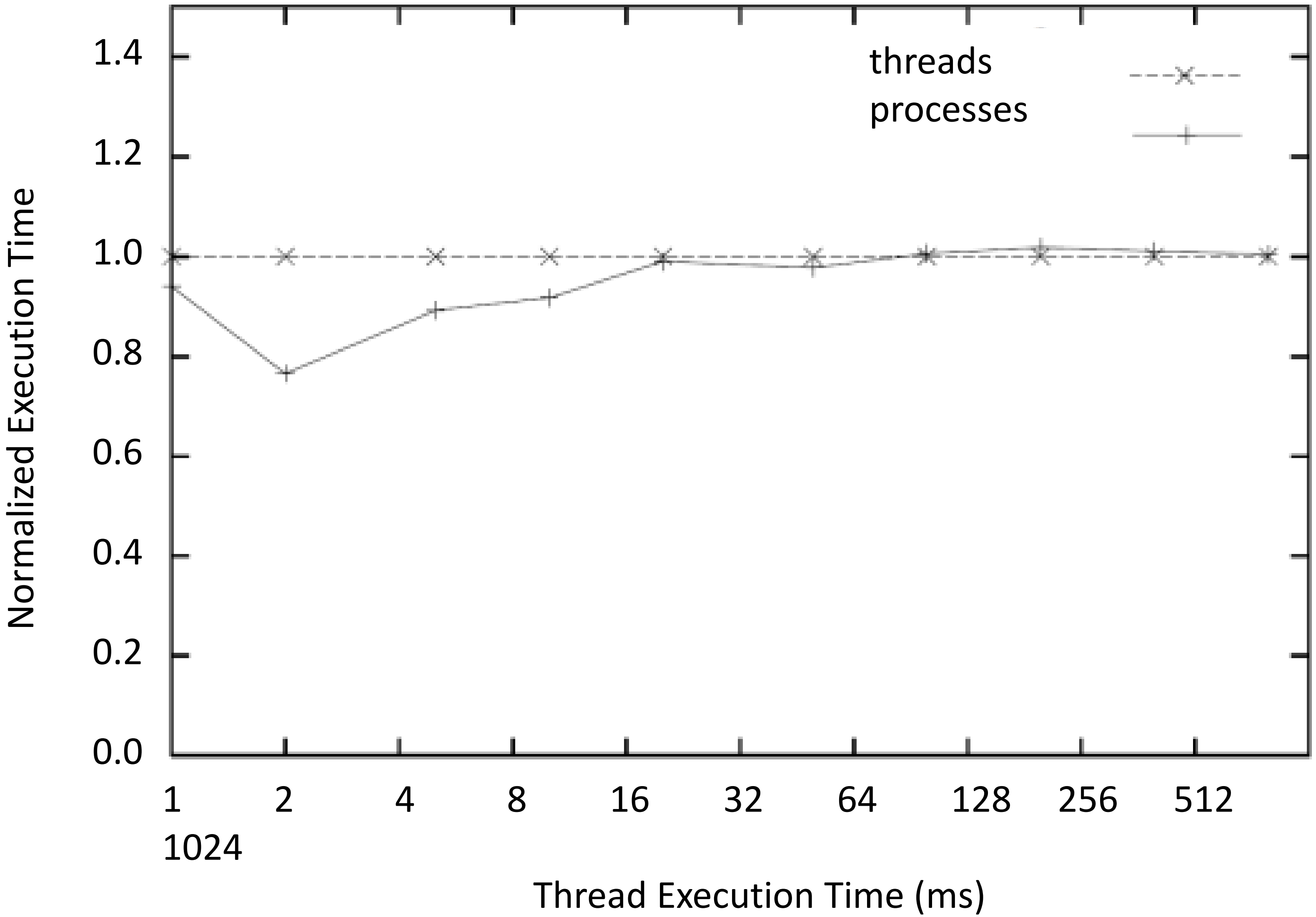
Isolated Memory Access

shared address space

disjoint address space

- Processes have separate address spaces → Implies that updates to shared memory are not visible
- Updates are made visible only at synchronization points
- Code regions between synchronization operations behave as atomic transactions

Performance: Processes vs. Threads

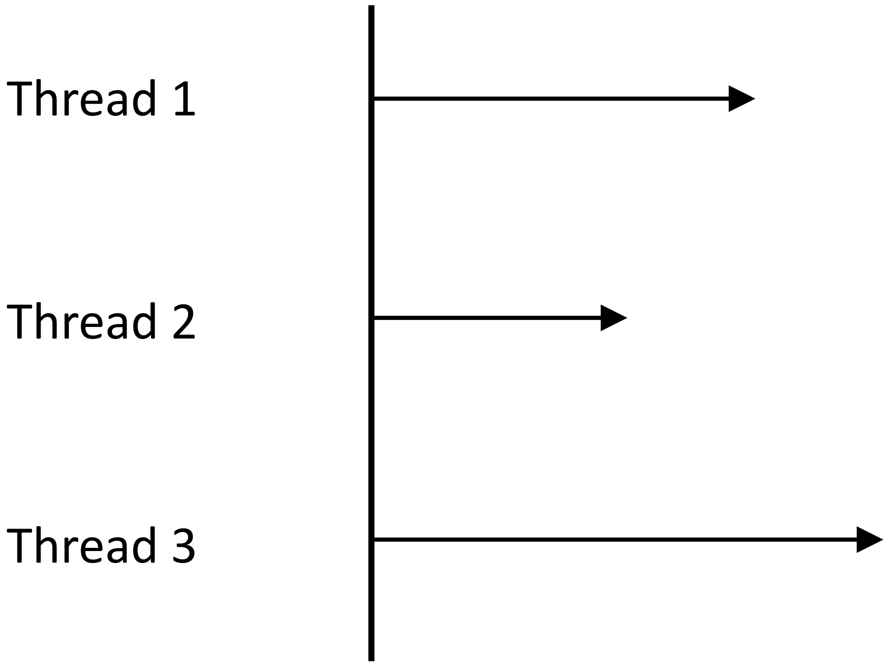


Challenges to Consider with Memory Isolation

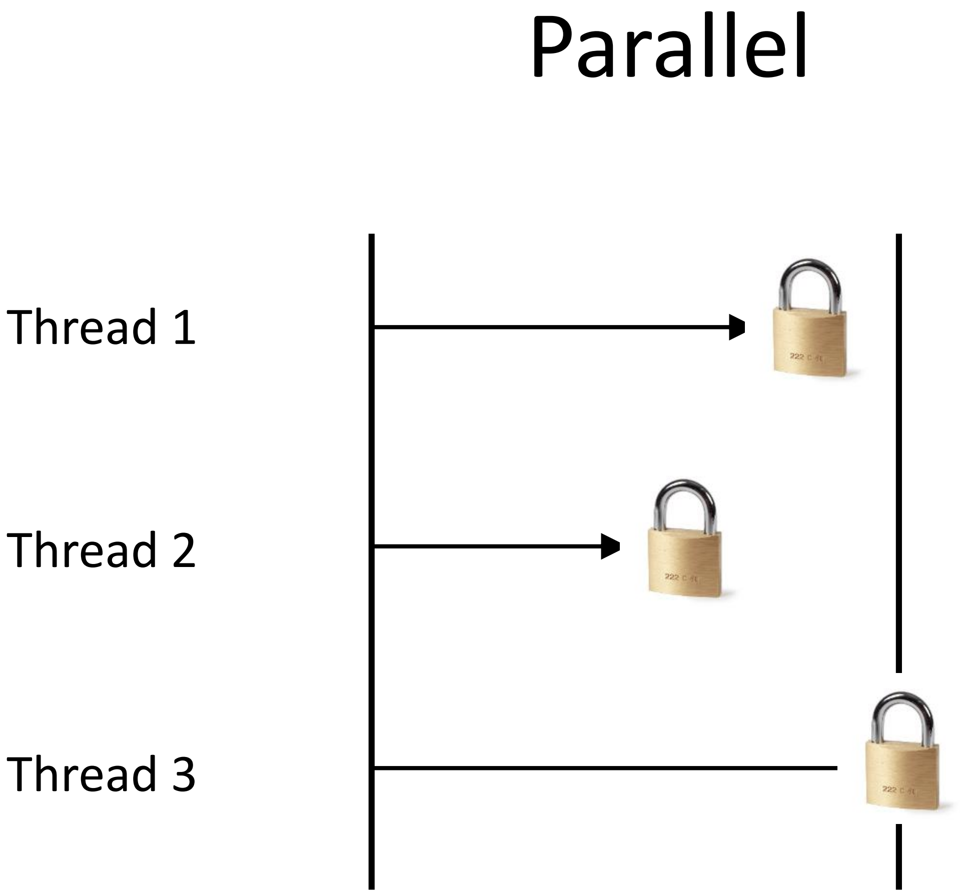
- DTHREADS now needs to explicitly manage shared resources like file descriptors
- Needs to generate deterministic thread and process ids
- Uses memory mapped files to share shared data (e.g., globals, heap) across processes
  - Two copies are created – one is read-only and the other (CoW) is for local updates

DTHREADS Phases

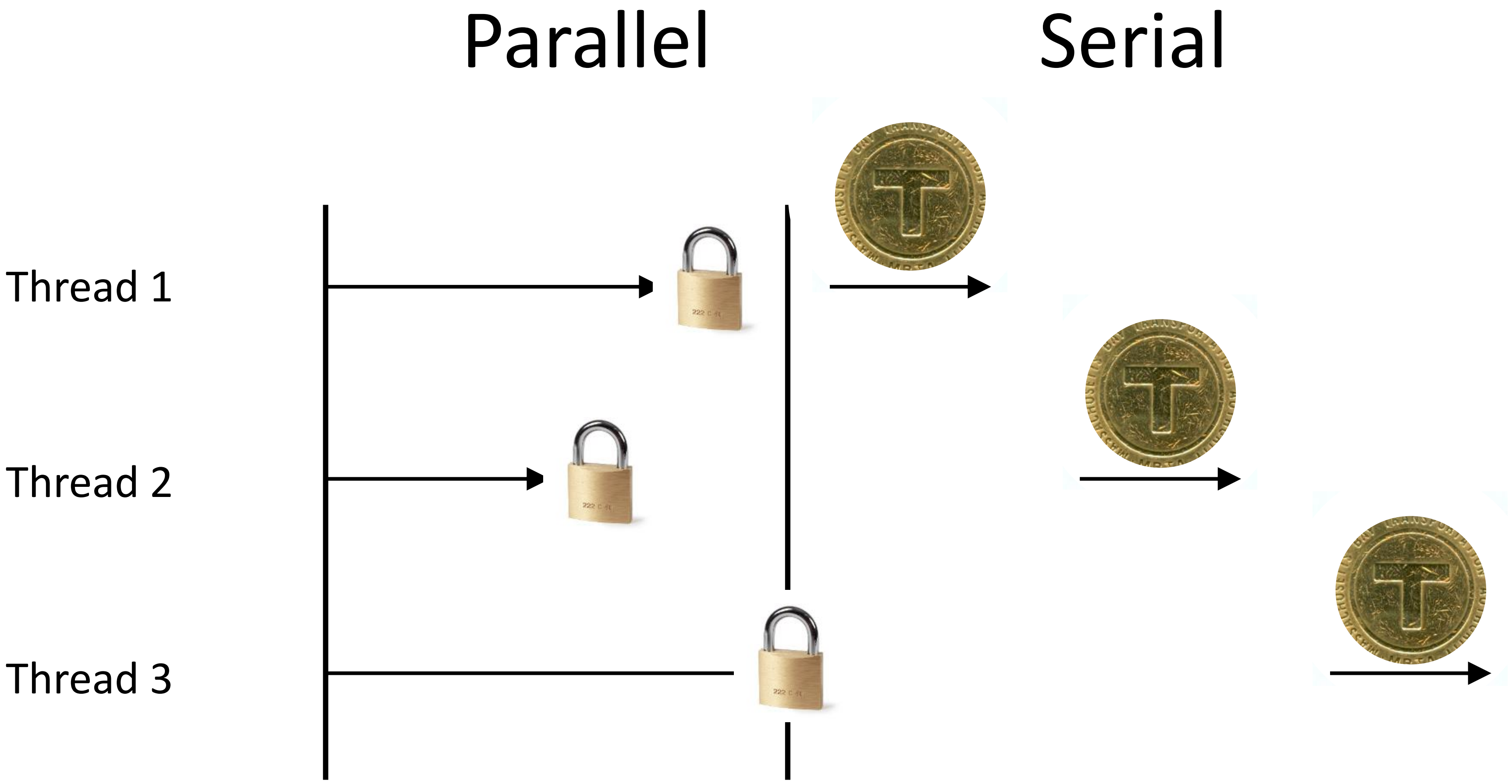
Parallel



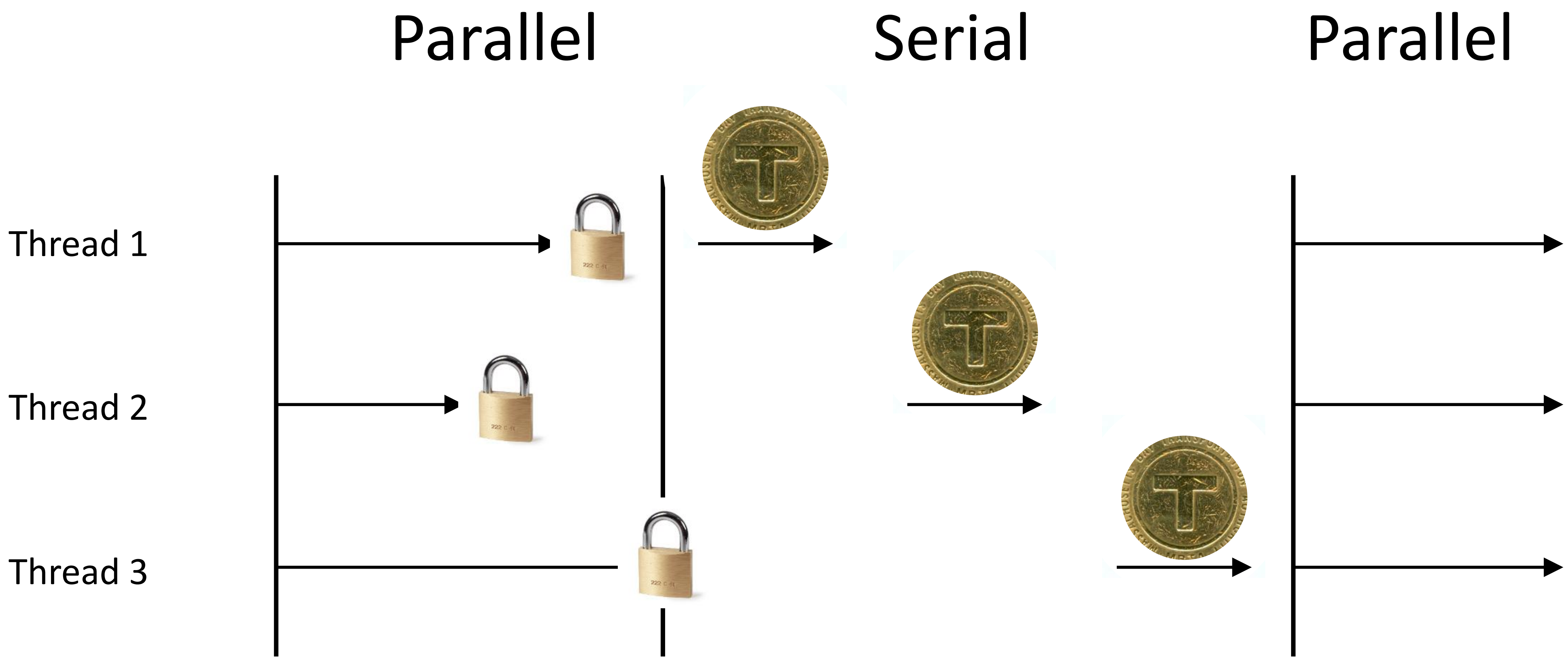
DTHREADS Phases



DTHREADS Phases



DTHREADS Phases



Shared-Memory Updates in Parallel Phase

- Threads have a read-only mapping of the shared pages at the beginning of the parallel phase
- Reads are performed from the shared page
- Upon a write, a private copy of the page is created (CoW) and the write operates on the private copy

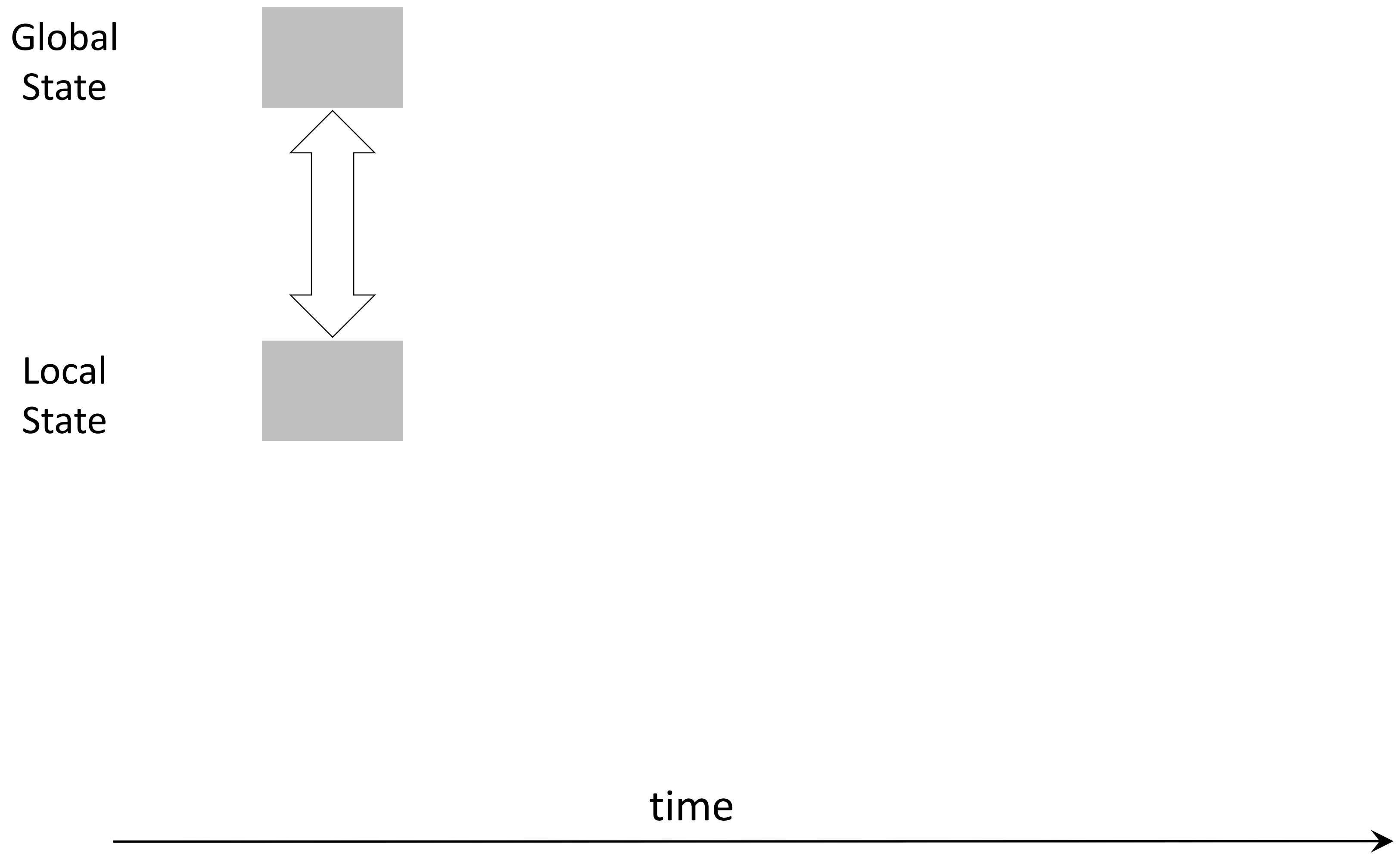
“Shared Memory”

Snapshot pages before modifications

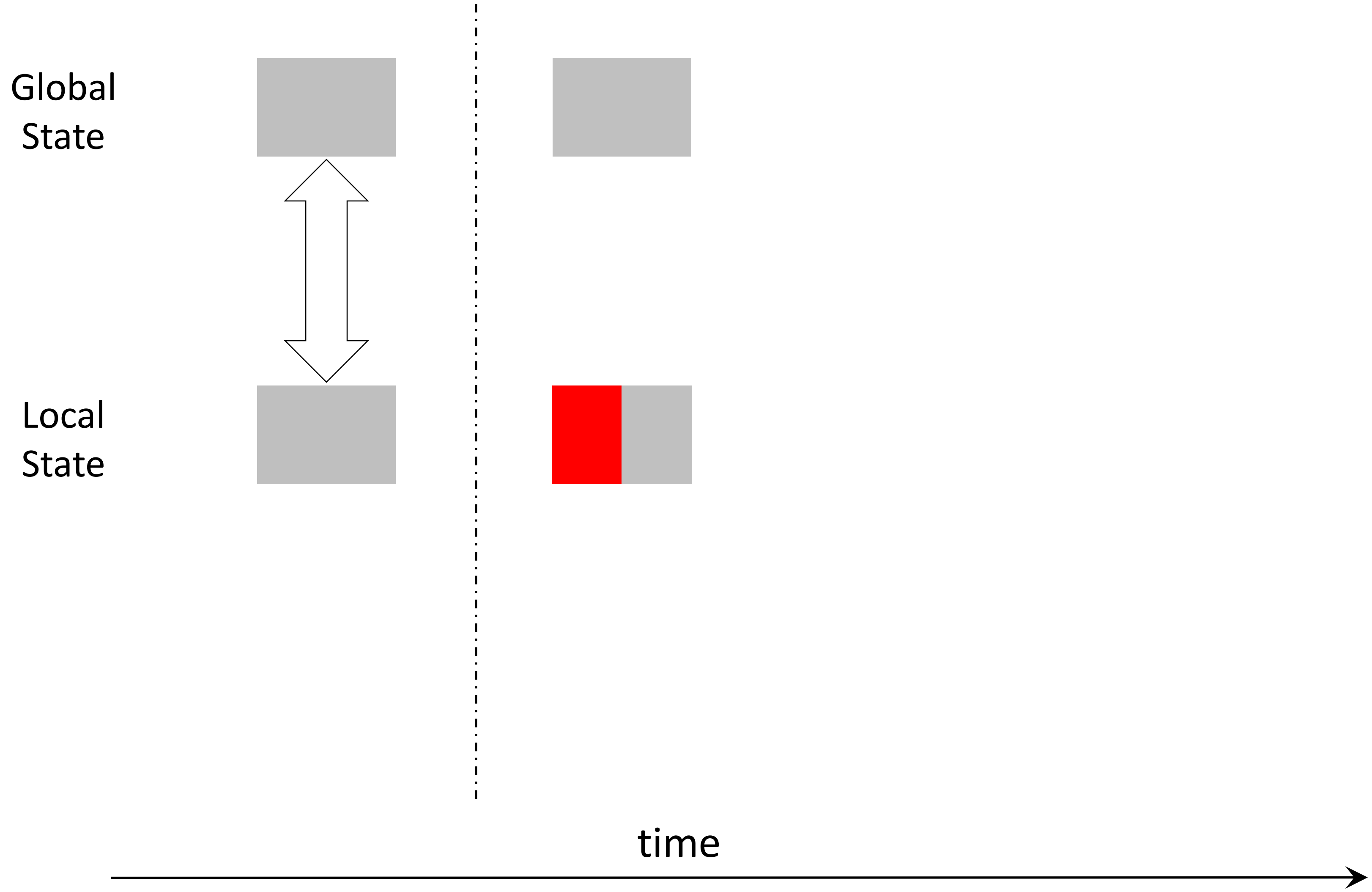
“Shared Memory”

Write back diffs

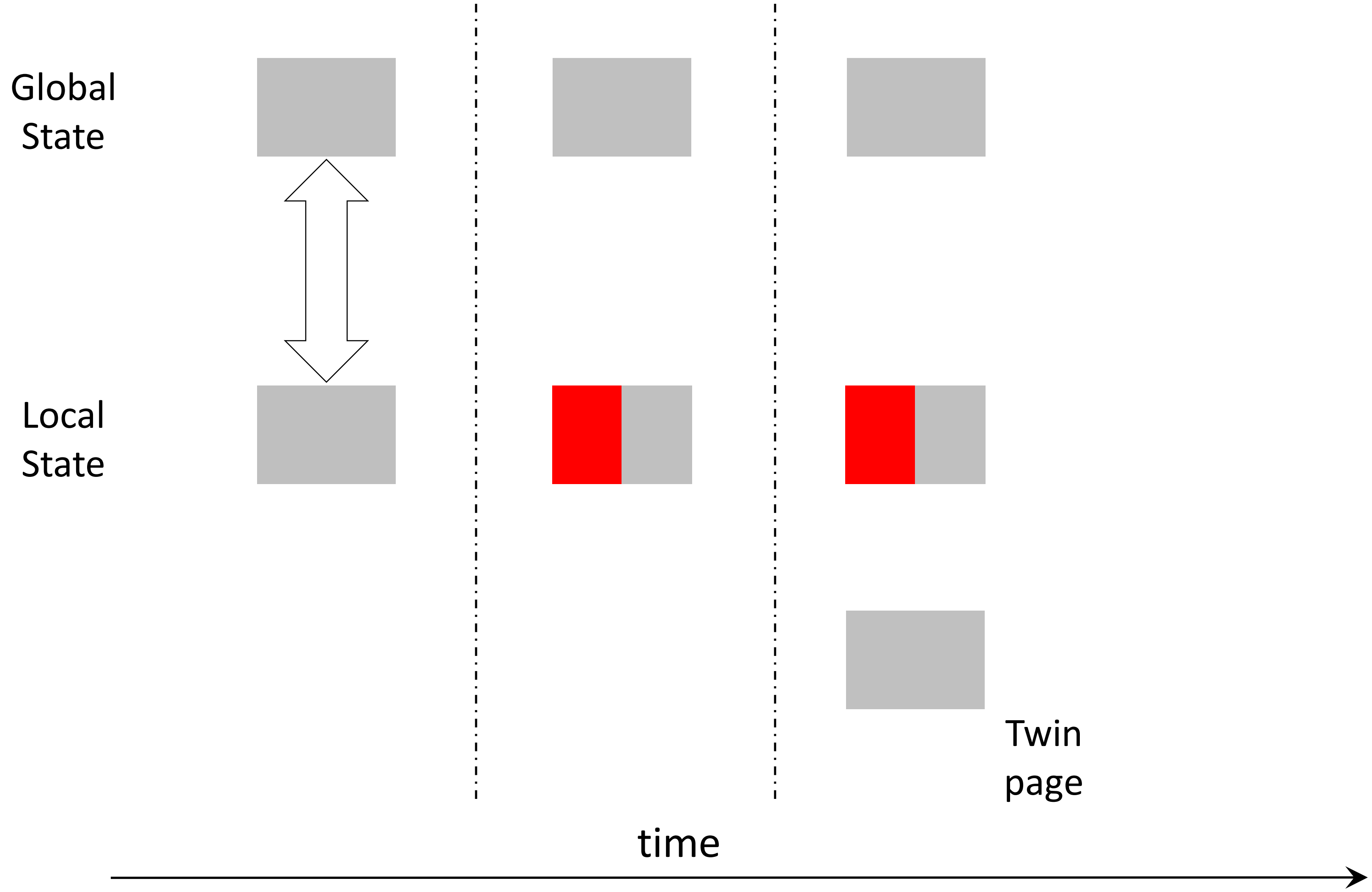
Commit Protocol



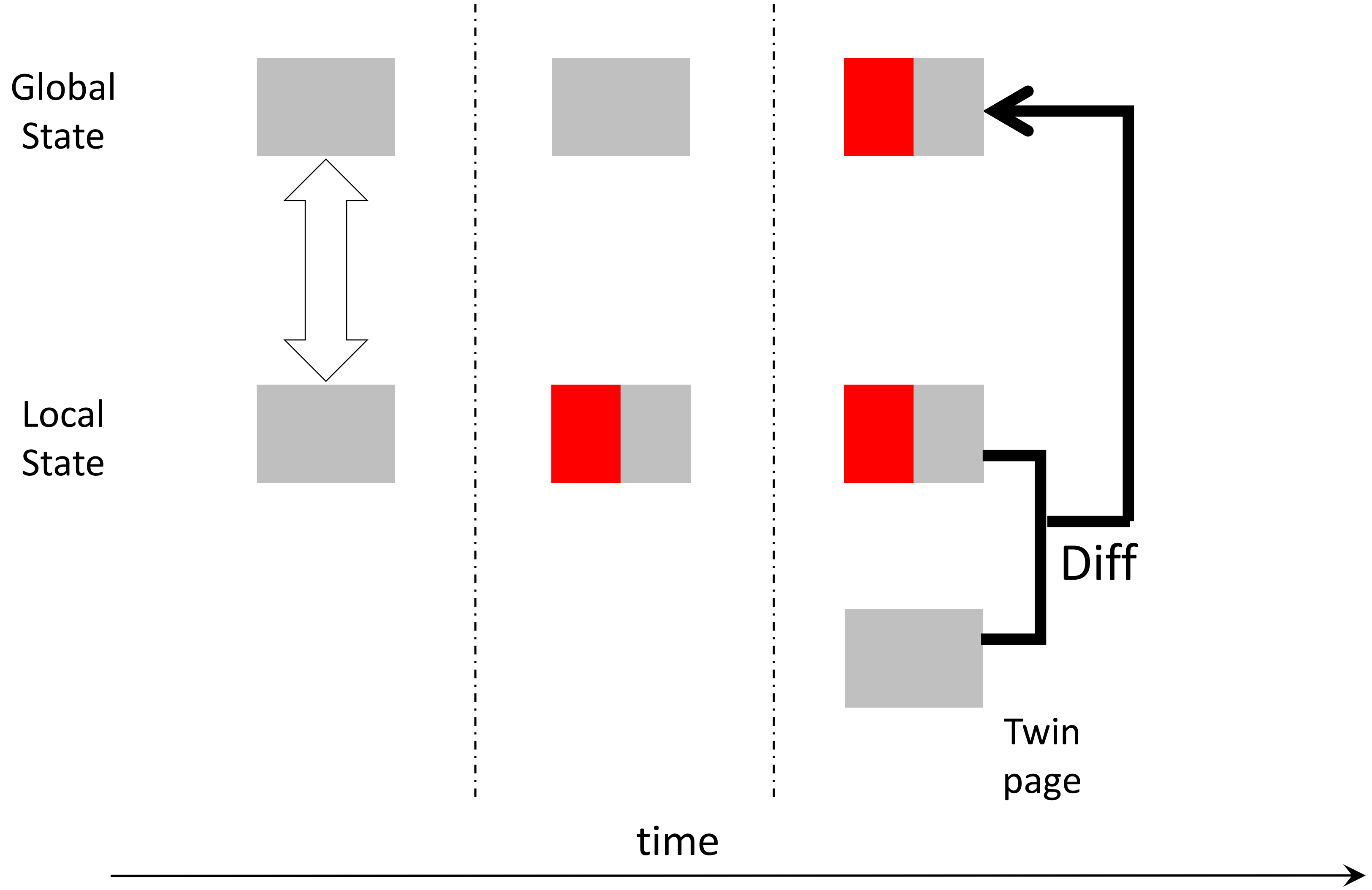
Commit Protocol



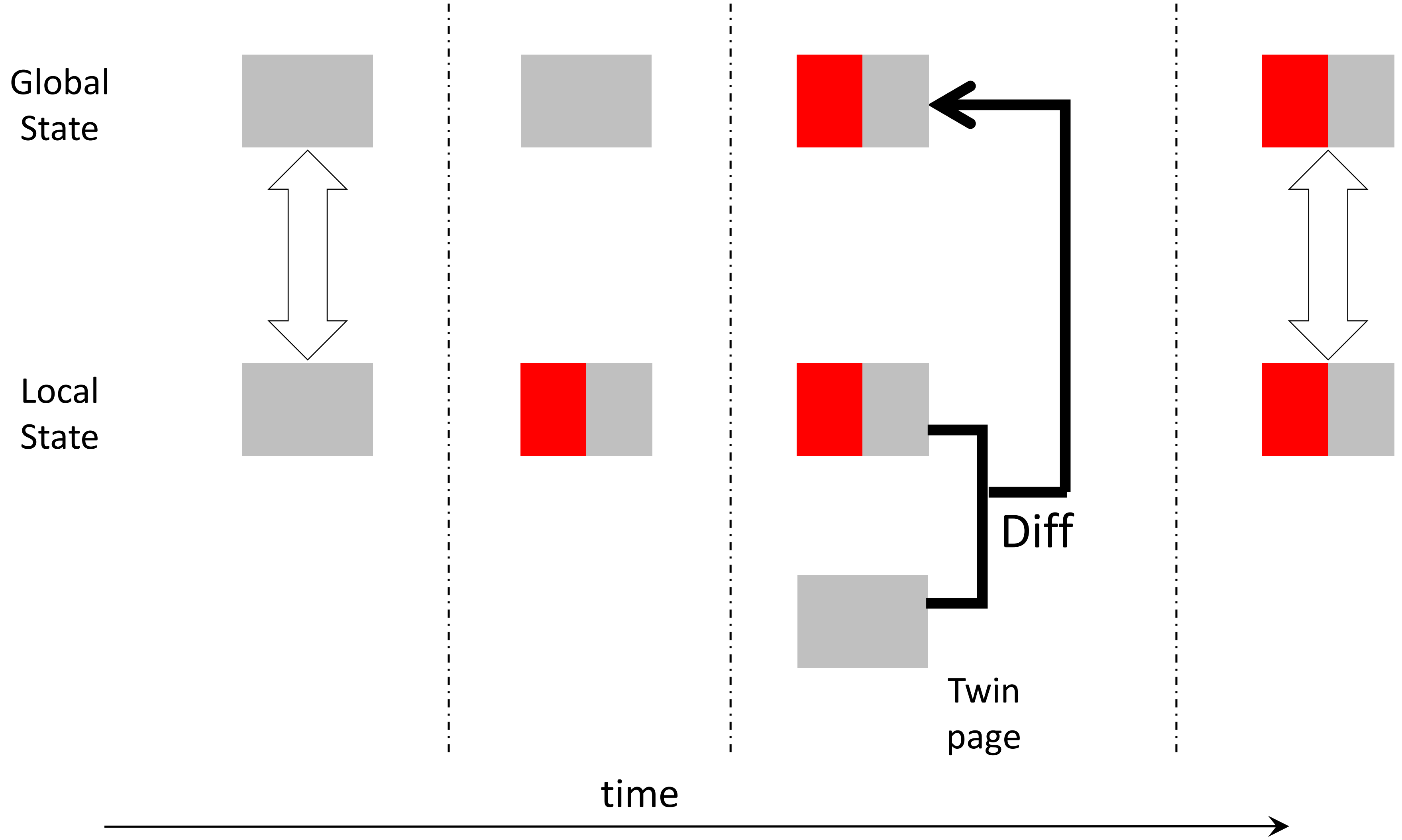
Commit Protocol



Commit Protocol



Commit Protocol



Commit Protocol

- During commit, DTHREADS compare the local copy with a “twin” copy of the original shared page
  - Writes back only the different bytes
  - First thread can copy back the whole page
- Private pages are released at the end of the serial phase

Deterministic Execution Example with DTHREADS

```
int a = 0;
int b = 0;
int main() {
  spawn(thread1);
  spawn(thread2);
  print(a, b);
}
```

DTHREADS will always generate (1, 1) as the output, how?

```
void thread1() {
  if (b == 0) {
    a = 1;
  }
}

void thread2() {
  if (a == 0) {
    b = 1;
  }
}
```

DTHREADS Example Execution

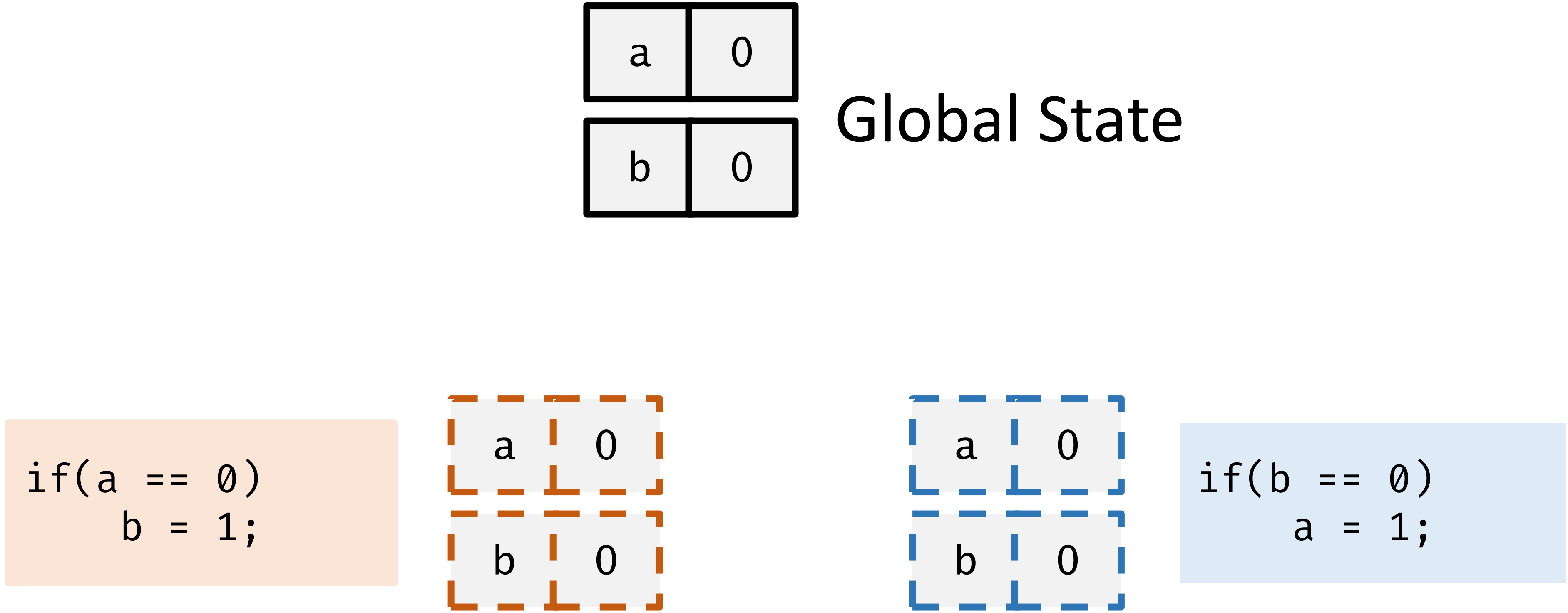
a	0
b	0

Global State

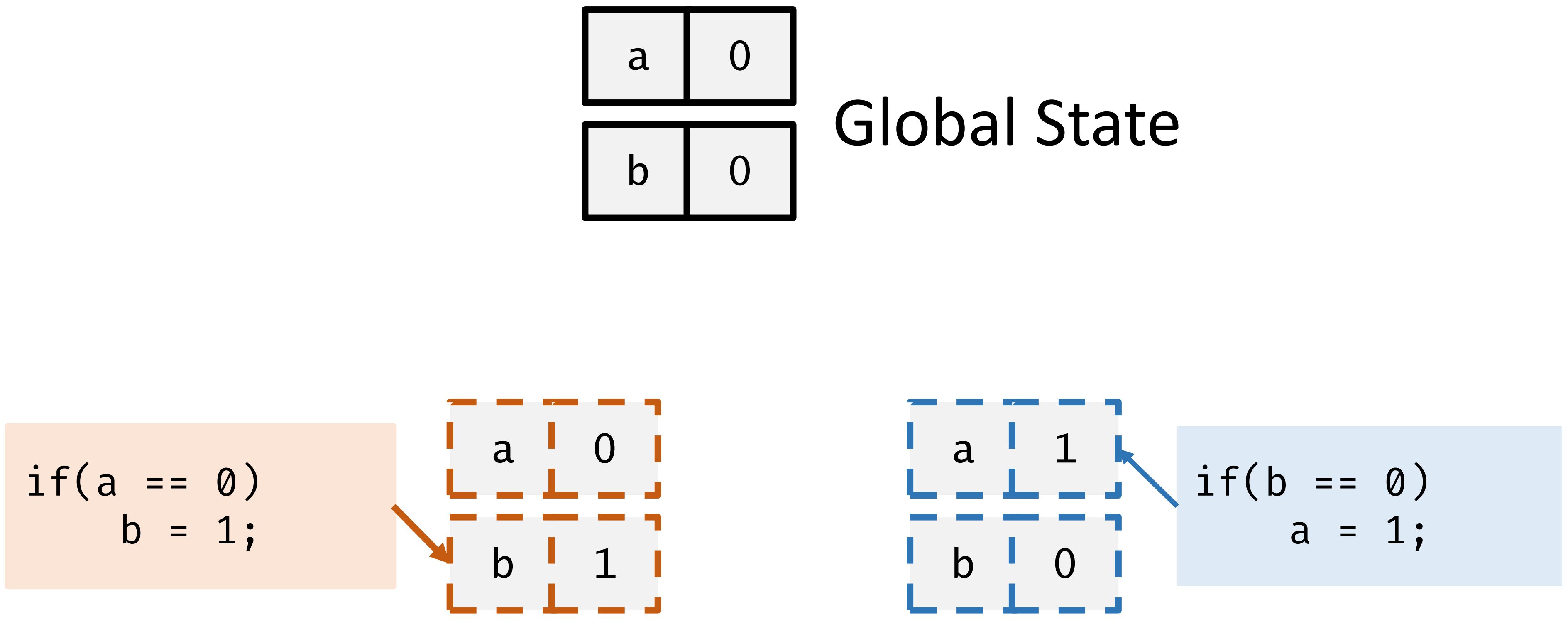
```
if(a == 0)
    b = 1;
```

```
if(b == 0)
    a = 1;
```

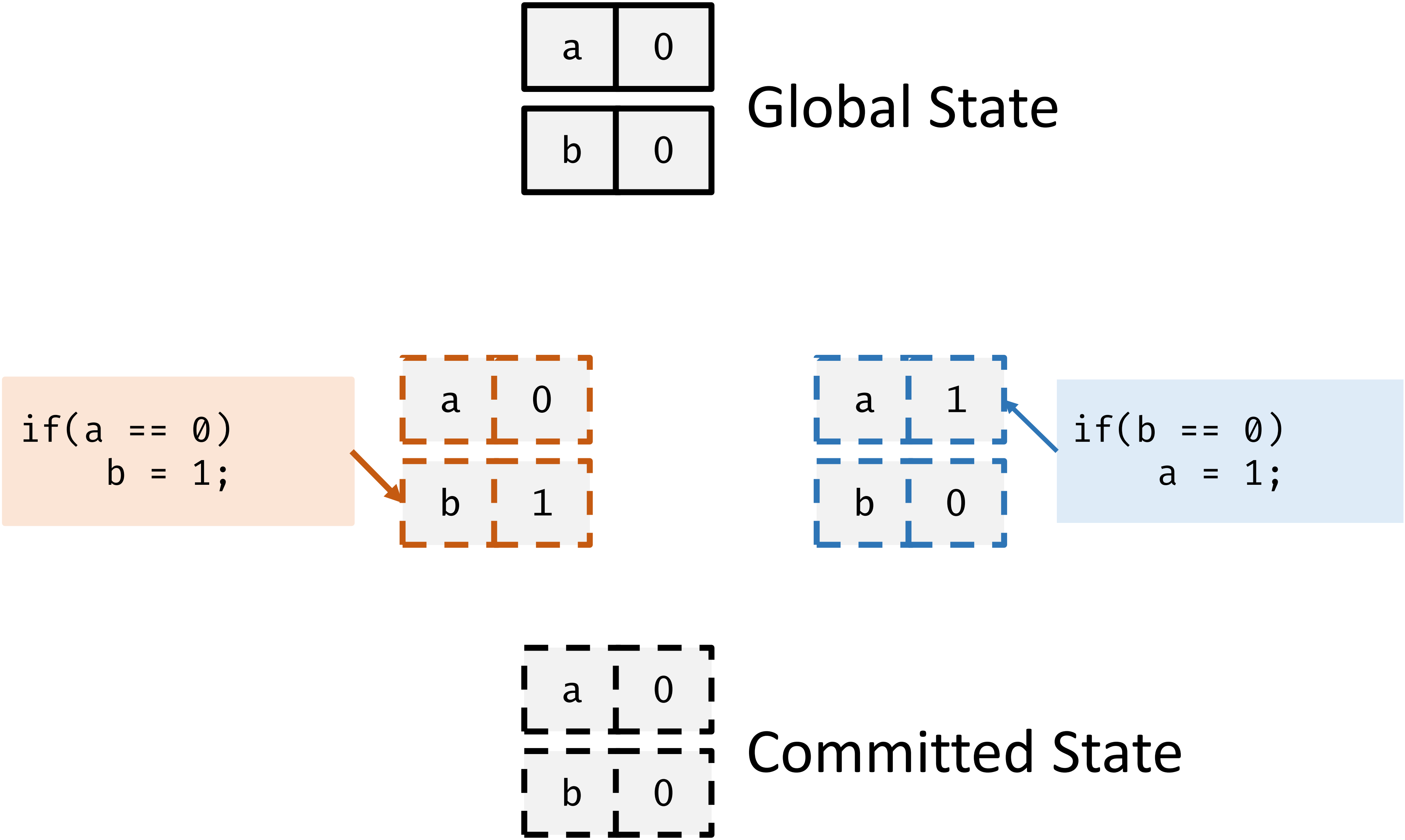
DTHREADS Example Execution



DTHREADS Example Execution



DTHREADS Example Execution



DTHREADS Example Execution

a	0
b	0

Global State

```
if(a == 0)
    b = 1;
```

a	0
b	1

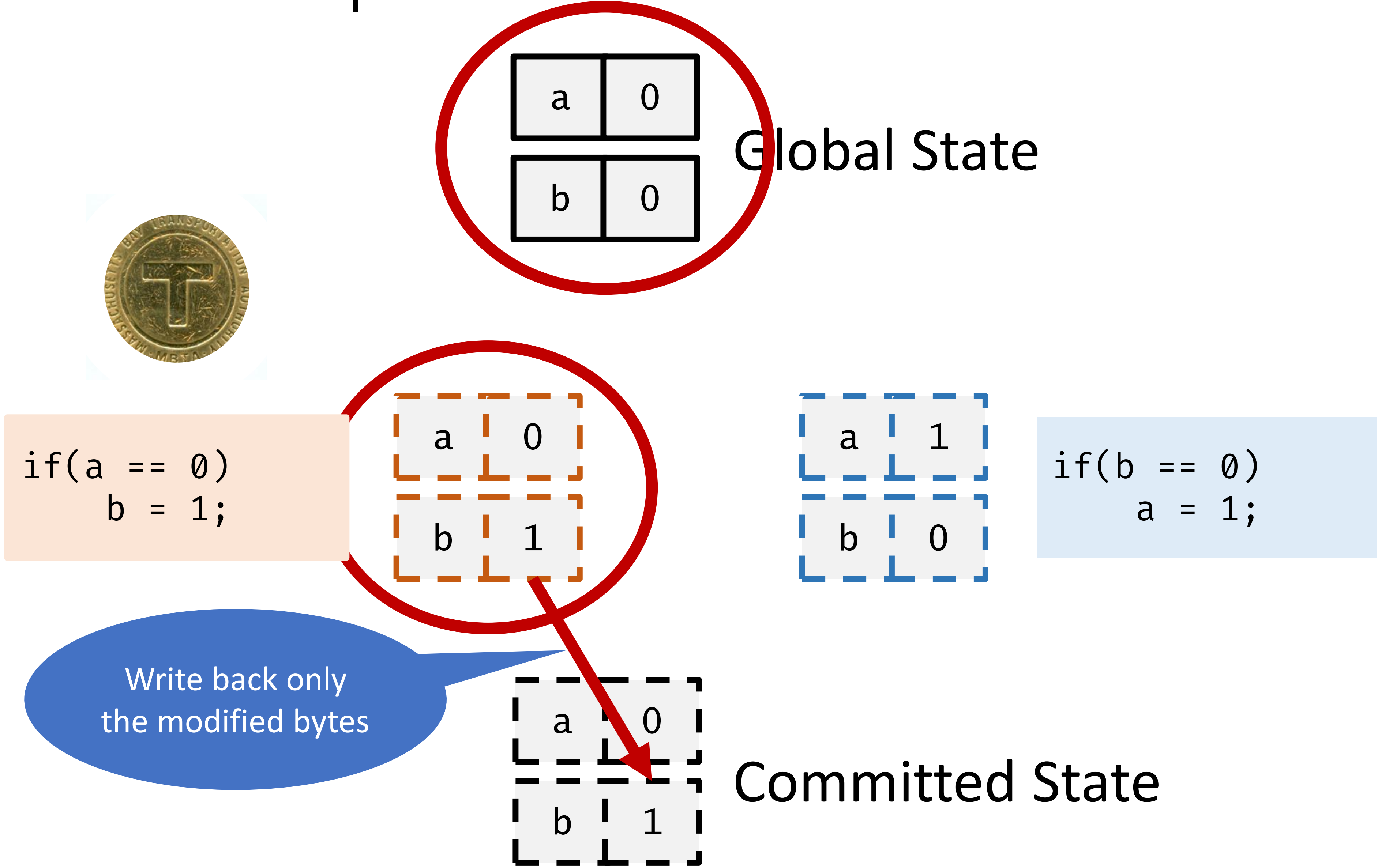
a	1
b	0

```
if(b == 0)
    a = 1;
```

a	0
b	0

Committed State

DTHREADS Example Execution



DTHREADS Example Execution

a	0
b	0

Global State

```
if(a == 0)
    b = 1;
```

a	0
b	1

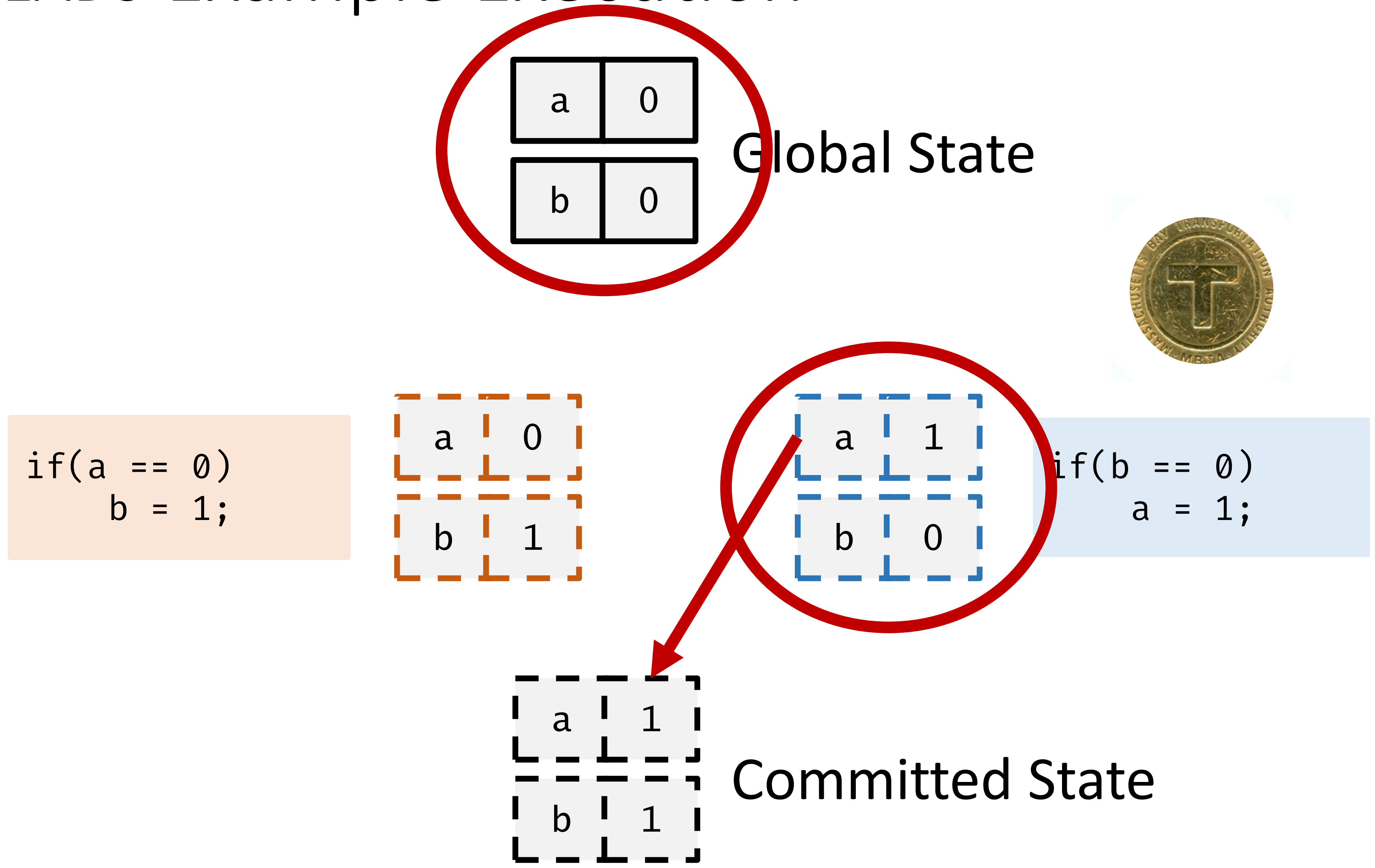
a	1
b	0

```
if(b == 0)
    a = 1;
```

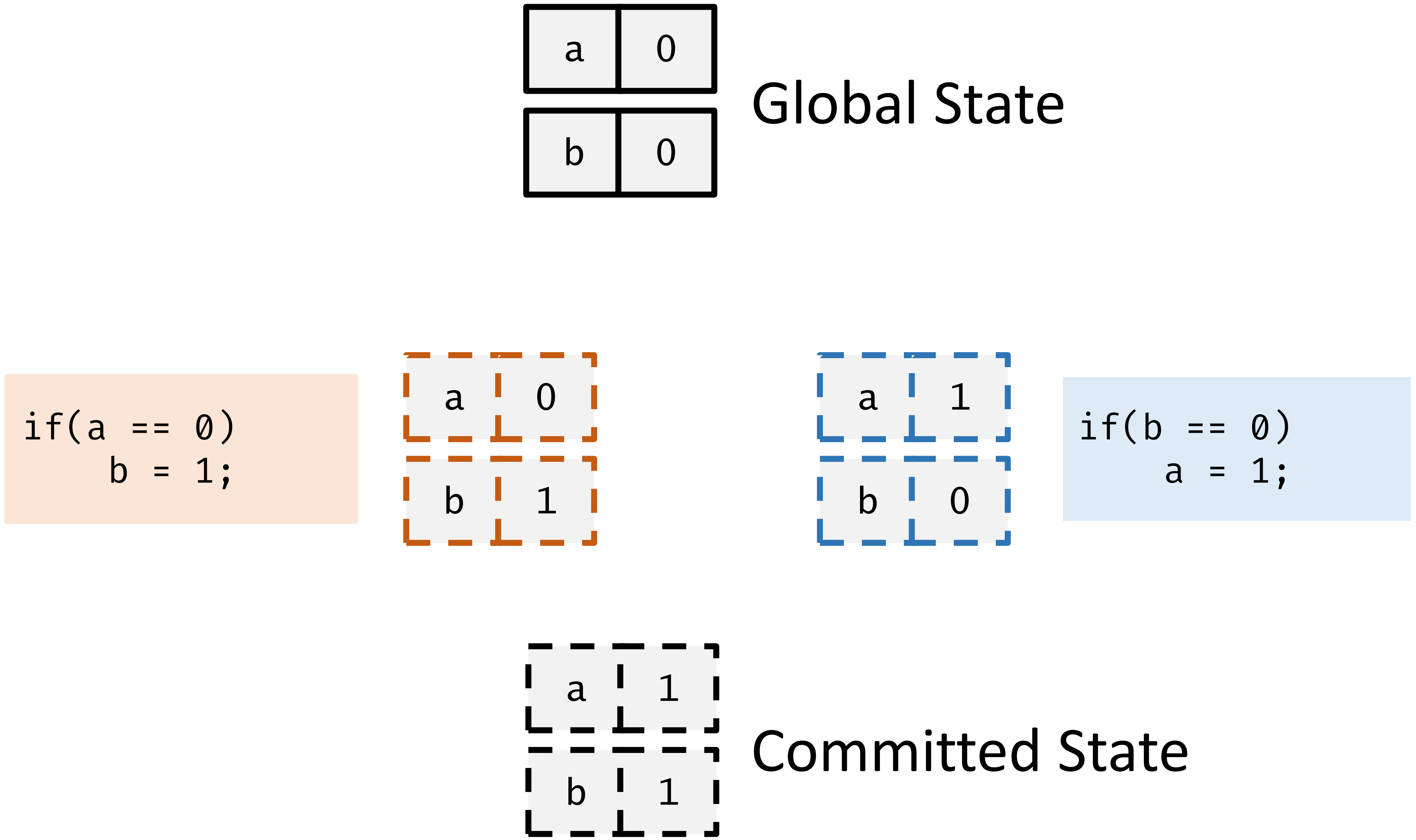
a	0
b	1

Committed State

DTHREADS Example Execution



DTHREADS Example Execution



DTHREADS Example Execution

a	1
b	1

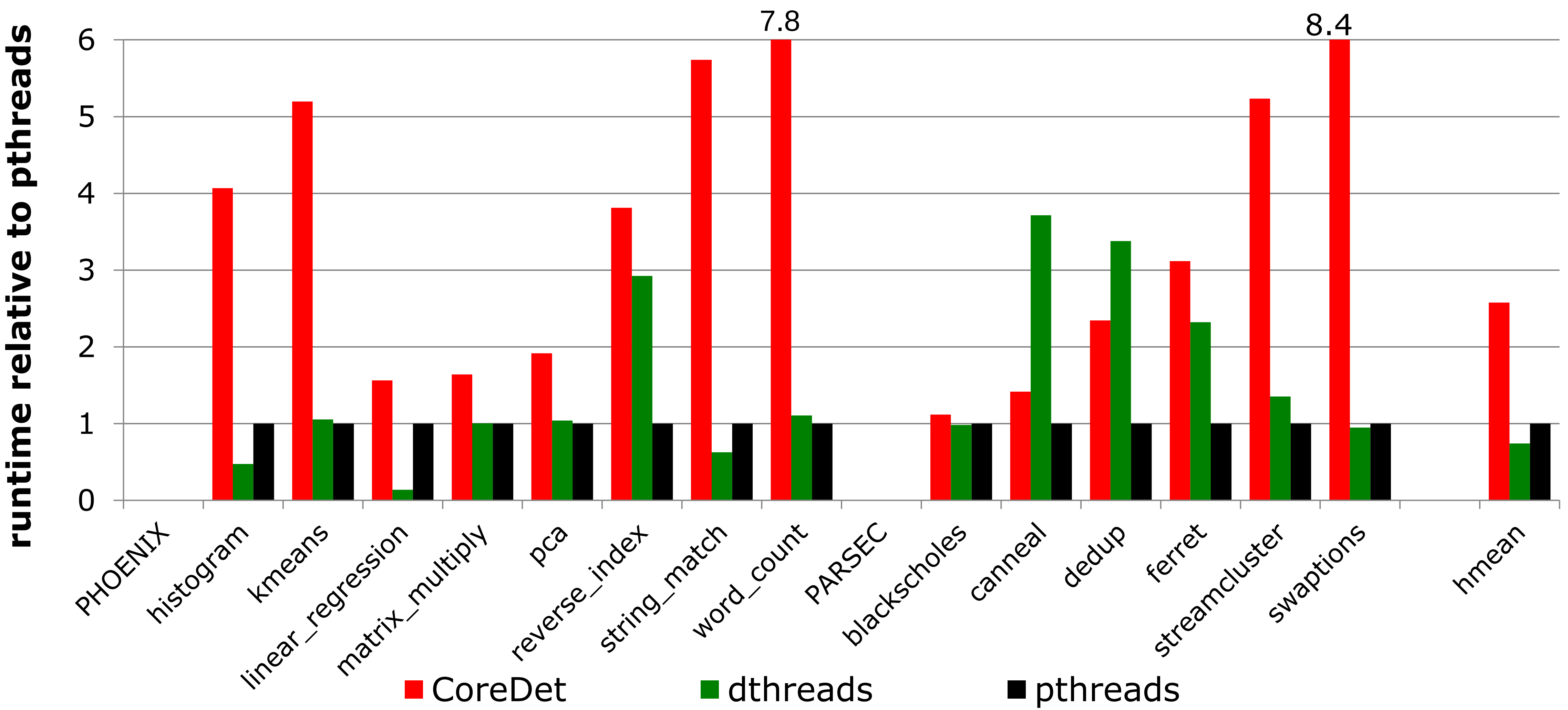
Global State

```
if(a == 0)
    b = 1;
```

a	0

b	0

```
if(b == 0)
    a = 1;
```



Generally as fast or faster than pthreads

References

- D. Hovemeyer and W. Pugh. Finding Concurrency Bugs in Java. PODC 2004.
- S. Burckhardt et al. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. ASPLOS 2010.
- M. Musuvathi. Randomized Algorithms for Concurrency Testing. CONCUR 2017.
- S. Nagarakatte et al. Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection. PLDI 2012.
- M. Musuvathi et al. Finding and Reproducing Heisenbugs in Concurrent Programs. OSDI 2008.
- S. Burckhardt et al. CHESS: Analysis and Testing of Concurrent Programs. PLDI 2009 Tutorial.
- T. Liu et a. DTHREADS: Efficient and Deterministic Multithreading. SOSP 2011.