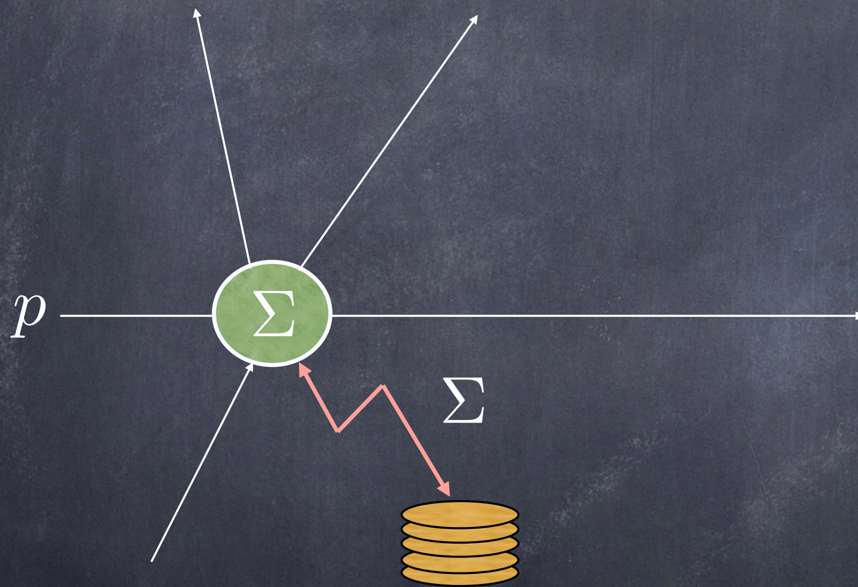
Some like it hot

- Hot Backups process information from the primary as soon as they receive it
- Cold Backups log information received from primary, and process it only if primary fails
- Rollback Recovery implements cold backups cheaply:
  - the primary logs directly to stable storage the information needed by backups
  - if the primary crashes, a newly initialized process is given content of logs—backups are generated "on demand"

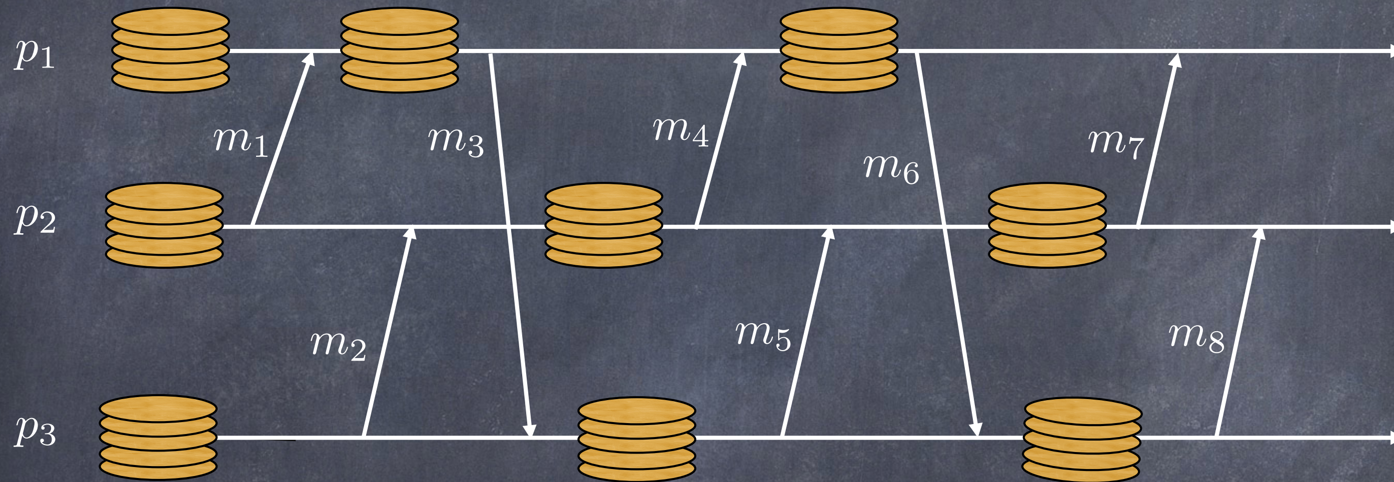
Rollback-Recovery

Uncoordinated Checkpointing

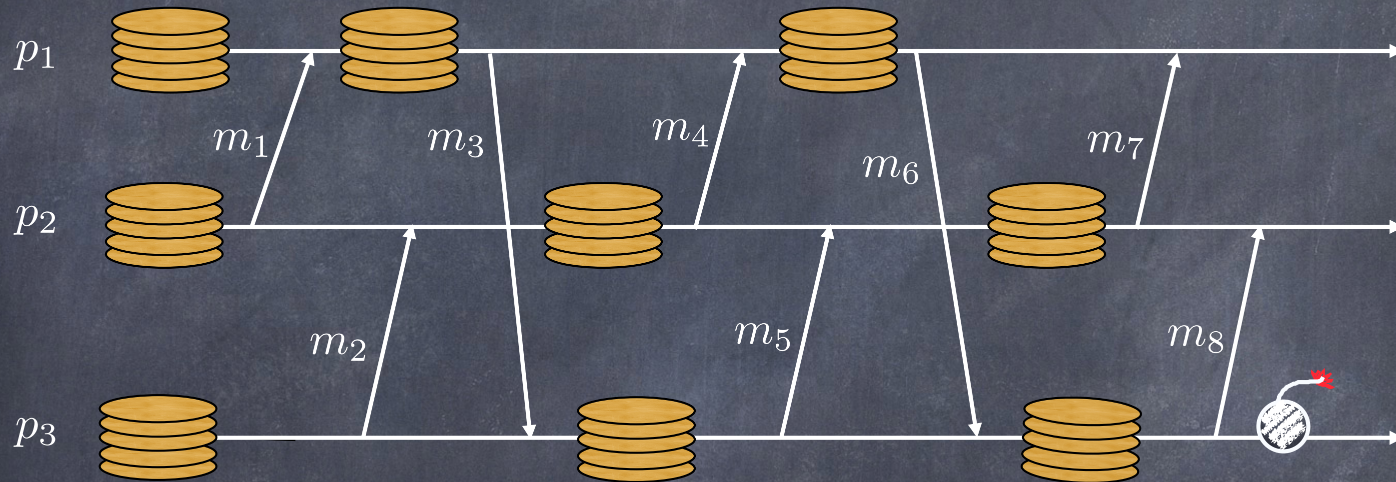


- Easy to understand
- No synchronization overhead
- Flexible
  - can choose when to checkpoint
- To recover from a crash:
  - go back to last checkpoint
  - restart

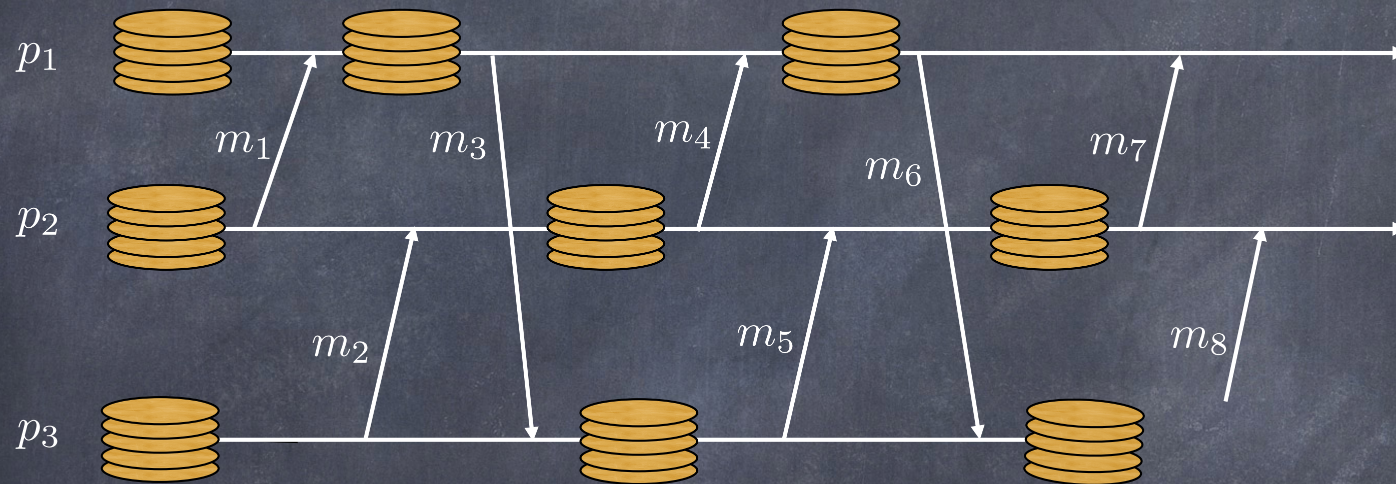
The Domino Effect



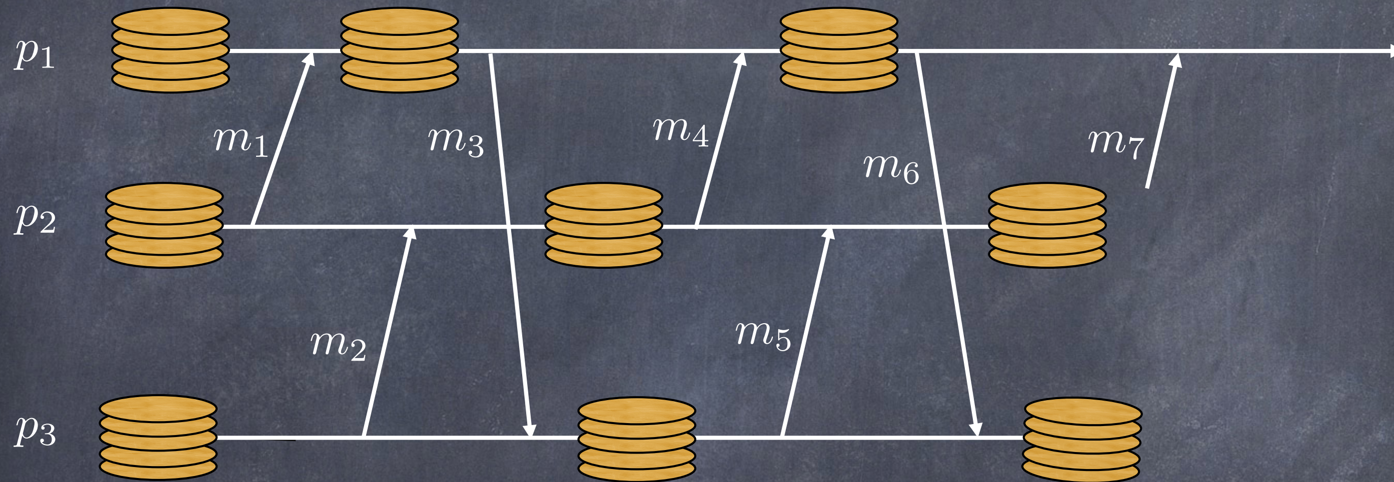
The Domino Effect



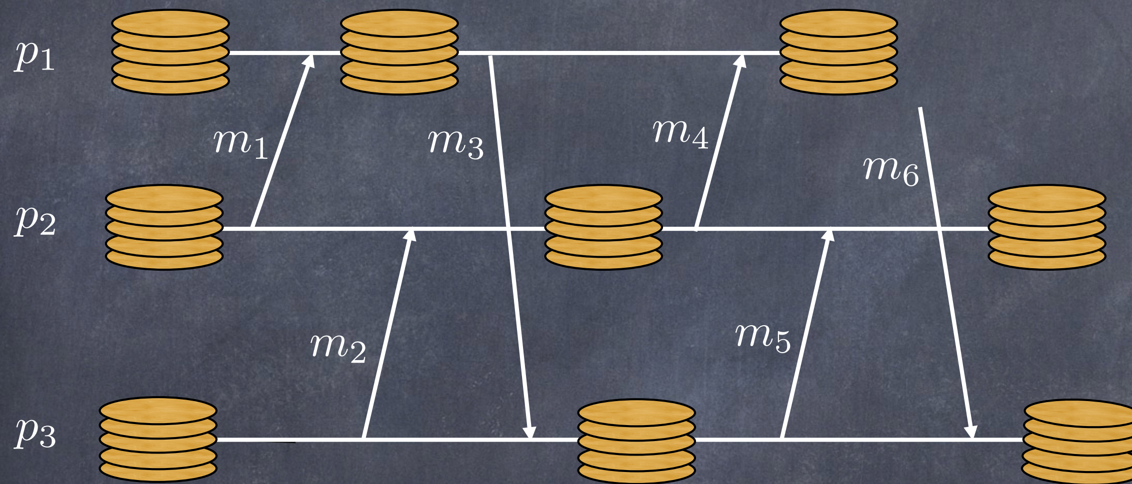
The Domino Effect



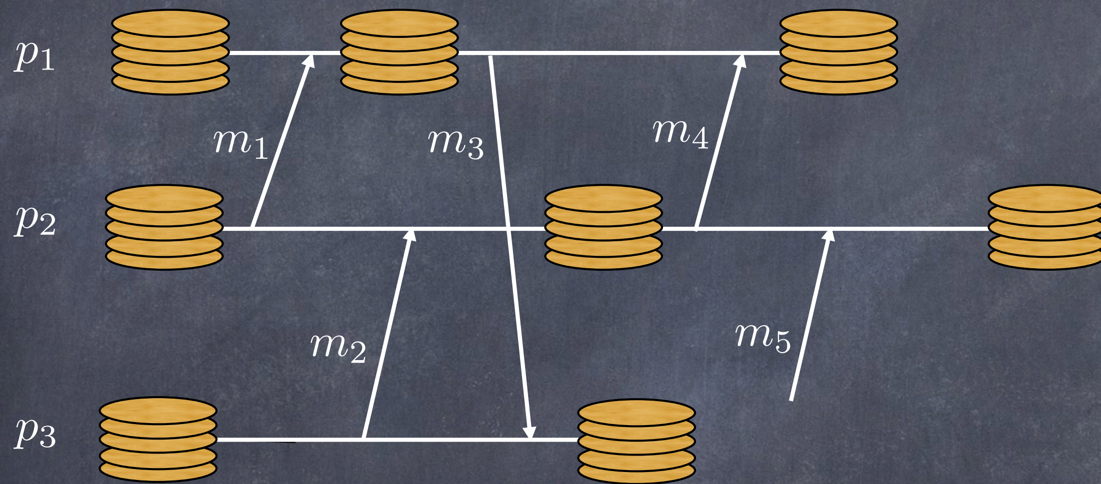
The Domino Effect



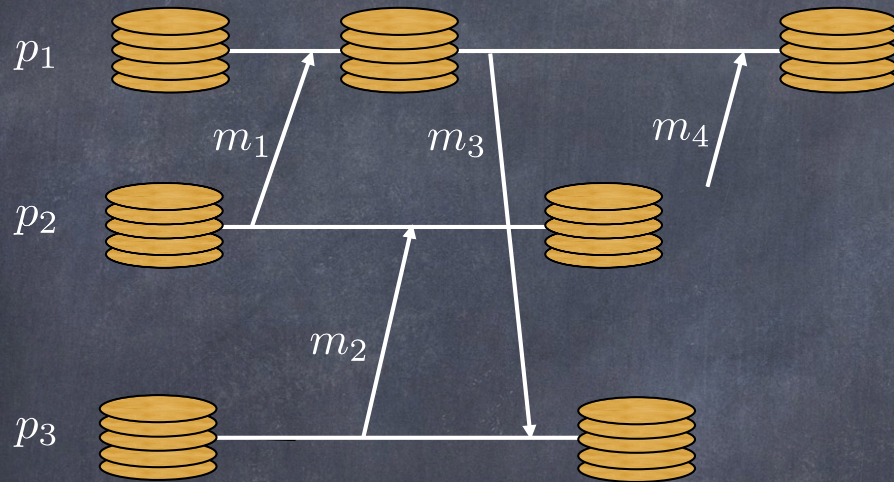
The Domino Effect



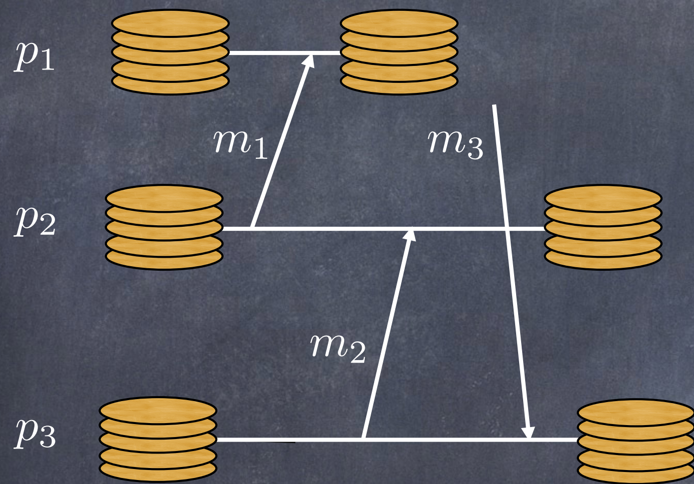
The Domino Effect



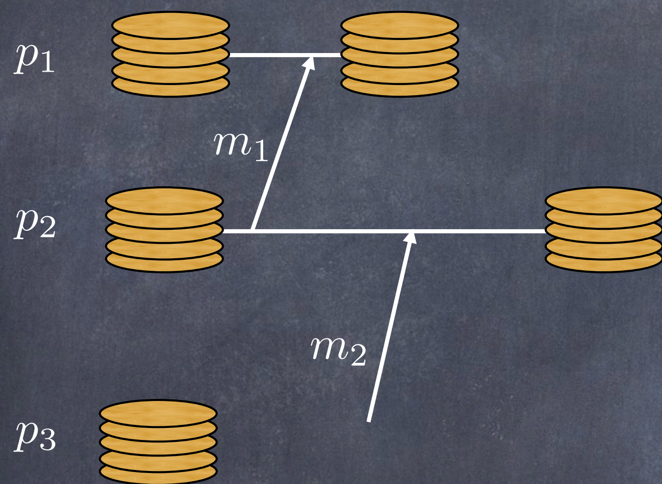
The Domino Effect



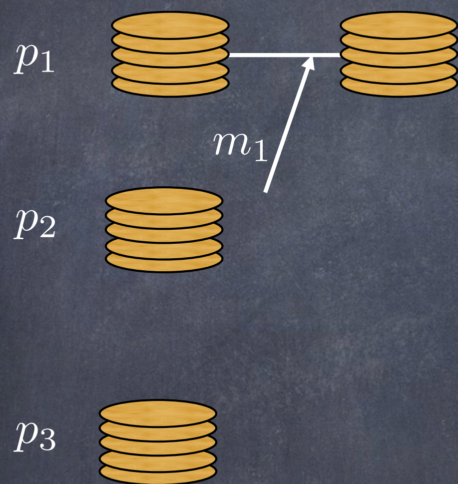
The Domino Effect



The Domino Effect



The Domino Effect



The Domino Effect

p_1

p_2

p_3

How to Avoid the Domino Effect

Coordinated Checkpointing

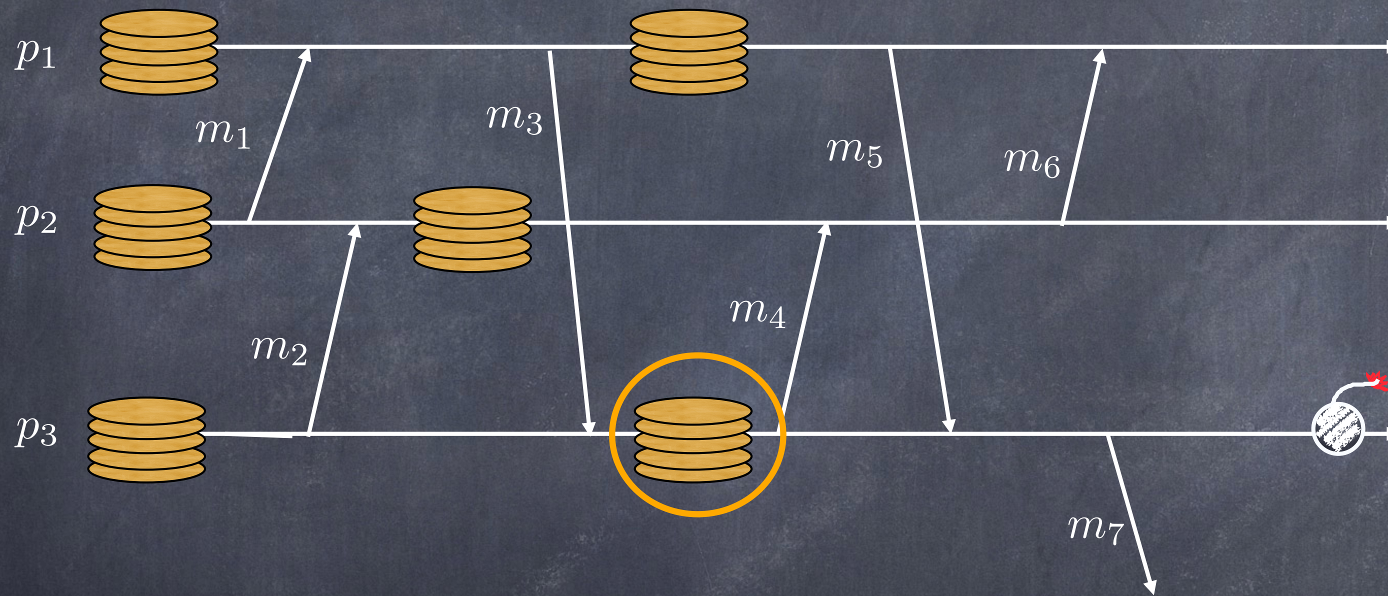
- Easy Garbage Collection
- No independence
- Synchronization Overhead

Communication Induced Checkpointing

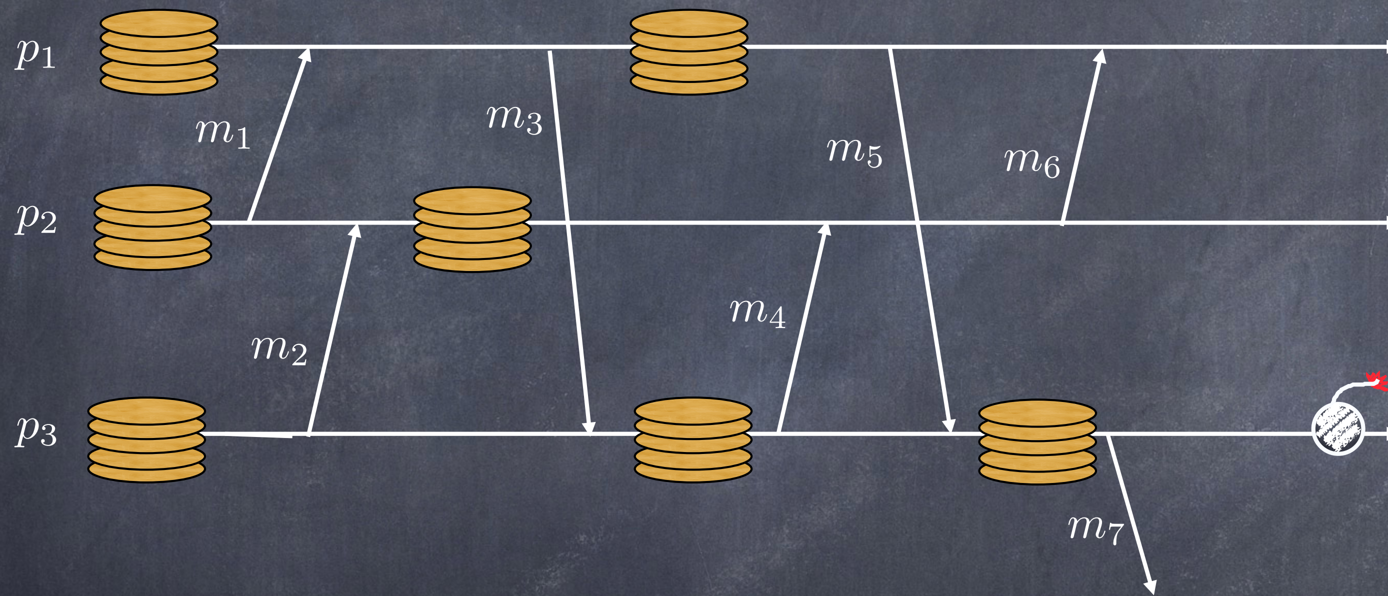
detect dangerous communication patterns and checkpoint appropriately

- Less synchronization
- Less independence

The Output Commit Problem

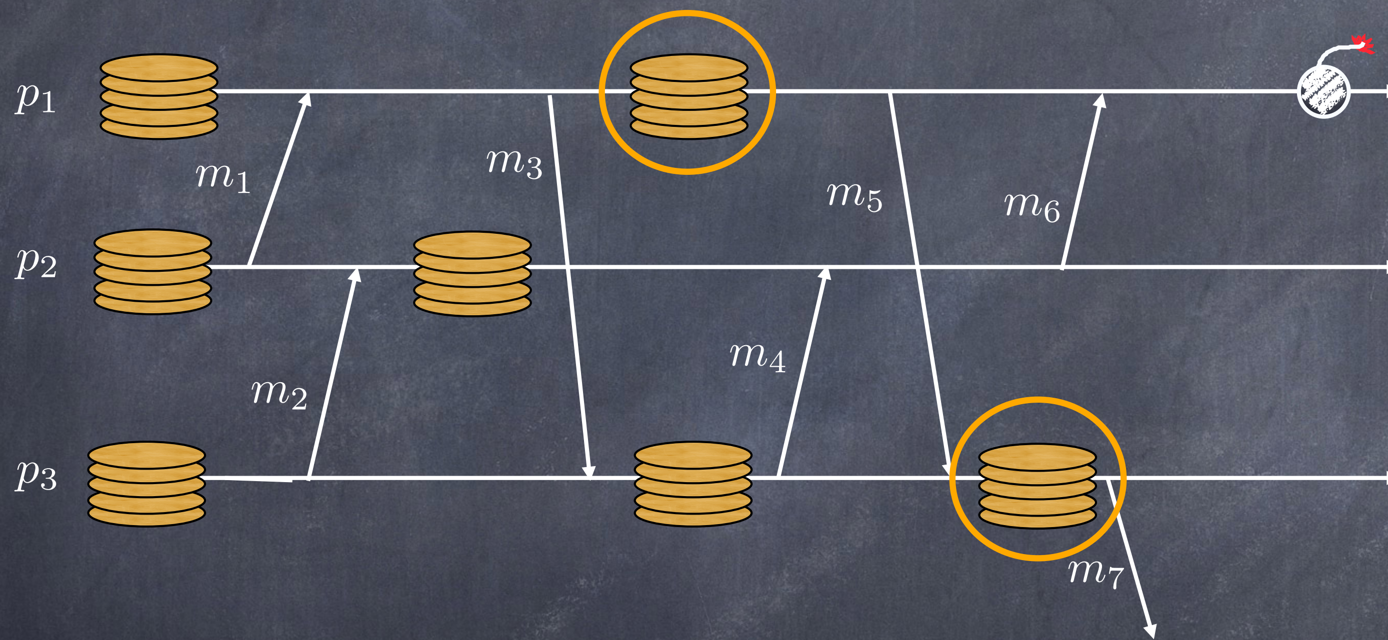


The Output Commit Problem



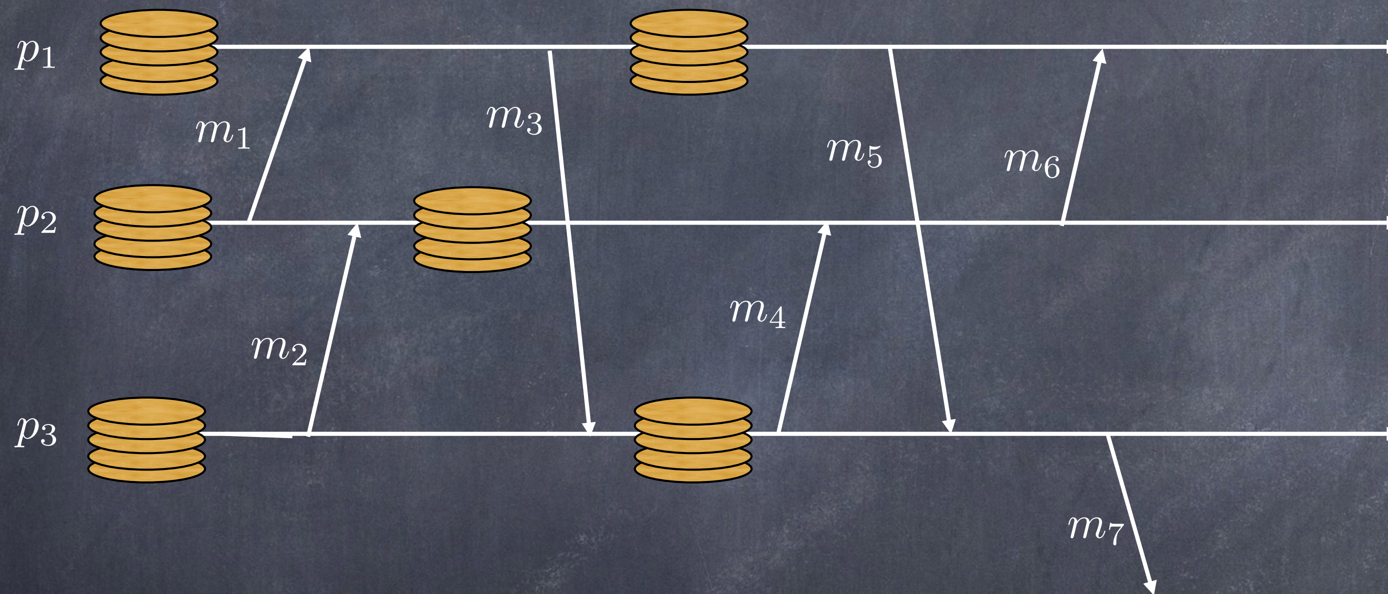
External Environment

The Output Commit Problem



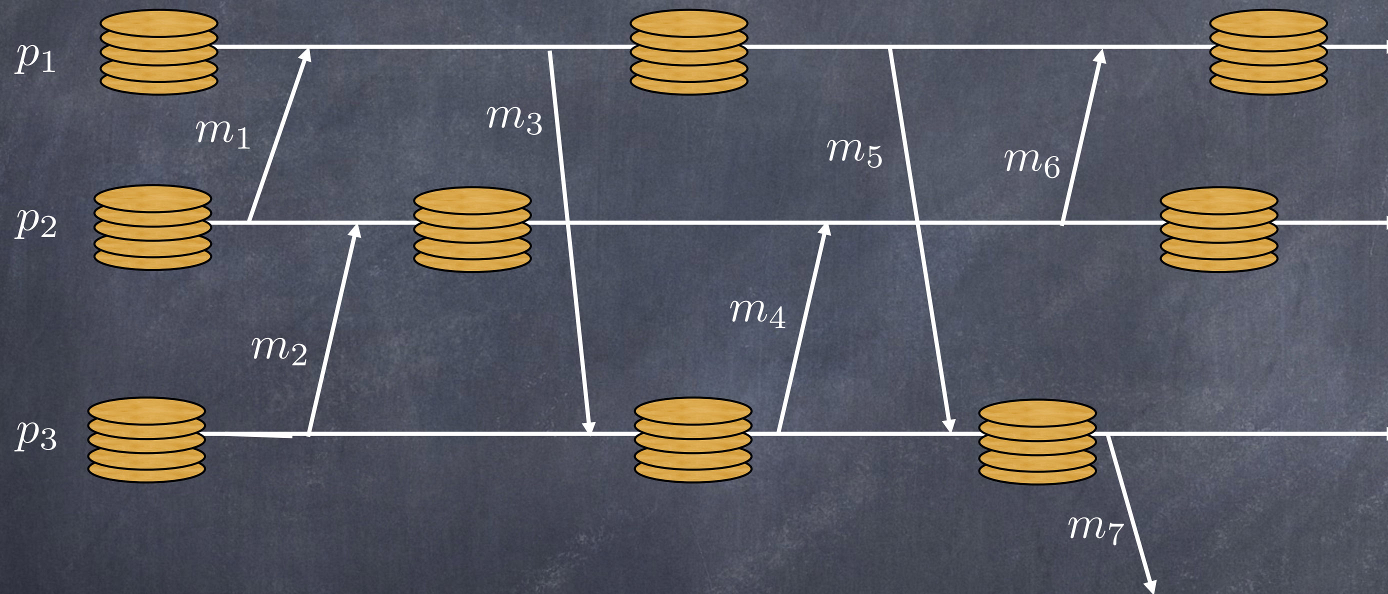
External Environment

The Output Commit Problem



External Environment

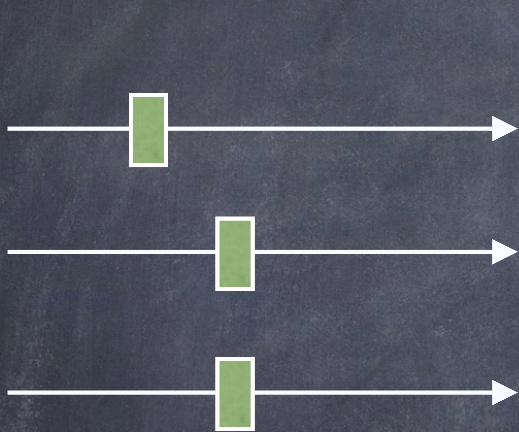
The Output Commit Problem



- Coordinated checkpoint for every output commit
- High overhead if frequent I/O with external environment

External Environment

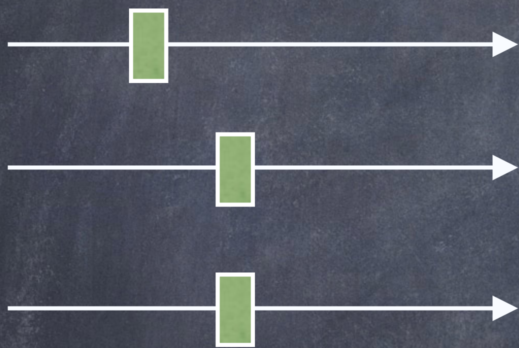
Distributed Checkpointing at a Glance



Independent

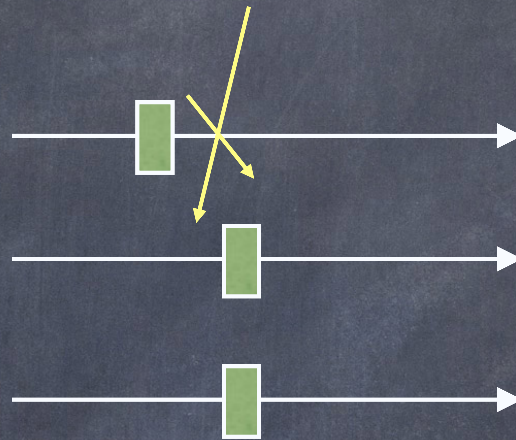
+ Simplicity
+ Autonomy
+ Scalability
- Domino effect

Distributed Checkpointing at a Glance



Independent

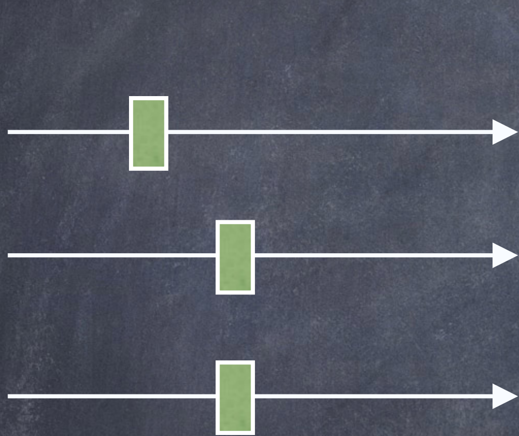
- \+ Simplicity
- \+ Autonomy
- \+ Scalability
- \- Domino effect

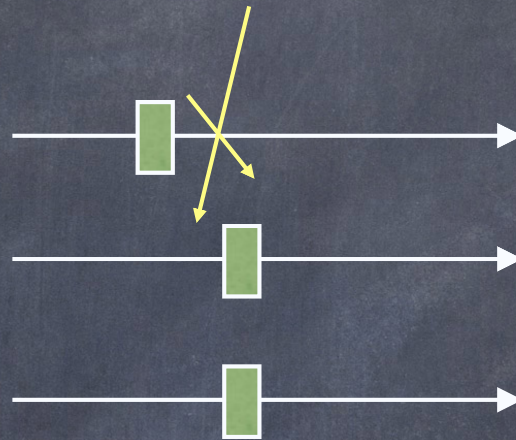


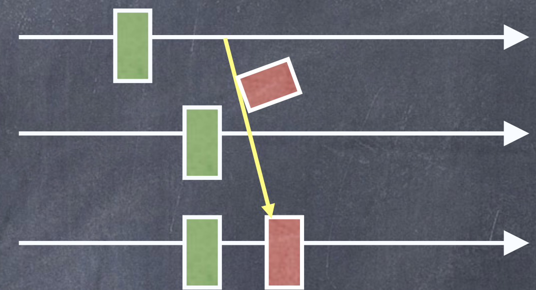
Coordinated

- \+ Consistent states
- \+ Good performance
- \+ Garbage Collection
- \- Scalability

Distributed Checkpointing at a Glance







Independent	Coordinated	Communication-induced
+ Simplicity	+ Consistent states	+ Consistent states
+ Autonomy	+ Good performance	+ Autonomy
+ Scalability	+ Garbage Collection	+ Scalability
- Domino effect	- Scalability	- None is true

Message Logging

- Can avoid domino effect
- Works with coordinated checkpoint
- Works with uncoordinated checkpoint
- Can reduce cost of output commit
- More difficult to implement

How It Works

To tolerate crash failures:

- periodically checkpoint application state;
- log on stable storage determinants of non-deterministic events executed after checkpointed state.

Recovery:

- restore latest checkpointed state;
- replay non-deterministic events according to determinants

Logging Determinants

Determinants for message delivery events:

message $m = \langle m.dest, m.rsn, m.data \rangle$

Logging Determinants

Determinants for message delivery events:

message $m = \langle m.dest, m.rsn, m.data \rangle$

Or alternatively:

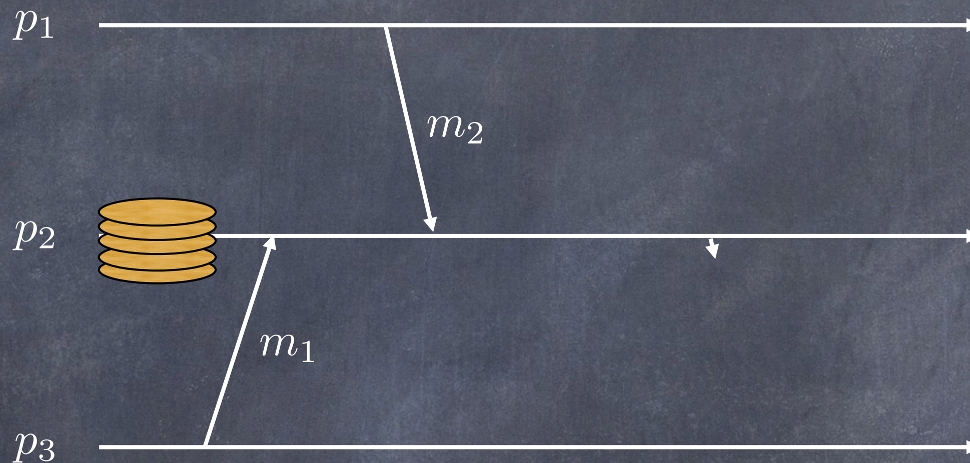
message $m = \langle m.dest, m.rsn, \underline{m.source, m.ssn} \rangle$

Pessimistic Logging

Logged determinants

$\langle m_1, p_2, 1 \rangle$

$\langle m_2, p_2, 2 \rangle$

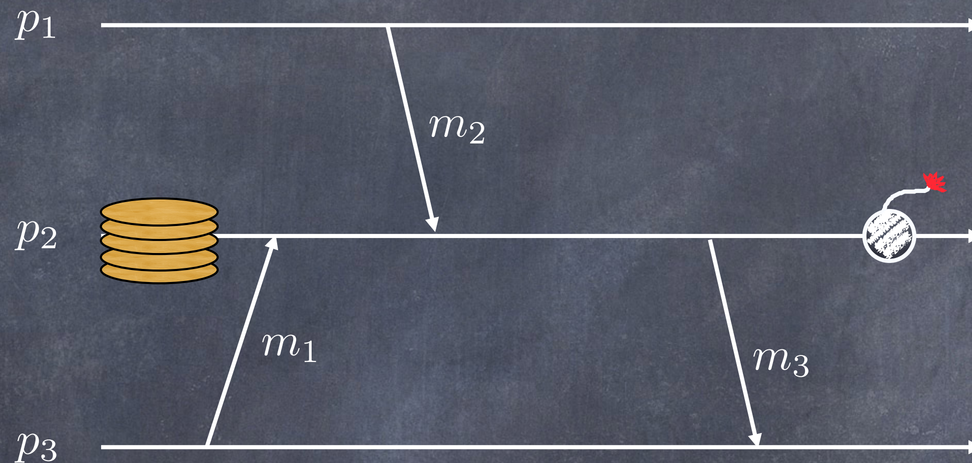


Pessimistic Logging

Logged determinants

$\langle m_1, p_2, 1 \rangle$

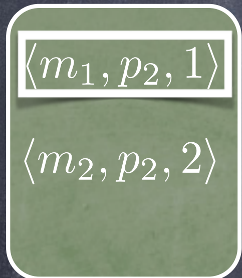
$\langle m_2, p_2, 2 \rangle$

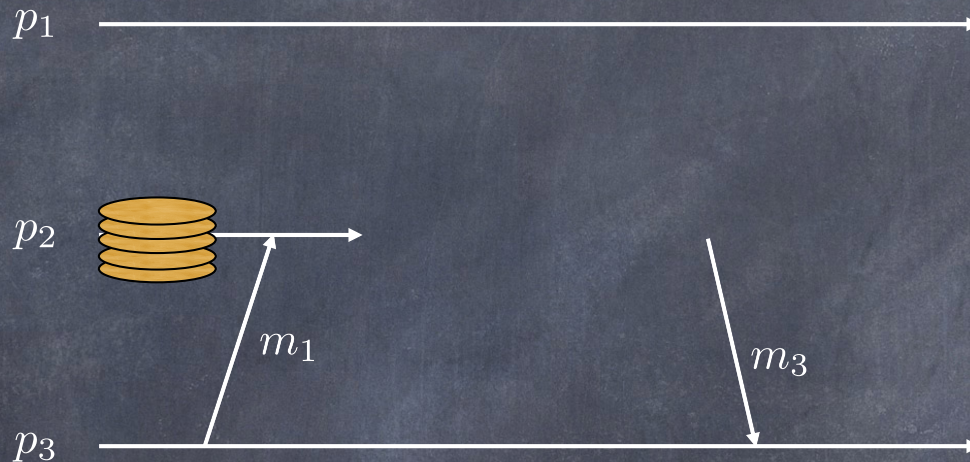


Pessimistic Logging

Pessimistic Logging

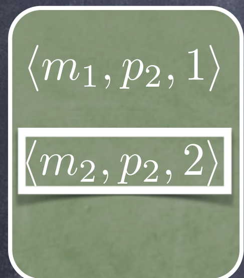
Logged determinants

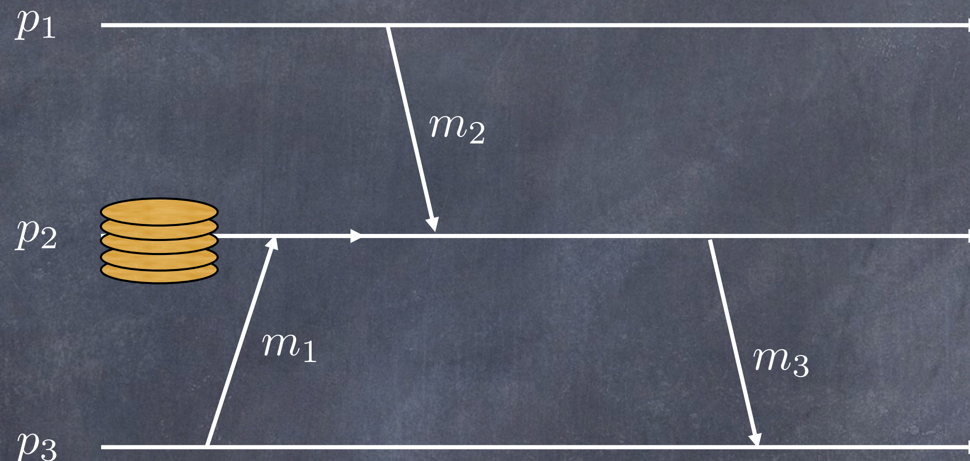




Pessimistic Logging

Logged determinants





Never creates orphans

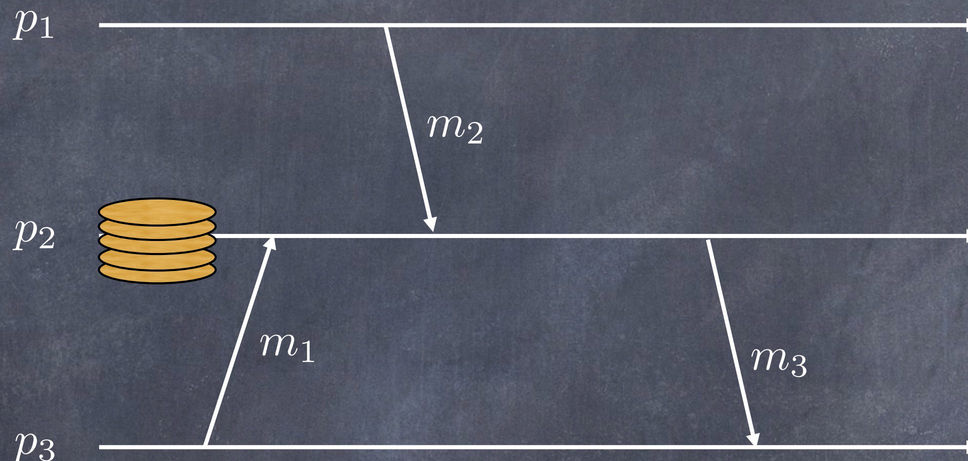
- straightforward recovery
- may incur blocking

Pessimistic Logging

Logged determinants

$\langle m_1, p_2, 1 \rangle$

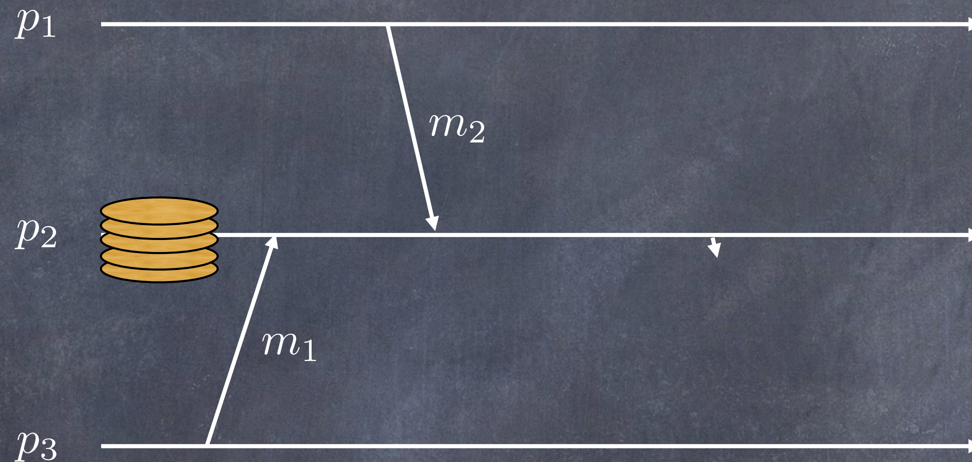
$\langle m_2, p_2, 2 \rangle$



Never creates orphans

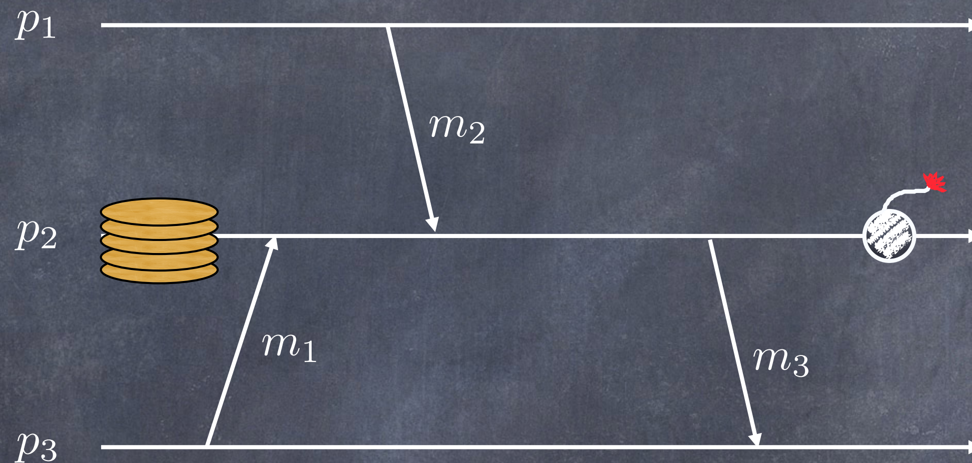
- straightforward recovery
- may incur blocking, while waiting for determinants to become stable

Optimistic Logging



Optimistic Logging

Logged determinants



Optimistic Logging

Kills orphans during recovery

- non-blocking during failure-free executions
- rollback of correct processes
- complex recovery

Causal Logging

- ☑ No blocking in failure-free executions
- ☑ No orphans
- ☑ No additional messages
- ☑ Tolerates multiple concurrent failures
- ☑ Keeps determinant in volatile memory
- ☑ Localized output commit

Preliminary Definitions

Given a message m sent from m.source to m.dest,

$Depend(m)$ $$\left\{ p \in P \,\middle|\, \begin{array}{l} \vee(p = m.dest) \text{ and } p \text{ delivered } m \\ \vee(\exists e_p : (deliver_{m.dest}(m) \rightarrow e_p)) \end{array} \right\}$$

$Log(m)$ set of processes with a copy of the determinant of m in their volatile memory

p orphan of a set C of crashed processes:

$$(p \notin C) \wedge \exists m : (Log(m) \subseteq C \wedge p \in Depend(m))$$

The "No-Orphans" Consistency Condition

No orphans after crash C if:

$$\forall m : (Log(m) \subseteq C) \Rightarrow (Depend(m) \subseteq C)$$

No orphans after any C if:

$$\forall m : (Depend(m) \subseteq Log(m))$$

The Consistency Condition

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

Optimistic and Pessi

No orphans after crash C if:

$$\forall m : (Log(m) \subseteq C) \Rightarrow (Depend(m) \subseteq C)$$

Optimistic weakens it to:

$$\forall m : (Log(m) \subseteq C) \Rightarrow \Diamond(Depend(m) \subseteq C)$$

Optimistic and Pessimistic

No orphans after crash C if:

$$\forall m : (Log(m) \subseteq C) \Rightarrow (Depend(m) \subseteq C)$$

Optimistic weakens it to:

$$\forall m : (Log(m) \subseteq C) \Rightarrow \Diamond(Depend(m) \subseteq C)$$

No orphans after any crash if:

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

Pessimistic strengthens it to:

$$\forall m : (\neg stable(m) \Rightarrow |Depend(m)| \leq 1)$$

Causal Message Logging

No orphans after any crash of size at most f if:

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

Causal Message Logging

No orphans after any crash of size at most f if:

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

Causal strengthens it to:

$$\forall m : \left(\neg stable(m) \Rightarrow \left(\begin{array}{l} \wedge(Depend(m) \subseteq Log(m)) \\ \wedge\Diamond(Depend(m) = Log(m)) \end{array}\right)\right)$$

An Example

Causal Logging:

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

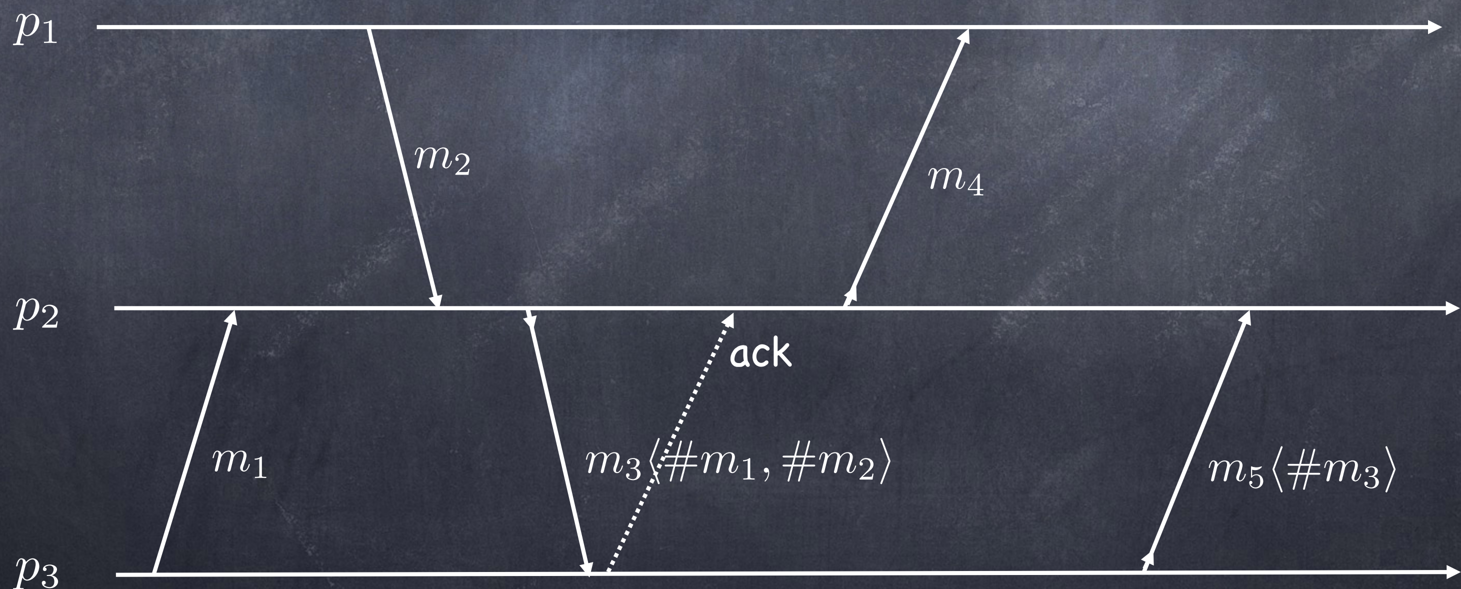
If $f = 1$, $\quad stable(m) \equiv |Log(m)| \geq 2$

An Example

Causal Logging:

$$\forall m : (\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m)))$$

If $f = 1$, $\quad stable(m) \equiv |Log(m)| \geq 2$



Deterministic Replay for Multiprocessors

Why?

Record and reproduce multithreaded executions

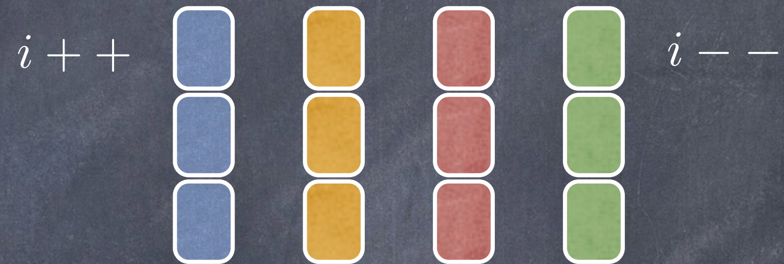
- Debugging
- Program analysis
- Forensics and Intrusion Detection
- Fault tolerance

What is hard?

On a uniprocessor

- Only one thread at a time updates shared memory
  - concurrency is simulated
- Record scheduling decisions
  - fewer than SM accesses

On a multiprocessor



- Threads actually update shared memory concurrently

Instrument each memory operation
10-20X

Reduced logging + offline search
Slow replay

Detect dependencies using memory protection bits
Up to 9x

Hardware support
Custom HW

What to replay?

- Exact reproducibility is hard and expensive…
- … but no need to replay the exact execution
  - Aim for observationally indistinguishable
    - produce same set of states S
    - produce same set of outputs O
    - match a possible execution of the program

When to replay?

- Online: in parallel with the original execution
  - fault tolerance, parallel security check
- Offline: after the original execution has completed
  - debugging, forensics, etc

Online Multiprocessor Replay

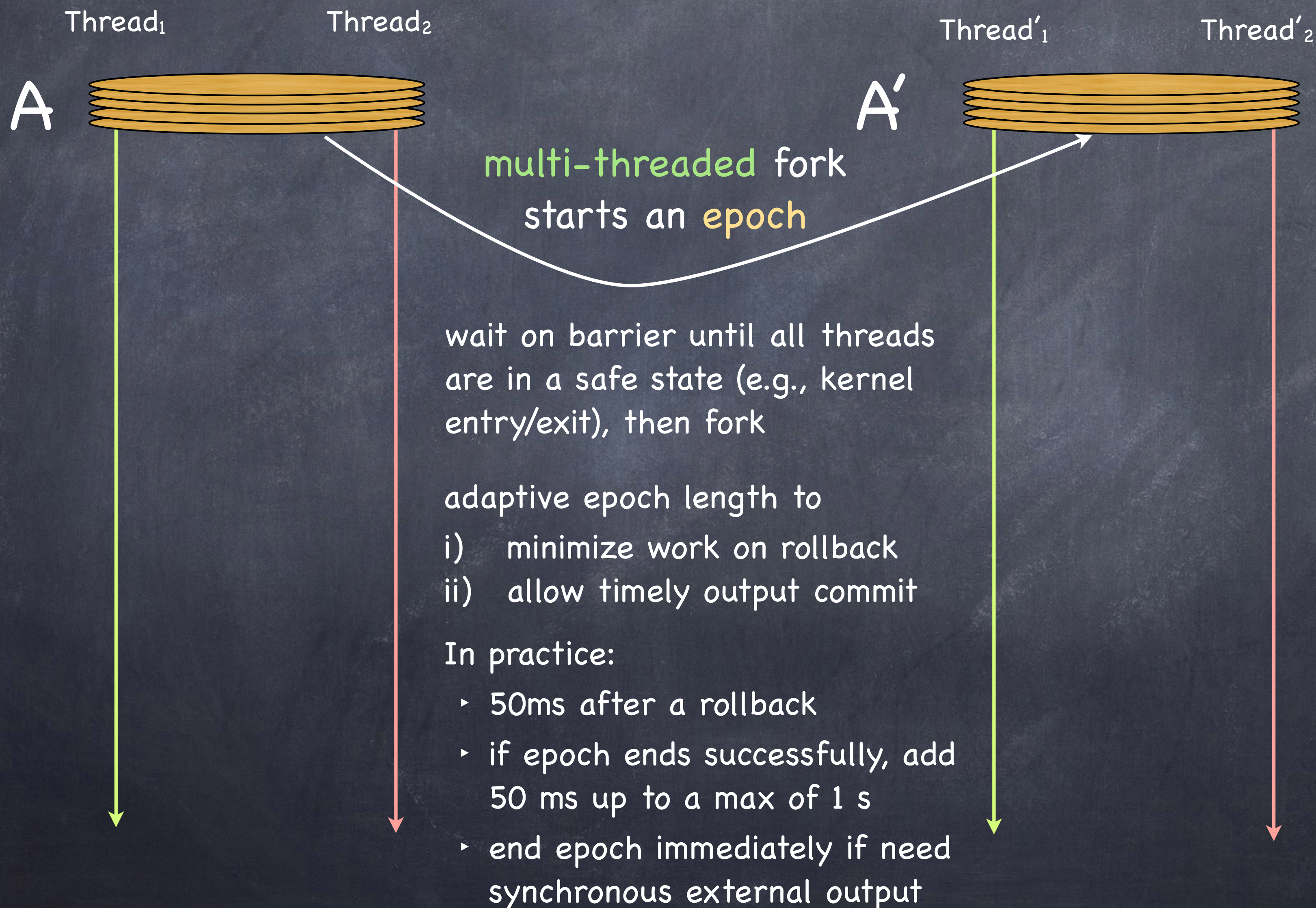
Respec, ASPLOS '10

speculate check

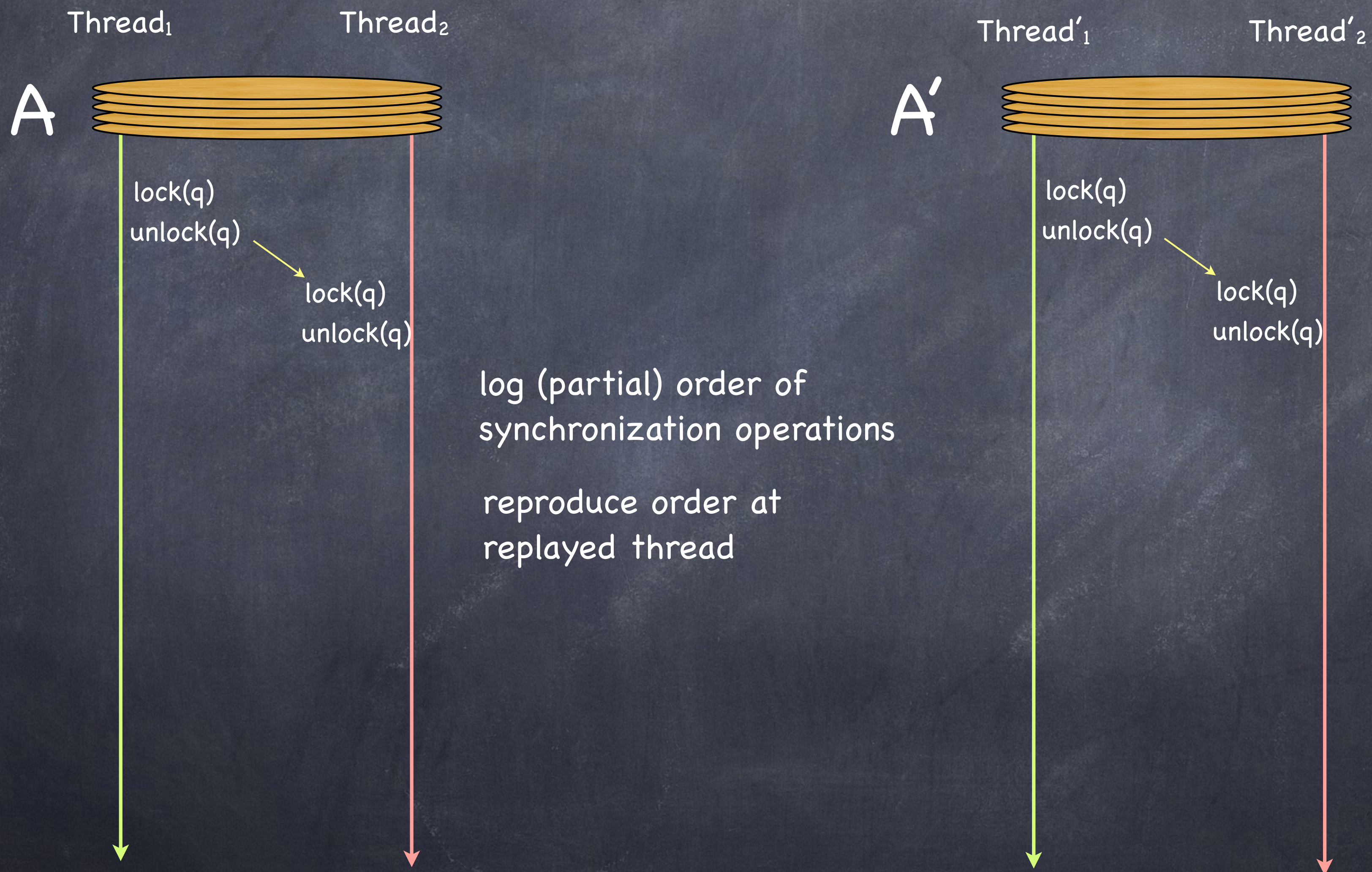
Key idea: "~~trust~~ but ~~verify~~"

1. Speculate execution is data race free
2. Check efficiently for mis-speculation
3. On mis-speculation, rollback and retry

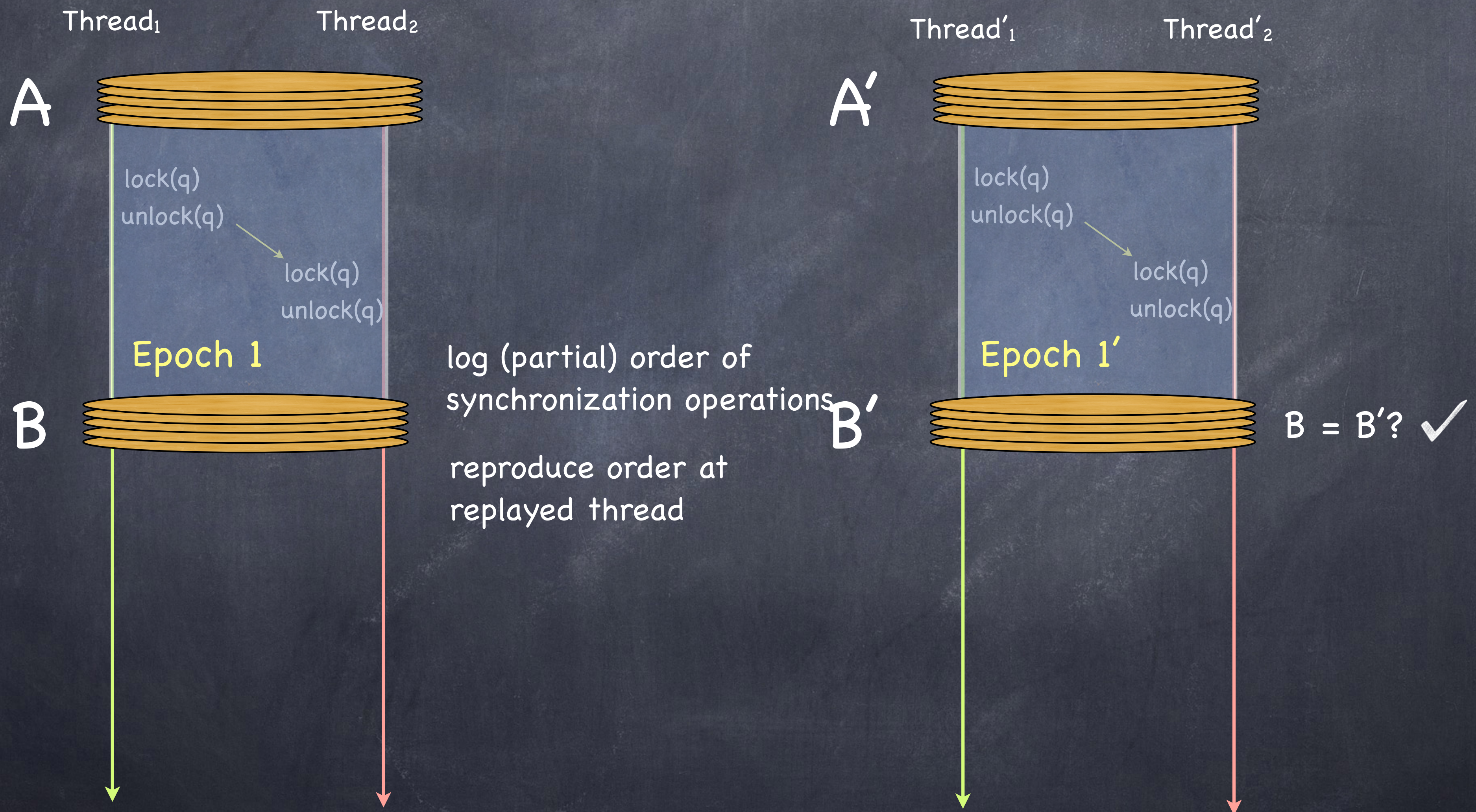
Speculate



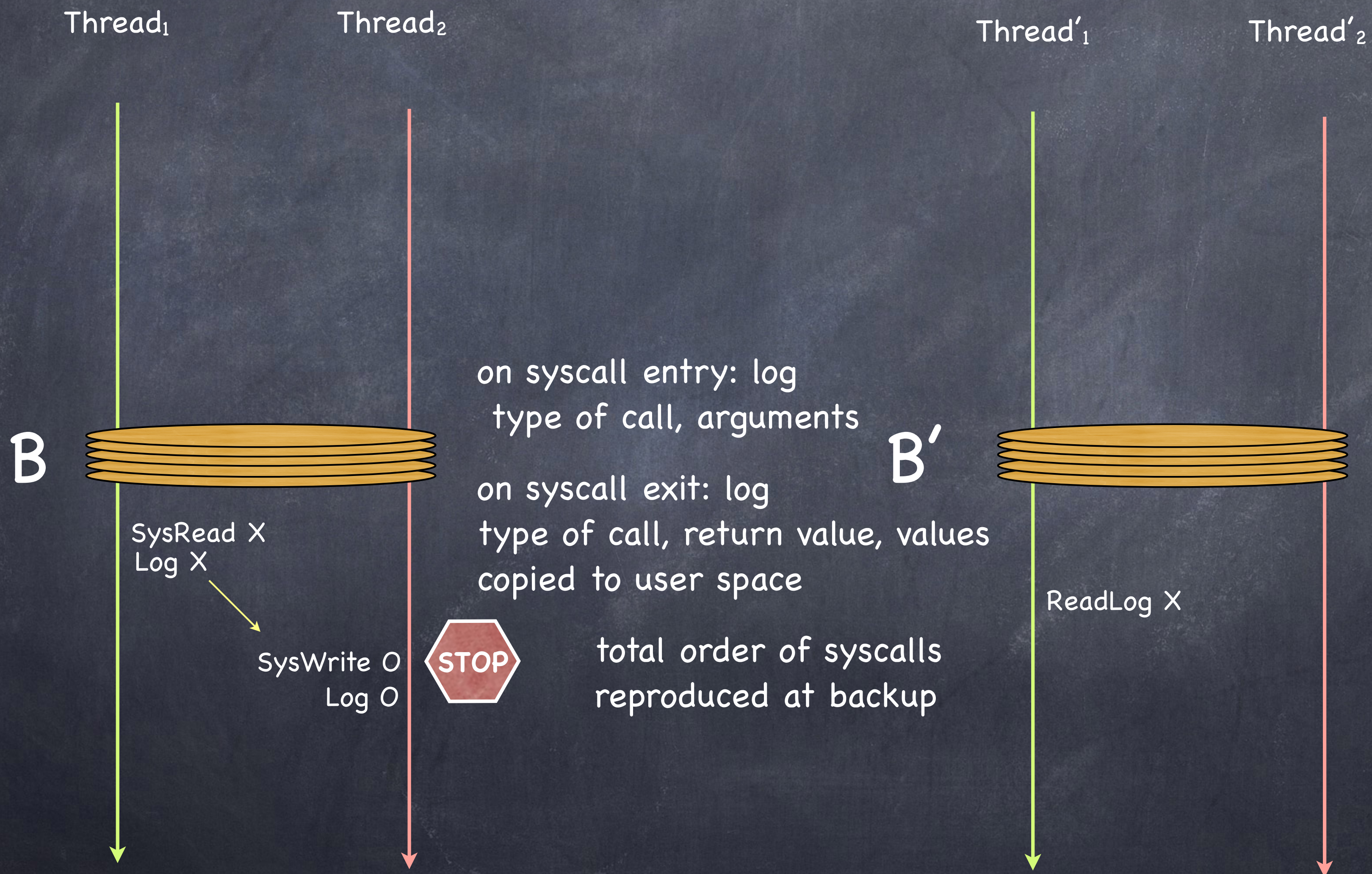
Speculate



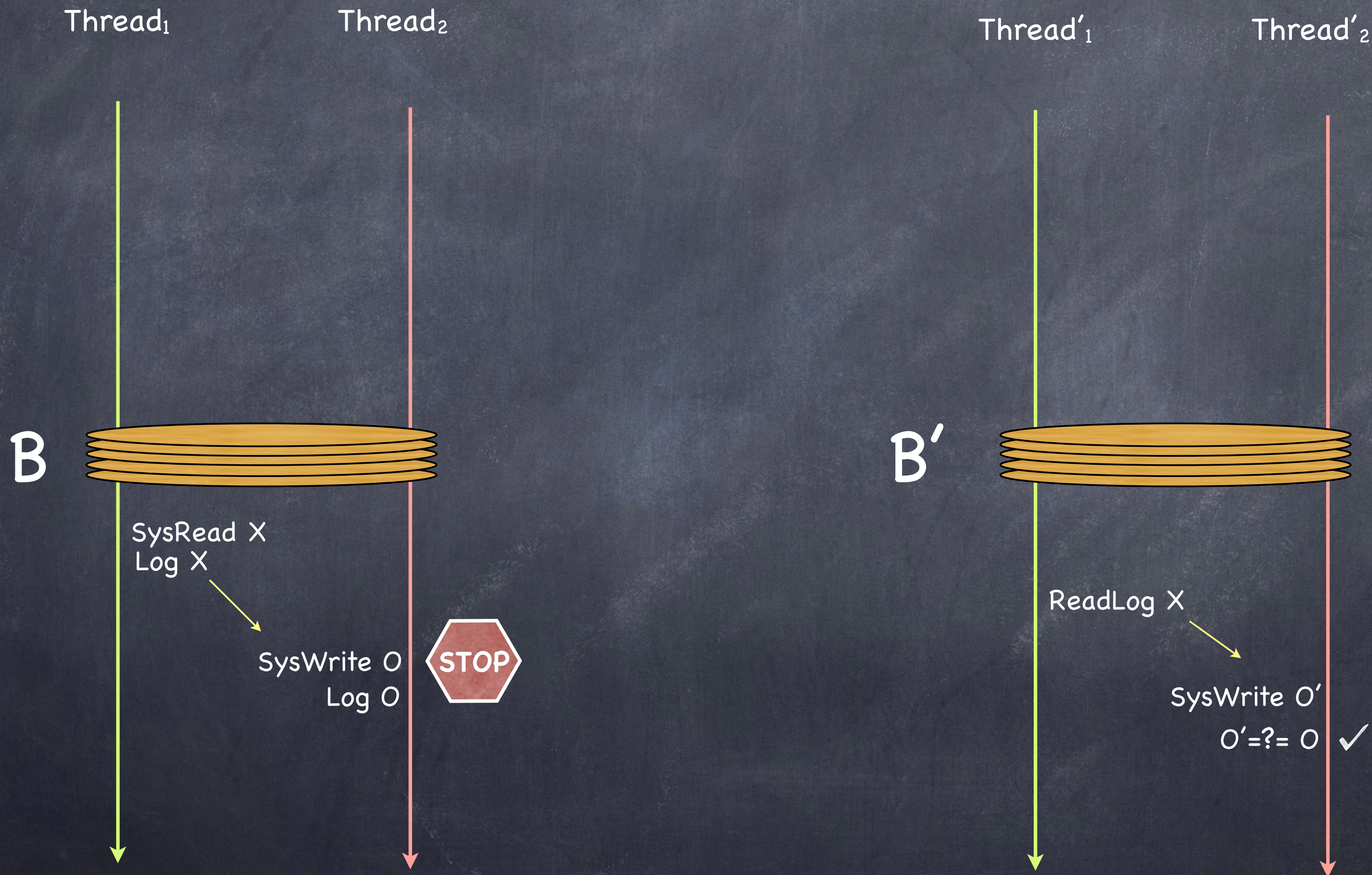
Speculate



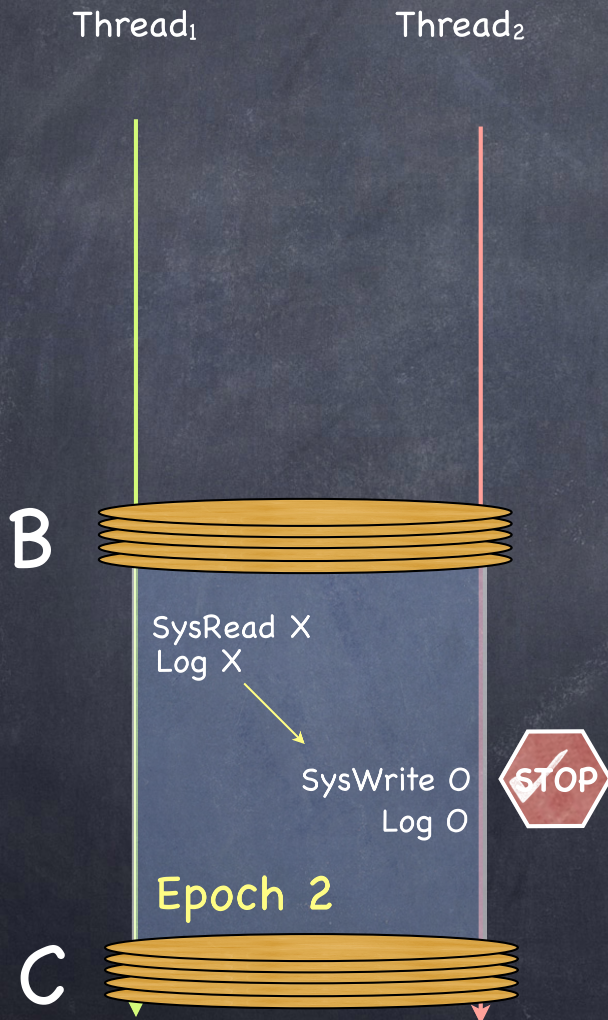
Speculate

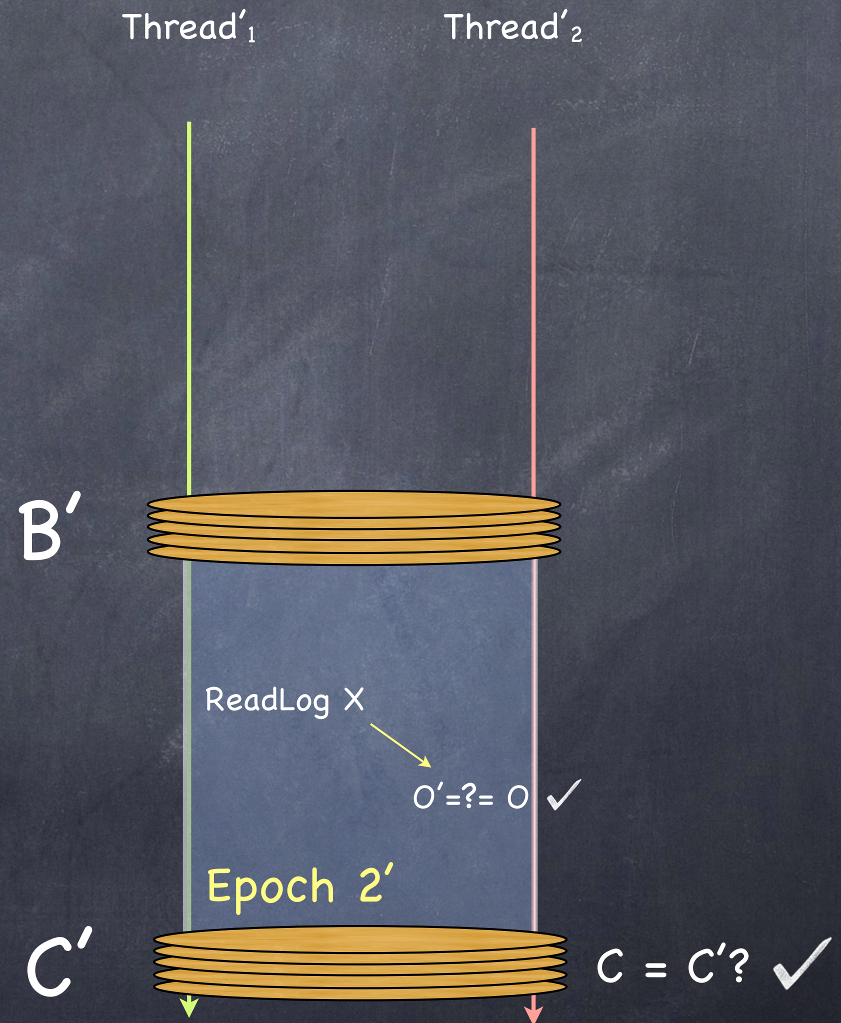


Check

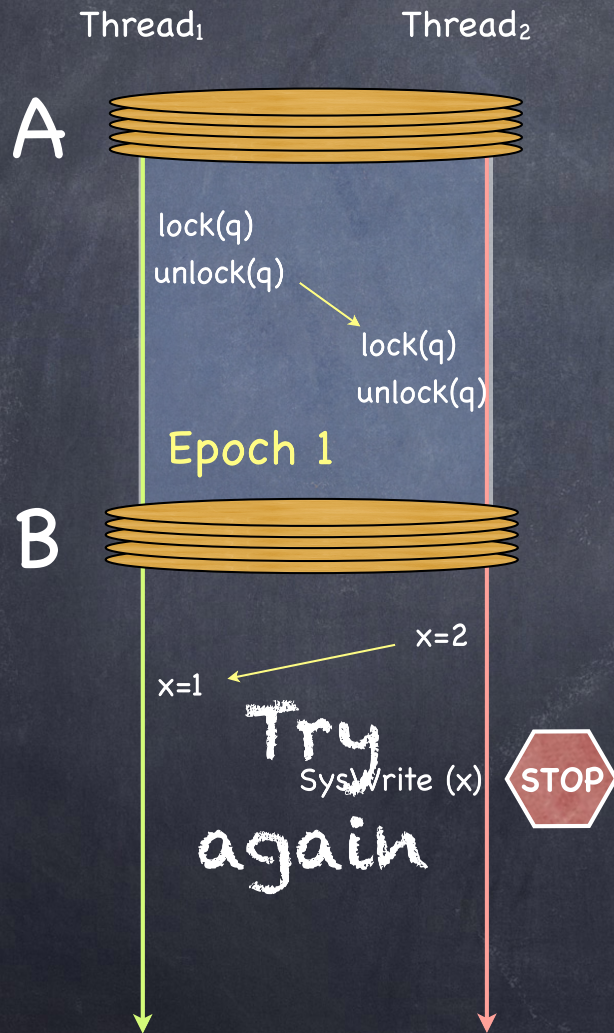


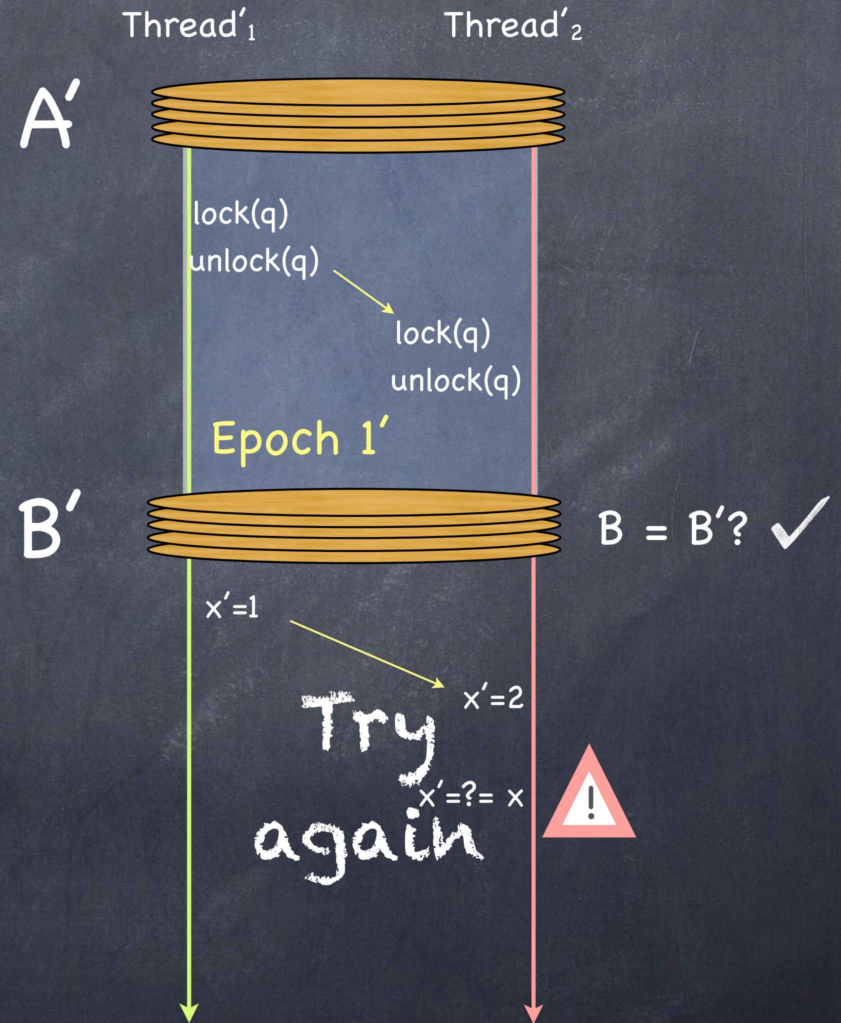
Check





Mis-speculation





Liveness

- Could record individual accesses...

Instead

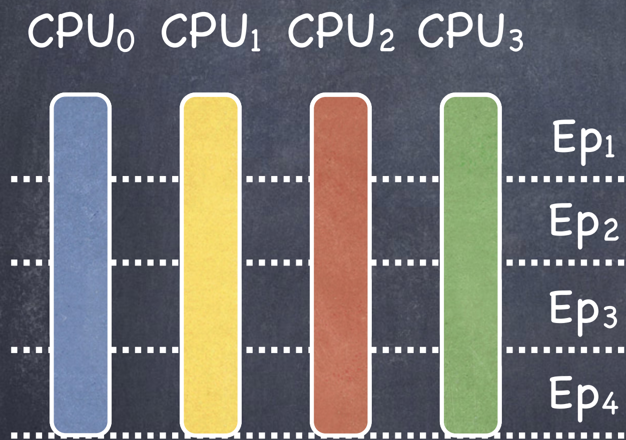
- Switch to uniprocessor execution
  - Record and replay one thread at a time
  - Take checkpoint when thread blocks
  - Resume parallel execution
  - Repeat, if necessary
- In theory, no better than uniprocessor
  - In practice, however...

Offline Multiprocessor Replay

- Respec could log replay info to stable storage, but...
  - expensive
  - with data races, may need several rollback/retry

Two Types of Parallelism

Thread-parallel

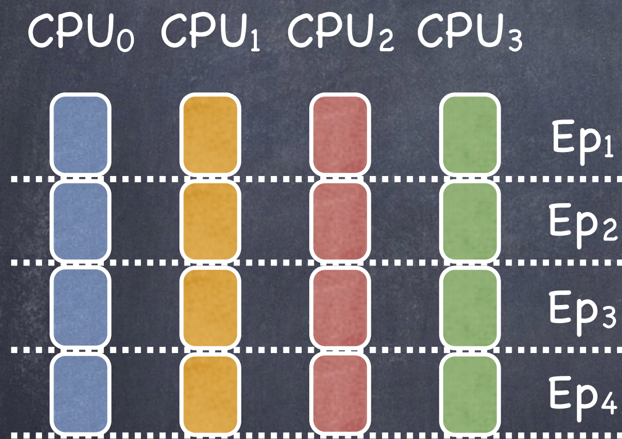


Epoch-parallel

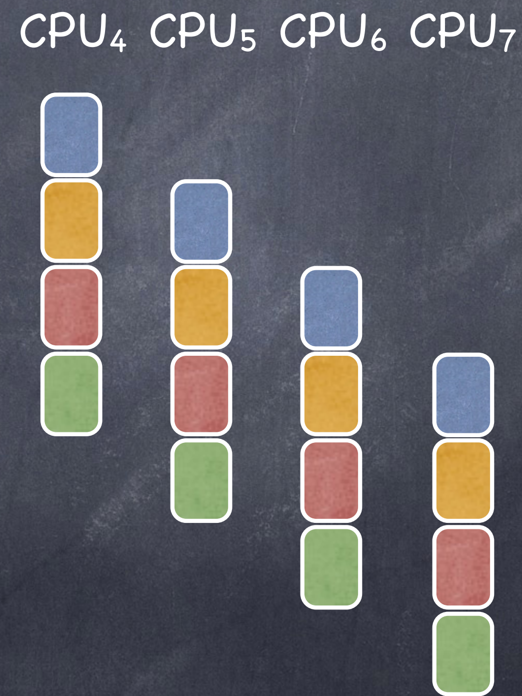
CPU4 CPU5 CPU6 CPU7

Two Types of Parallelism

Thread-parallel



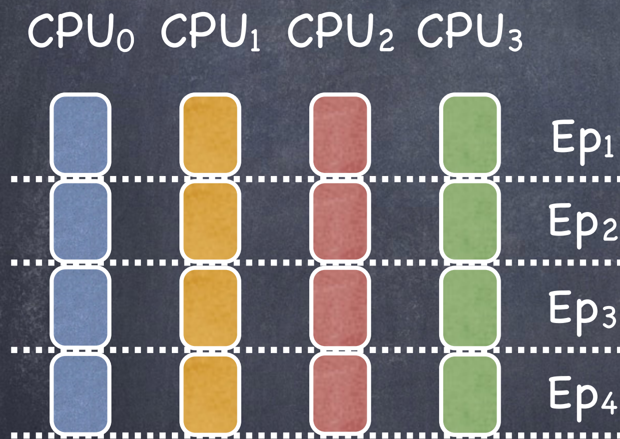
Epoch-parallel



Two Types of Parallelism

Thread-parallel

CPU$_0$ CPU$_1$ CPU$_2$ CPU$_3$



Epoch-parallel

CPU$_4$ CPU$_5$ CPU$_6$ CPU$_7$

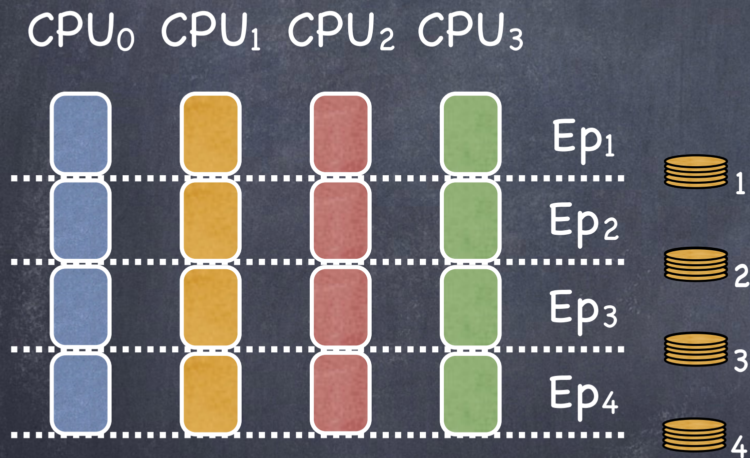
But how do we know whence to start the epoch?

Logging becomes easy!

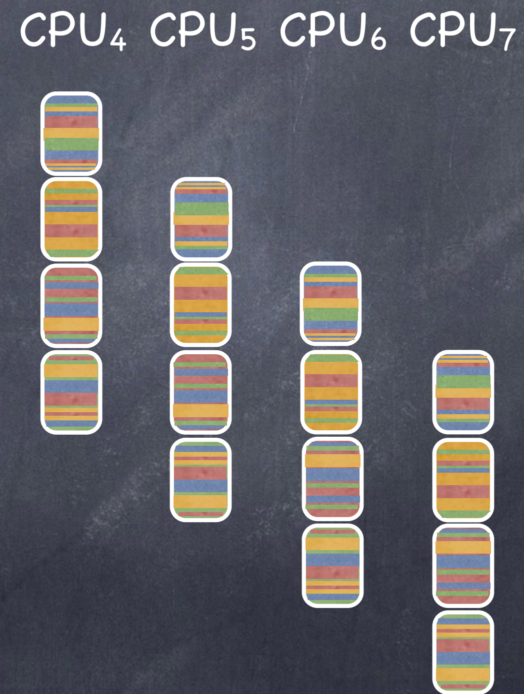
record context switches and non-deterministic system calls!

UniParallelism

Thread-parallel



Epoch-parallel



Managing divergence

- Let thread-parallel guide epoch-parallel
  - Use total order of system calls and partial order of low-level synchronization operations logged during TP execution to guide EP execution
  - “translate” this nformation (which a thread-parallel execution would send using Respec) into scheduling decisions, that are then recorded during the epoch parallel execution

On Divergence

- Roll back to checkpoint at the beginning of failed epoch
  - sometimes multiple rollbacks... or even no progress!
- But the execution being logged is the EP, not the TP!
  - merge kernel state of TP with memory and register state of EP
    - it's ok, since (logical) kernel states are identical, having executed the same system calls