Deterministic Replay for Multiprocessors

Why?

Record and reproduce multithreaded executions

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

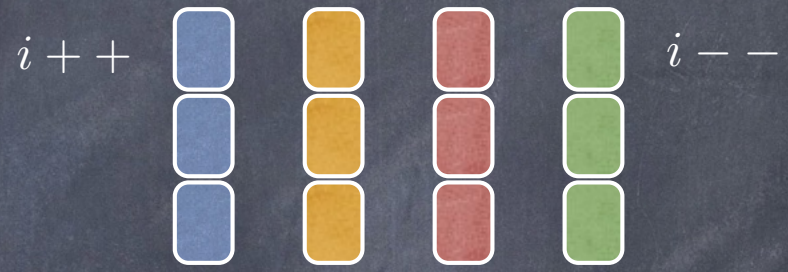
What is hard?

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

Detect dependencies using memory protection bits
Up to 9x

?

Reduced logging + offline search
Slow replay

Hardware support
Custom HW

What to replay?

- Exact reproducibility is hard and expensive...
- ... but no need to replay the exact execution
  - Aim for observational indistinguishability
    - 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

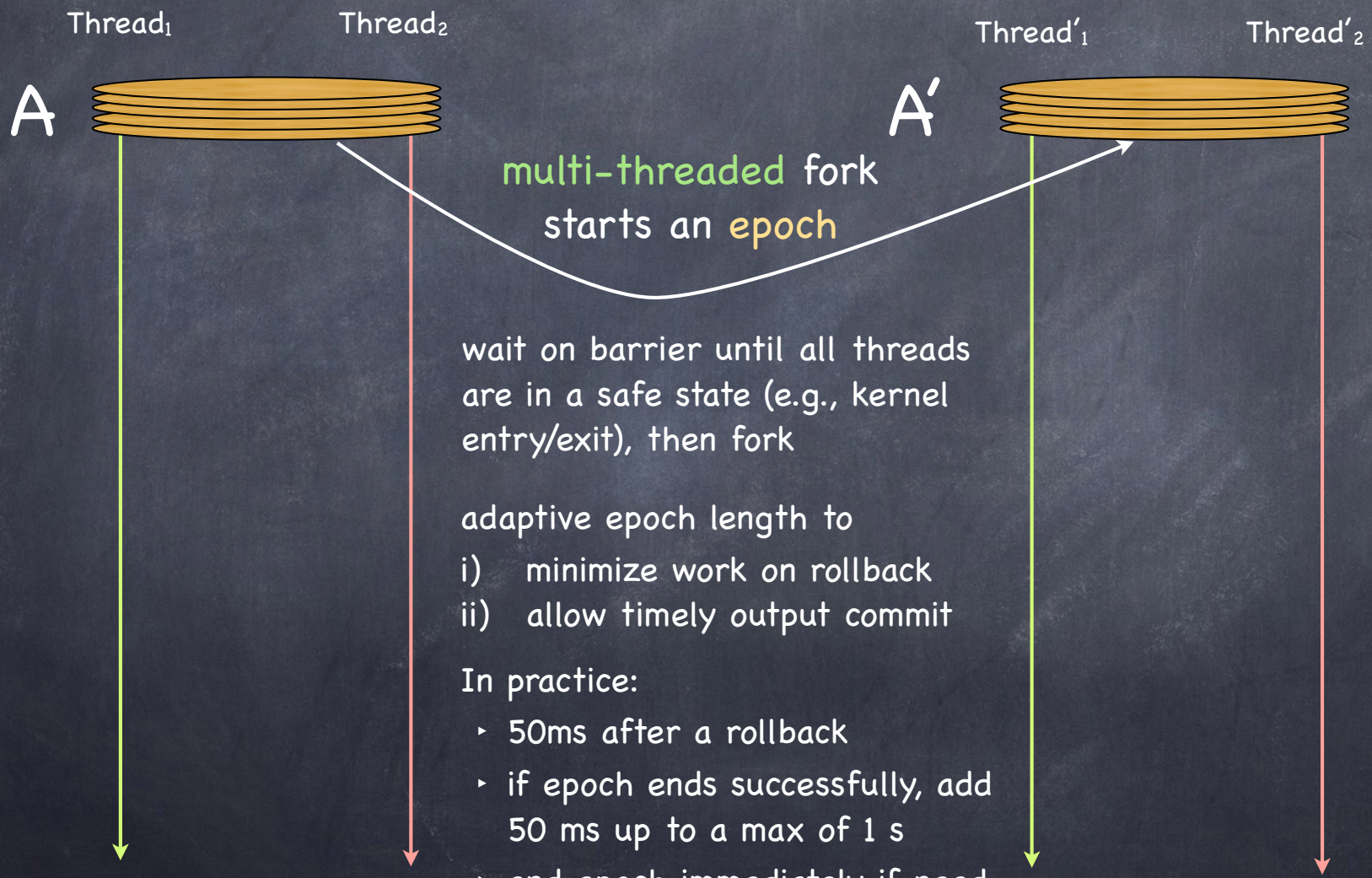
Online Multiprocessor Replay

Respec, ASPLOS '10

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

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

Speculate



wait on barrier until all threads are in a safe state (e.g., kernel entry/exit), then fork

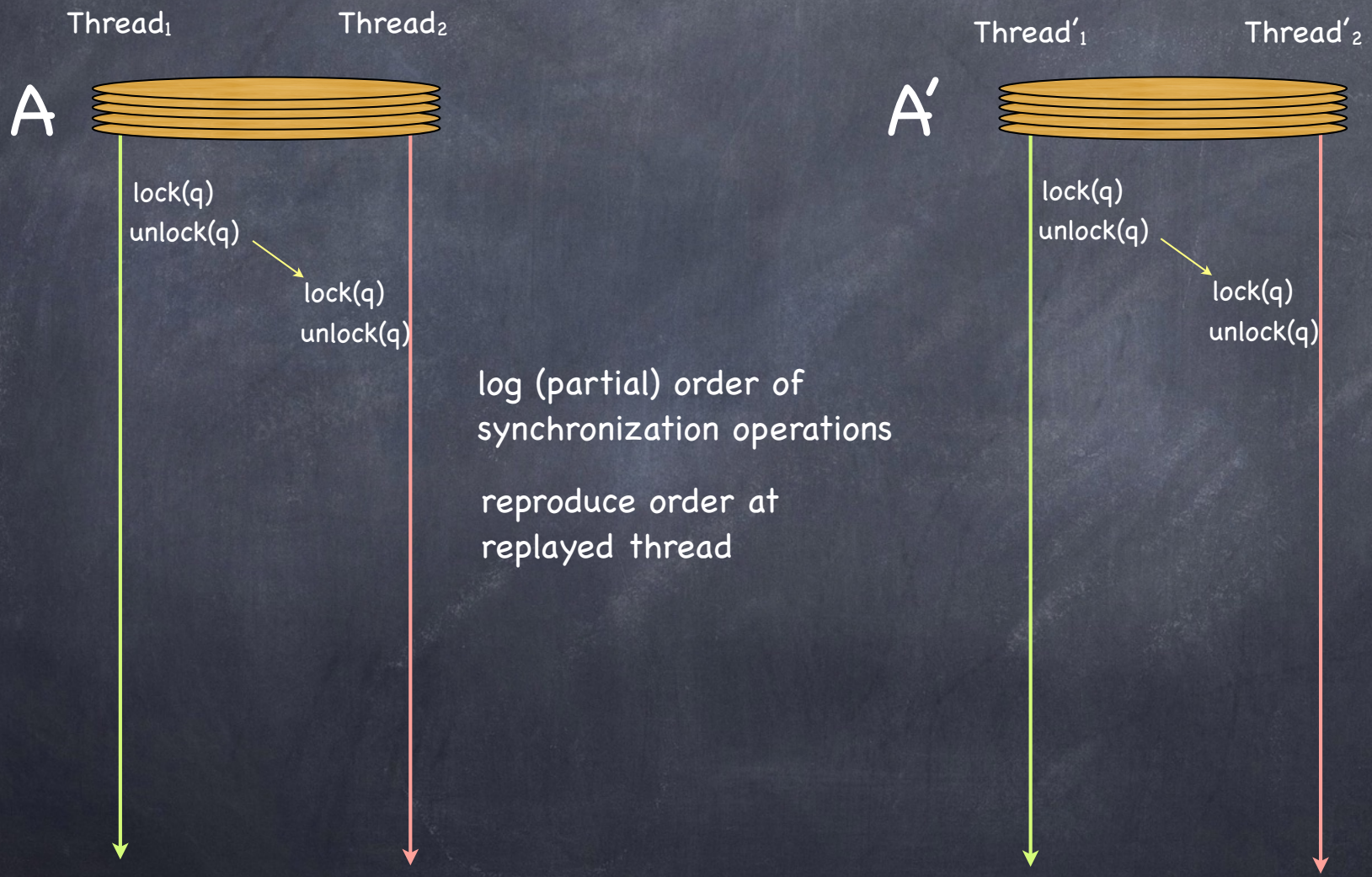
adaptive epoch length to

i) minimize work on rollback
ii) allow timely output commit

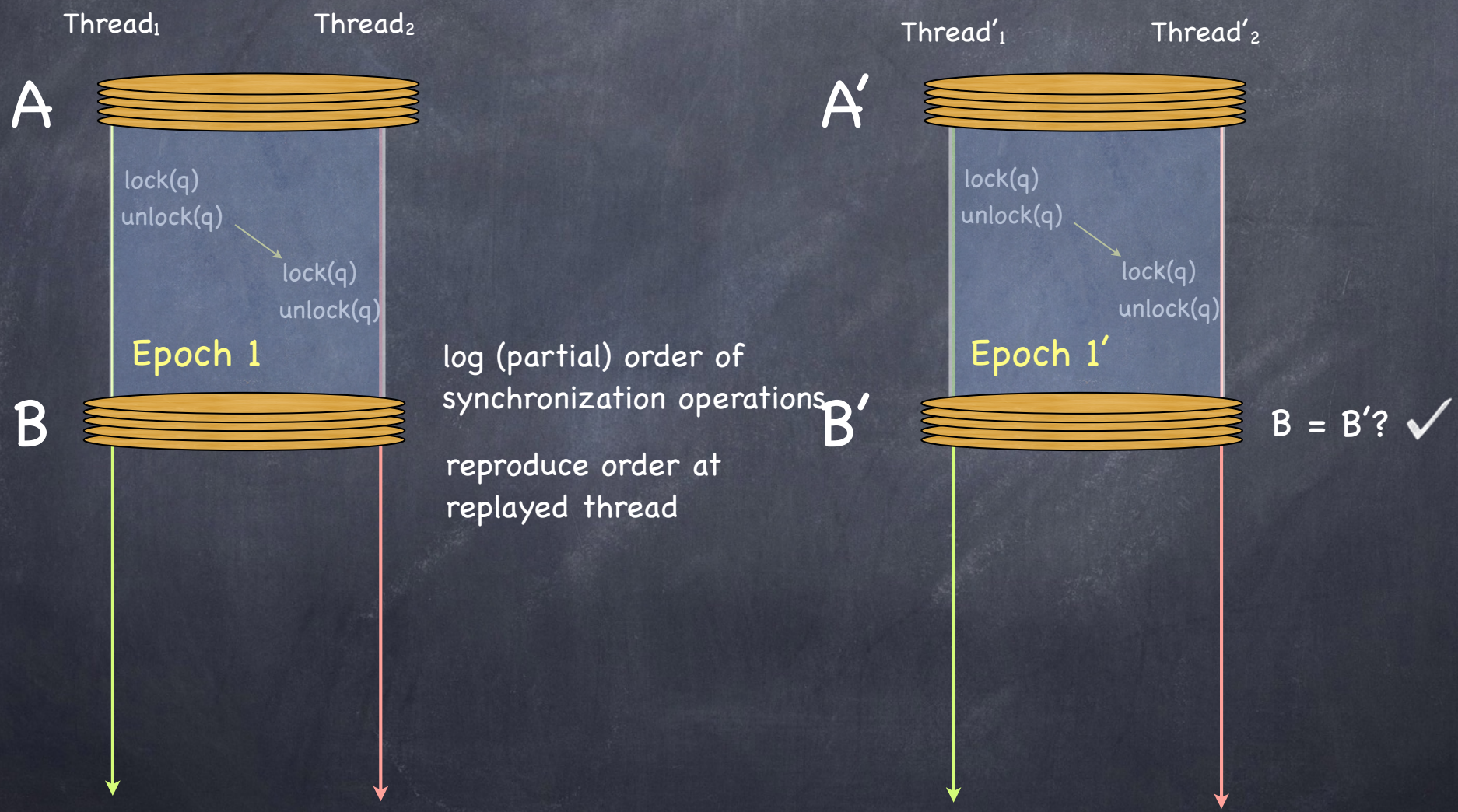
In practice:

- 50ms after a rollback
- if epoch ends successfully, add 50 ms up to a max of 1 s
- end epoch immediately if need synchronous external output

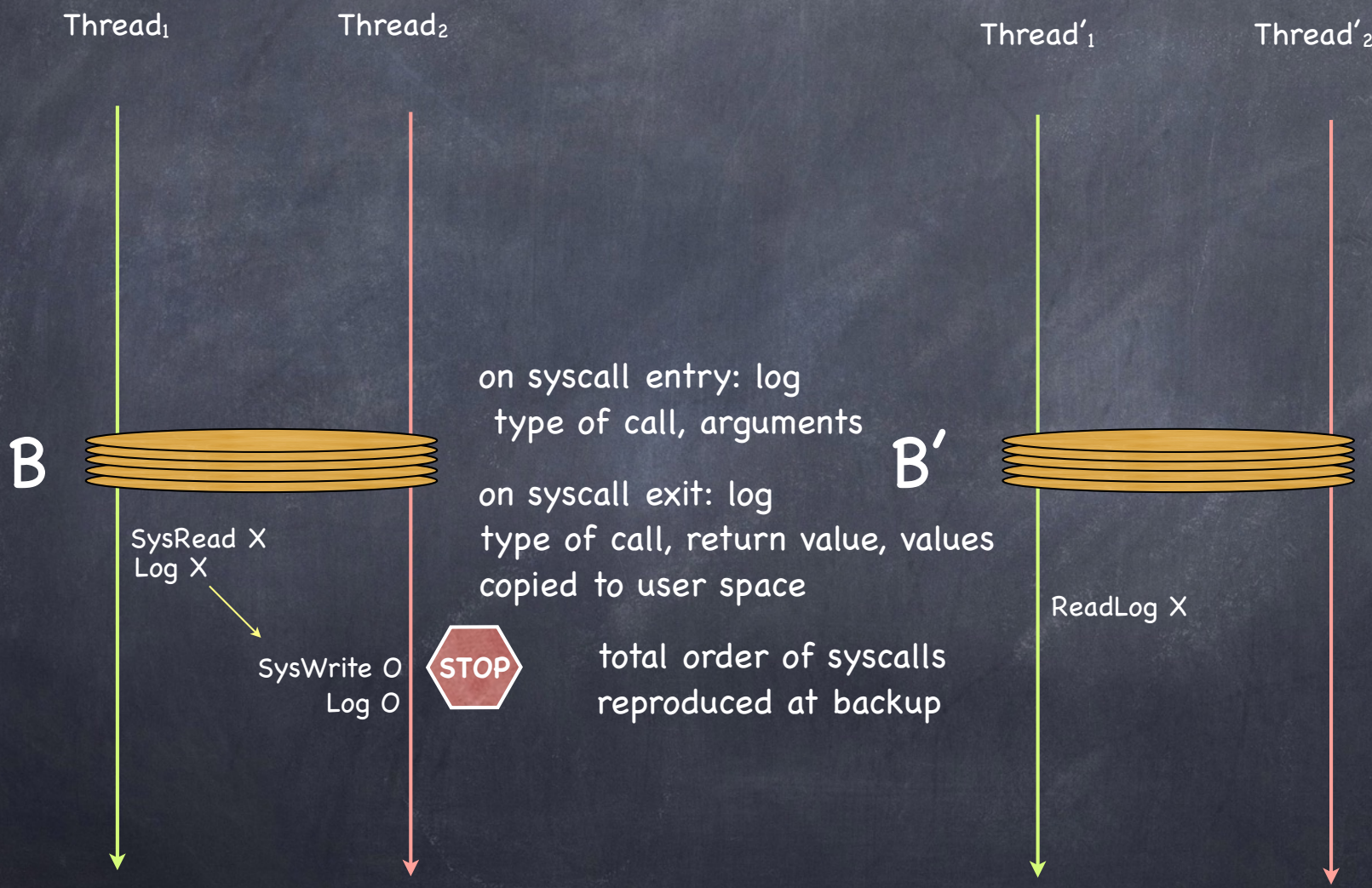
Speculate


Thread1
Thread2
A
lock(q)
unlock(q)
lock(q)
unlock(q)
log (partial) order of synchronization operations
reproduce order at replayed thread
Thread'1
Thread'2
A'
lock(q)
unlock(q)
lock(q)
unlock(q)

Speculate


Thread1
Thread2
A
lock(q)
unlock(q)
lock(q)
unlock(q)
Epoch 1
B
log (partial) order of synchronization operations
reproduce order at replayed thread
Thread'1
Thread'2
A'
lock(q)
unlock(q)
lock(q)
unlock(q)
Epoch 1'
B'
B = B'? ✓

Speculate

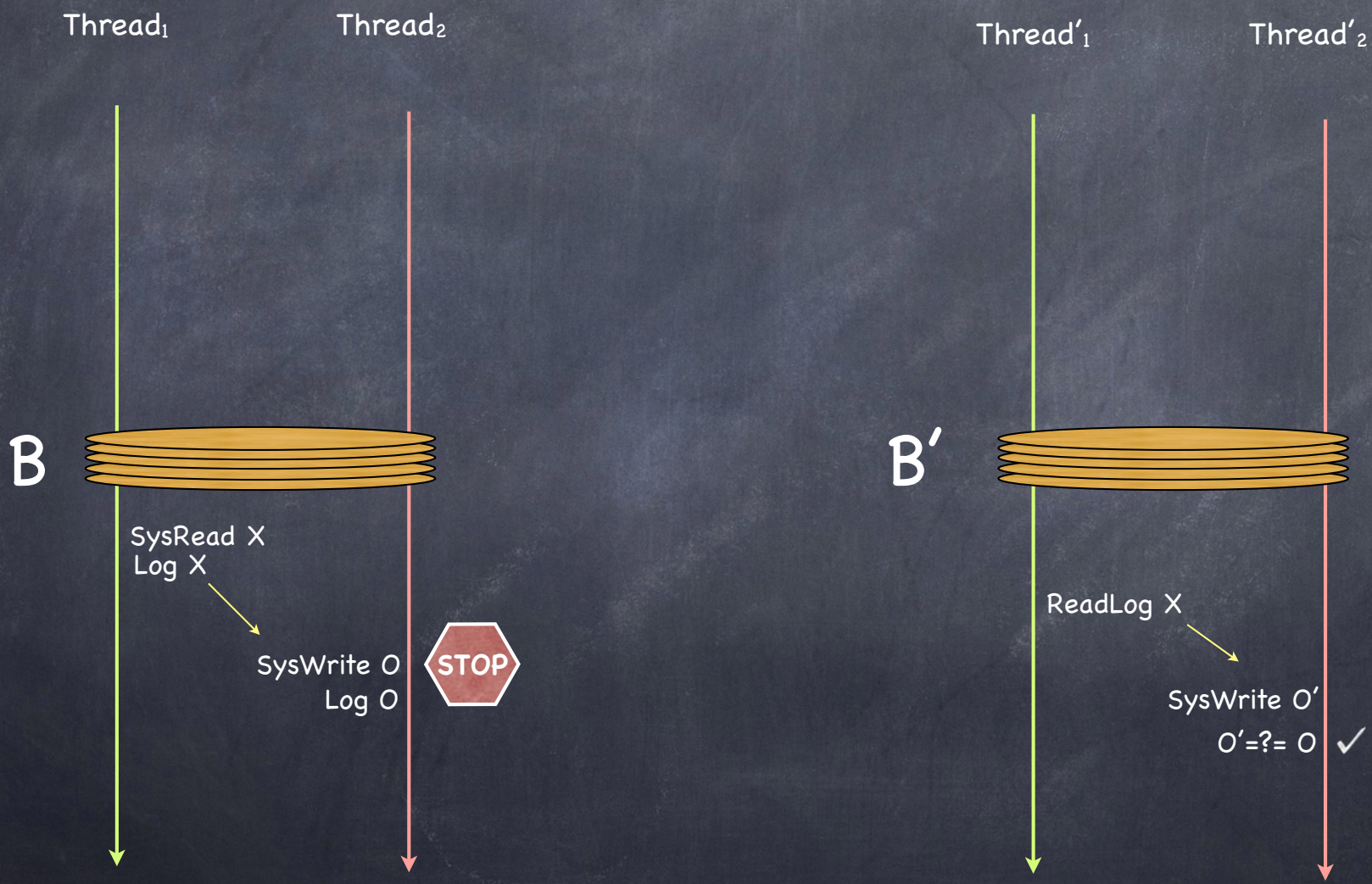


on syscall entry: log
type of call, arguments

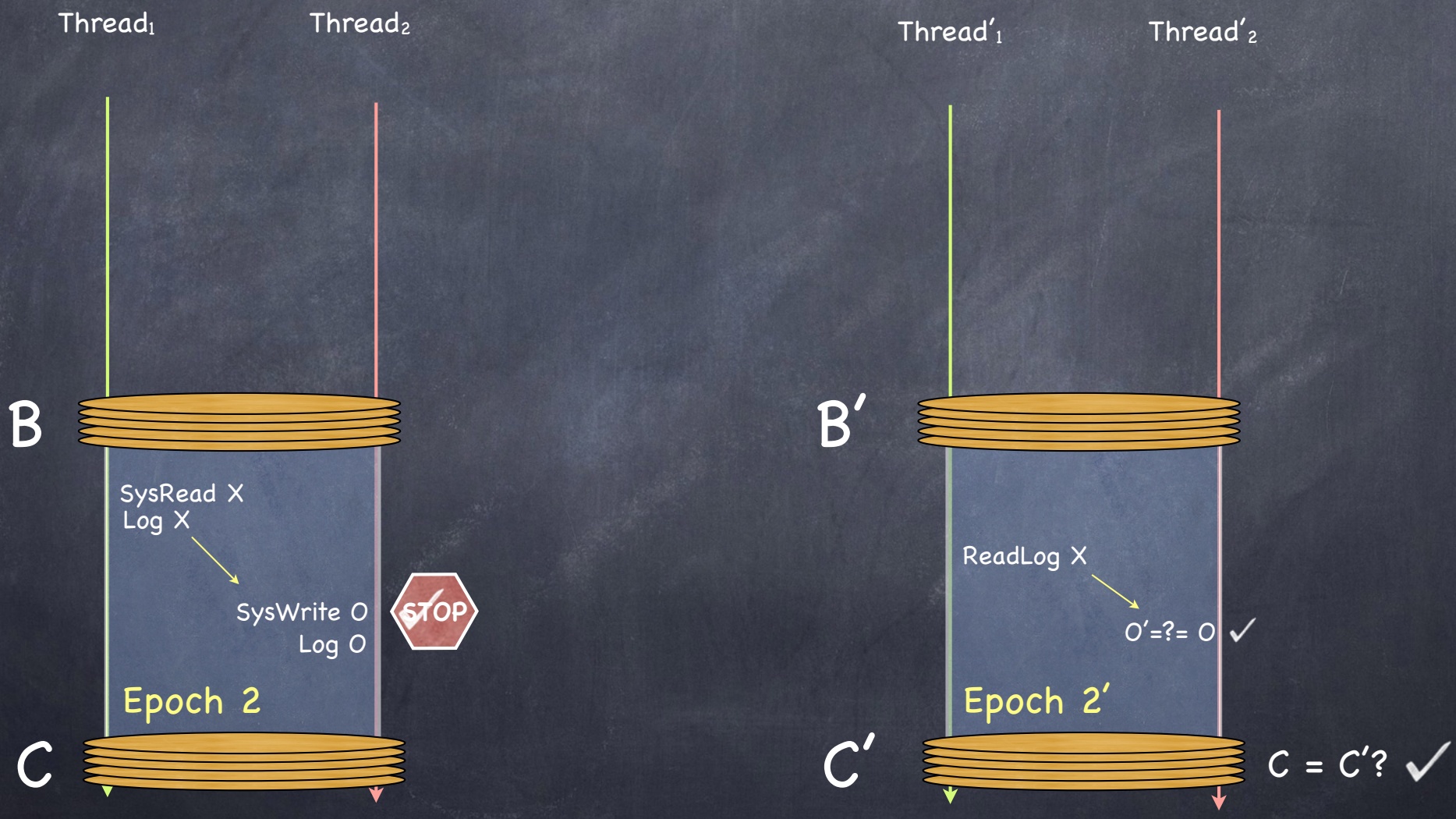
on syscall exit: log
type of call, return value, values copied to user space

total order of syscalls reproduced at backup

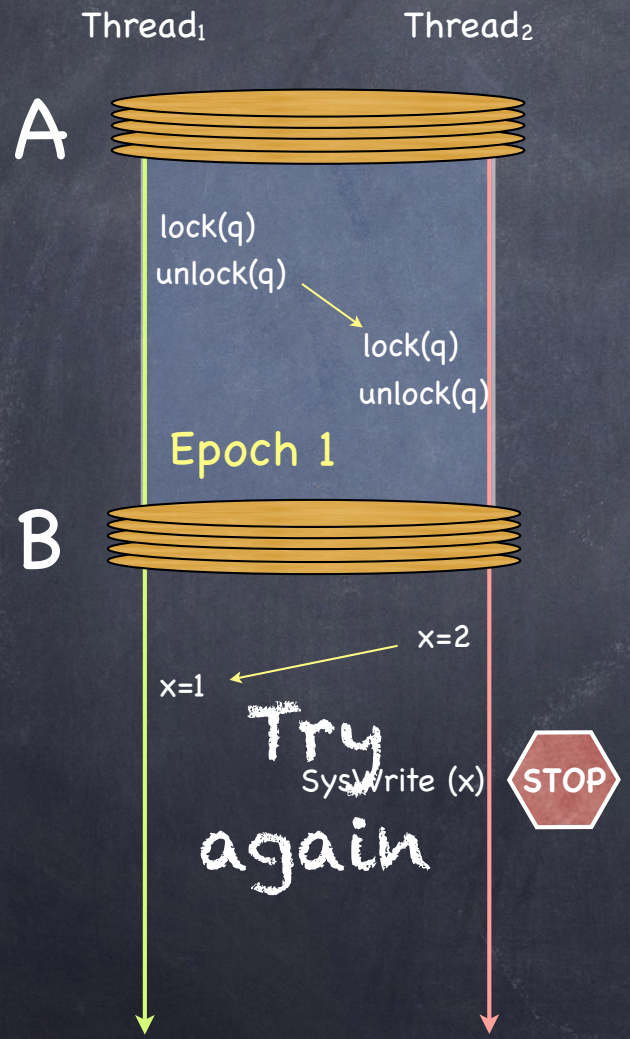
Check

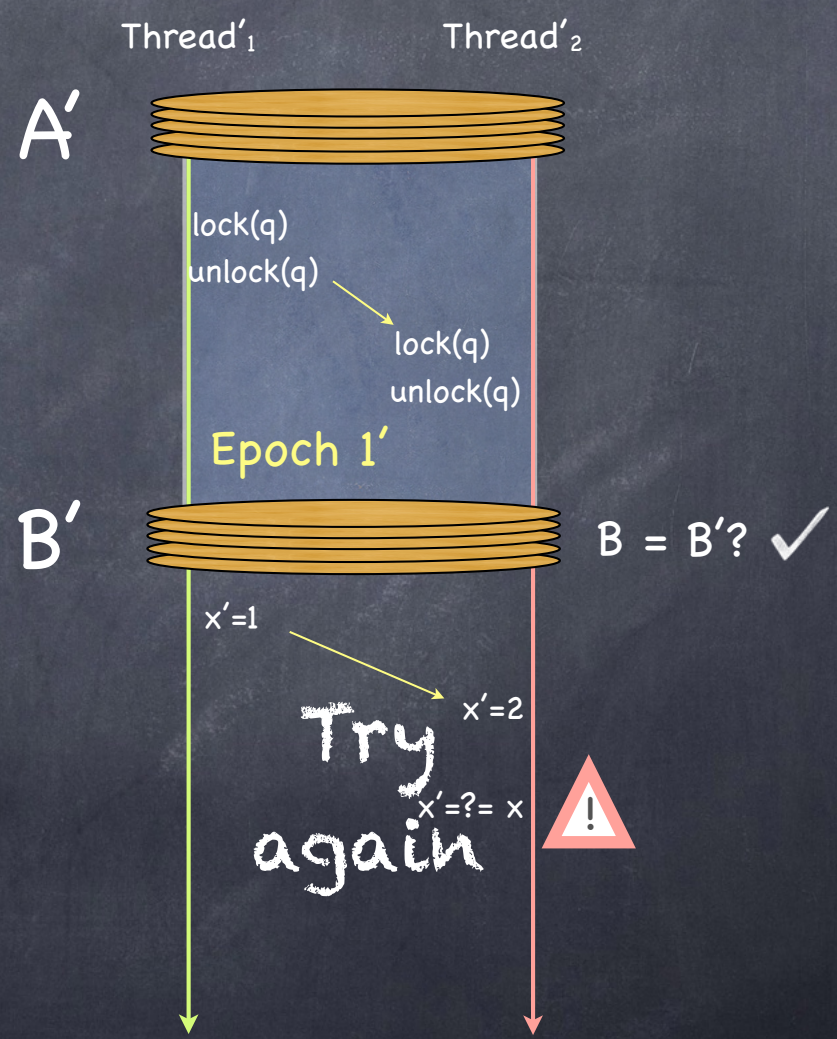

Thread1
Thread2
B
SysRead X
Log X
SysWrite O
Log O
STOP
Thread'1
Thread'2
B'
ReadLog X
SysWrite O'
O'=?= O

Check


Thread1
Thread2
B
SysRead X
Log X
SysWrite O
Log O
STOP
Epoch 2
C
Thread'1
Thread'2
B'
ReadLog X
O'=?= O
Epoch 2'
C'
C = C'?

Mis-speculation





Liveness

- Could record individual accesses...

Instead

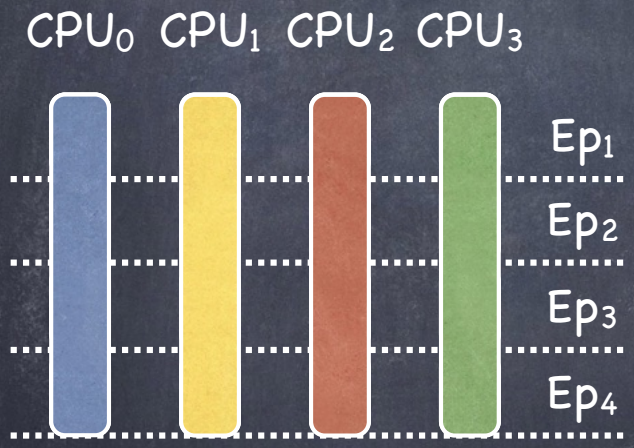
- Switch to uniprocessor execution
  - Record and replay one thread at a time
  - Take checkpoint when thread blocks
  - Resume parallel execution
  - Repeat, if necessary
- In theory, no better than uniprocessor
  - In practice, however...

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

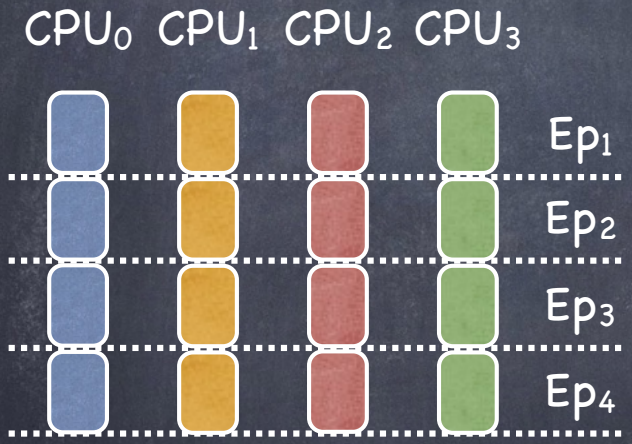


Epoch-parallel

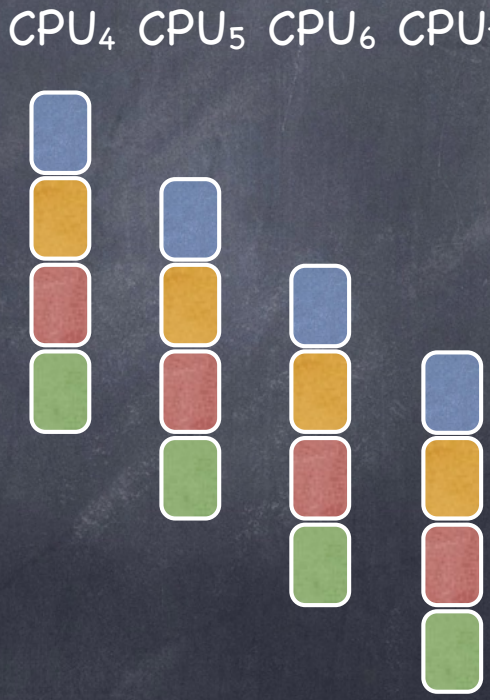
CPU4 CPU5 CPU6 CPU7

Two Types of Parallelism

Thread-parallel

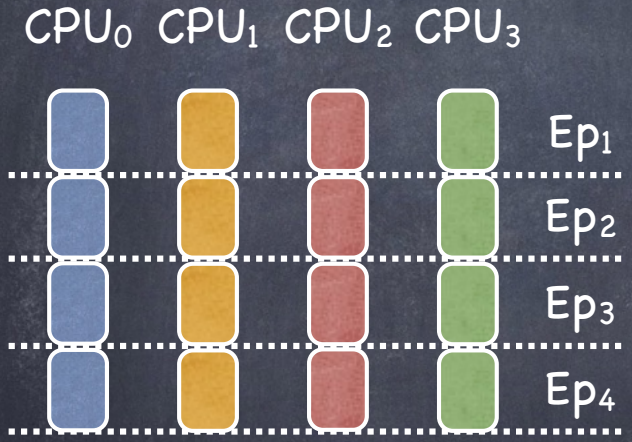


Epoch-parallel

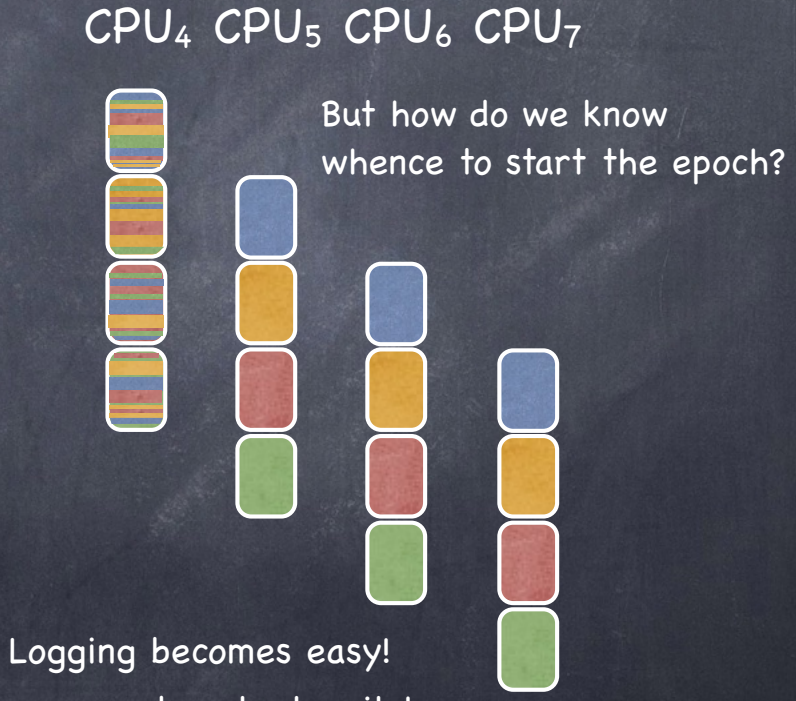


Two Types of Parallelism

Thread-parallel



Epoch-parallel

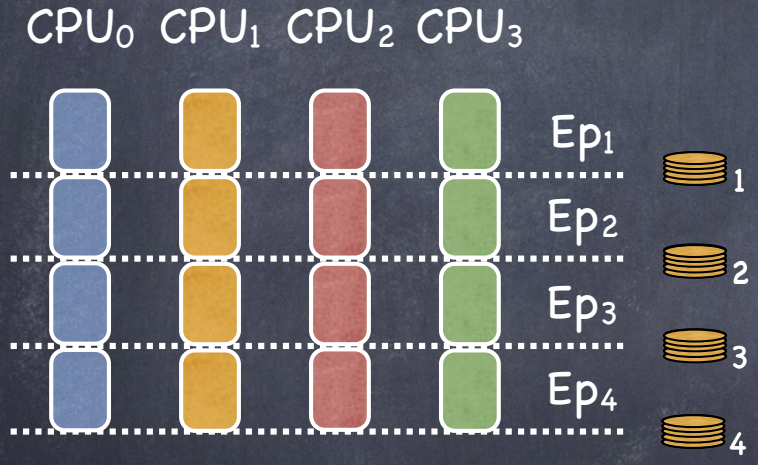


Logging becomes easy!

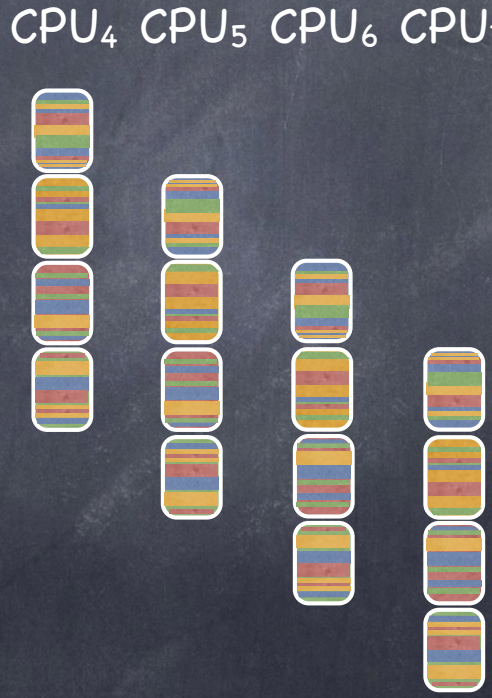
record context switches
and non-deterministic system calls!

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

Atomic Commit

The objective

Preserve data consistency for distributed transactions in the presence of failures

Model

- For each distributed transaction T:
  - one coordinator
  - a set of participants
- Coordinator knows participants; participants don't necessarily know each other
- Each process has access to a Distributed Transaction Log (DT Log) on stable storage

The setup

- Each process p_i has an input value $vote_i$:

$$vote_i \in \{\text{Yes, No}\}$$

- Each process p_i has output value $decision_i$:

$$decision_i \in \{\text{Commit, Abort}\}$$

AC Specification

AC-1: All processes that reach a decision reach the same one.

AC-2: A process cannot reverse its decision after it has reached one.

AC-3: The Commit decision can only be reached if all processes vote Yes.

AC-4: If there are no failures and all processes vote Yes then the decision will be Commit.

AC-5: If all failures are repaired and there are no more failures then all processes will eventually decide.

Comments

AC-1: All processes that reach a decision reach the same one.

AC-2: A process cannot reverse its decision after it has reached one

AC-3: The Commit decision can only be reached if all processes vote Yes

AC-4: If there are no failures and all processes vote Yes, then the decision will be Commit

AC-5: If all failures are reported and there are no more failures, then all processes will eventually decide

AC1:

- We do not require all processes to reach a decision
- We do not even require all correct processes to reach a decision! (impossible to accomplish if links fail)

AC4:

- Avoids triviality
- Allows Abort even if all processes have voted yes

NOTE:

- A process that does not vote Yes can unilaterally abort

Liveness & Uncertainty

- A process is uncertain if it has voted Yes but does not have sufficient information to commit
- While uncertain, a process cannot decide unilaterally
- Uncertainty + communication failures = blocking!

Liveness & Independent Recovery

- Suppose process p fails while running AC.
- If, during recovery, p can reach a decision without communicating with other processes, we say that p can independently recover
- Total failure (i.e. all processes fail) and no independent recovery = blocking

A few character-building facts

Proposition 1

If communication failures or total failures are possible, then every AC protocol may cause processes to become blocked

Proposition 2

No AC protocol can guarantee independent recovery of failed processes

2-Phase Commit

Coordinator c

Participant p_i

I. sends VOTE-REQ to all participants

2-Phase Commit

Coordinator c

Participant p_i

I. sends VOTE-REQ to all participants

II. sends $vote_i$ to Coordinator

if $vote_i$ = NO then

$decide_i$:= ABORT

halt

2-Phase Commit

Coordinator c

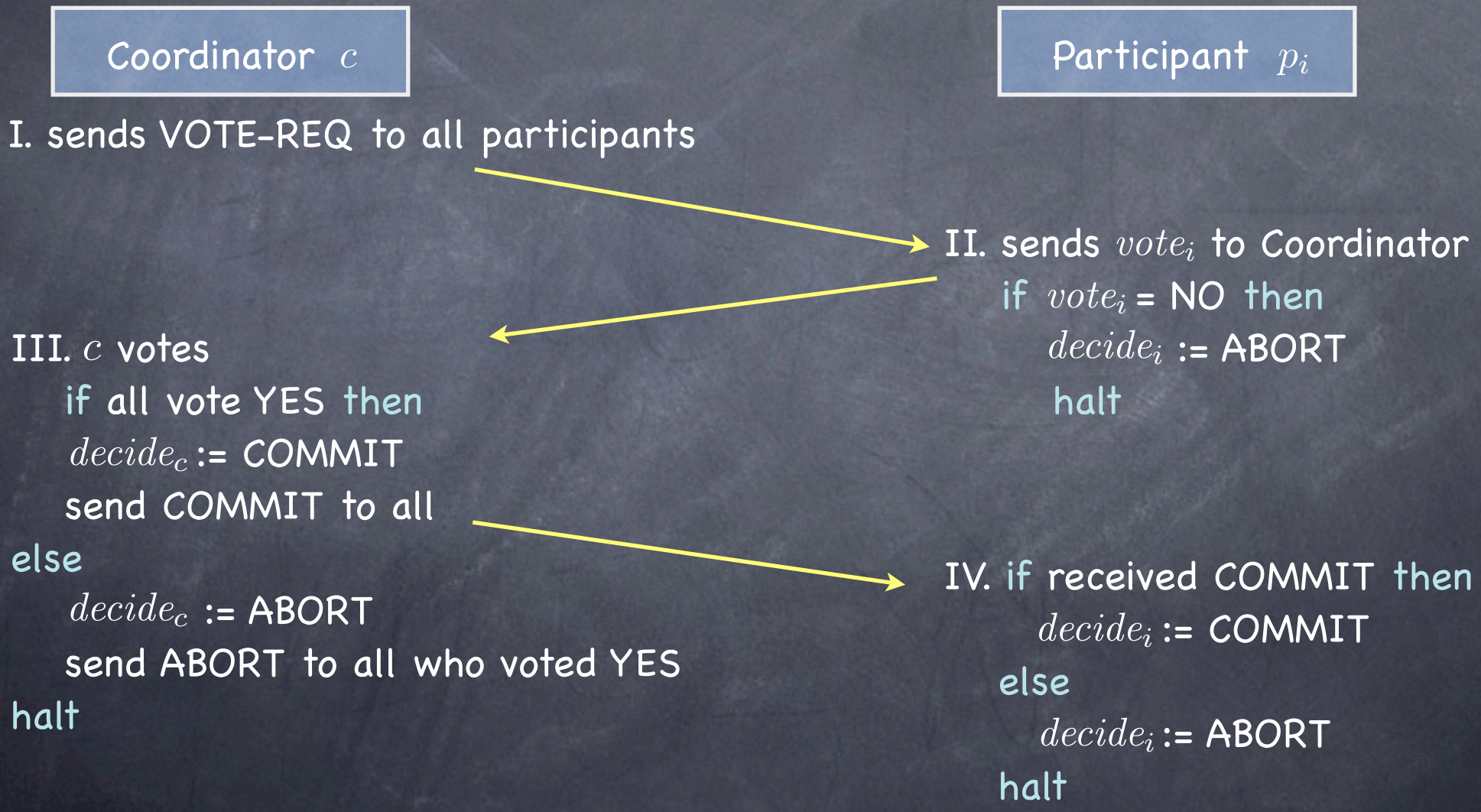
Participant p_i

I. sends VOTE-REQ to all participants

II. sends $vote_i$ to Coordinator
if $vote_i$ = NO then
$decide_i$:= ABORT
halt

III. c votes
if all vote YES then
$decide_c$:= COMMIT
send COMMIT to all
else
$decide_c$:= ABORT
send ABORT to all who voted YES
halt

2-Phase Commit



Notes on 2PC

- Satisfies AC-1 to AC-4
- But not AC-5 (at least "as is")
  i. A process may be waiting for a message that may never arrive
     - Use Timeout Actions
  ii. No guarantee that a recovered process will reach a decision consistent with that of other processes
     - Processes save protocol state in DT-Log

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Step 3 Coordinator is waiting for vote from participants

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Since it is has not cast its vote yet, p_i can decide ABORT and halt.

Step 3 Coordinator is waiting for vote from participants

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Since it is has not cast its vote yet, p_i can decide ABORT and halt.

Step 3 Coordinator is waiting for vote from participants

Coordinator can decide ABORT, send ABORT to all participants which voted YES, and halt.

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

Timeout actions

Processes are waiting on steps 2, 3, and 4

Step 2 p_i is waiting for VOTE-REQ from coordinator

Since it is has not cast its vote yet, p_i can decide ABORT and halt.

Step 3 Coordinator is waiting for vote from participants

Coordinator can decide ABORT, send ABORT to all participants which voted YES, and halt.

Step 4 p_i (who voted YES) is waiting for COMMIT or ABORT

p_i cannot decide: it must run a termination protocol

Termination protocols

I. Wait for coordinator to recover
   - It always works, since the coordinator is never uncertain
   - may block recovering process unnecessarily

II. Ask other participants

Cooperative Termination

- c appends list of participants to VOTE-REQ
- when an uncertain process p times out, it sends a DECISION-REQ message to every other participant q
- if q has decided, then it sends its decision value to p, which decides accordingly
- if q has not yet voted, then it decides ABORT, and sends ABORT to p
- What if every q is uncertain?

Logging actions

1. When c sends VOTE-REQ, it writes START-2PC to its DT Log
2. When p_i is ready to vote YES,
   i. p_i writes YES to DT Log
   ii. p_i sends YES to c (p_i writes also list of participants)
3. When p_i is ready to vote NO, it writes ABORT to DT Log
4. When c is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
5. When c is ready to decide ABORT, it writes ABORT to DT Log
6. After p_i receives decision value, it writes it to DT Log

p recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote Yes, writes Yes to DT Log before sending yes to coordinator (writes also list of participants)
   When participant is ready to vote No, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

p recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote Yes, writes Yes to DT Log before sending yes to coordinator (writes also list of participants)
   When participant is ready to vote No, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

- if DT Log contains START-2PC, then $p = c$:
  - if DT Log contains a decision value, then decide accordingly
  - else decide ABORT

p recovers

1. When coordinator sends VOTE-REQ, it writes START-2PC to its DT Log
2. When participant is ready to vote Yes, writes Yes to DT Log before sending yes to coordinator (writes also list of participants)
   When participant is ready to vote No, it writes ABORT to DT Log
3. When coordinator is ready to decide COMMIT, it writes COMMIT to DT Log before sending COMMIT to participants
   When coordinator is ready to decide ABORT, it writes ABORT to DT Log
4. After participant receives decision value, it writes it to DT Log

- if DT Log contains START-2PC, then $p = c$:
  - if DT Log contains a decision value, then decide accordingly
  - else decide ABORT
- otherwise, p is a participant:
  - if DT Log contains a decision value, then decide accordingly
  - else if it does not contain a Yes vote, decide ABORT
  - else (Yes but no decision) run a termination protocol

2PC and blocking

- Blocking occurs whenever the progress of a process depends on the repairing of failures
- No AC protocol is non blocking in the presence of communication or total failures
- But 2PC can block even with non-total failures and no communication failures among operating processes!

3-Phase Commit

- Two approaches:
  1. Focus only on site failures
     - Non-blocking, unless all sites fails
     - Timeout $\equiv$ site at the other end failed
     - Communication failures can produce inconsistencies
  2. Tolerate both site and communication failures
     - partial failures can still cause blocking, but less often than in 2PC

Blocking and uncertainty

Why does uncertainty lead to blocking?

Blocking and uncertainty

Why does uncertainty lead to blocking?

- An uncertain process does not know whether it can safely decide COMMIT or ABORT because some of the processes it cannot reach could have decided either

Blocking and uncertainty

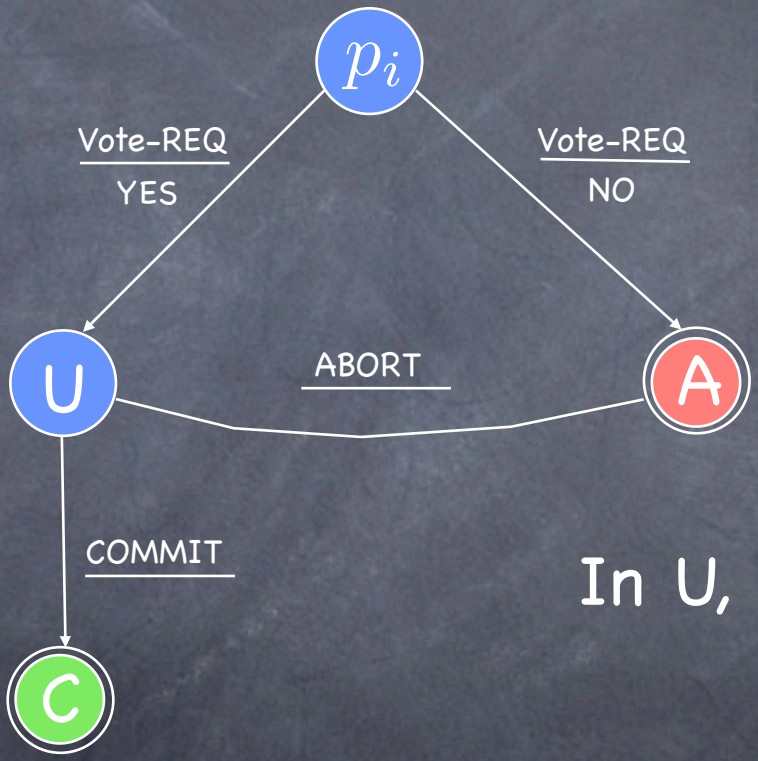
Why does uncertainty lead to blocking?

- An uncertain process does not know whether it can safely decide COMMIT or ABORT because some of the processes it cannot reach could have decided either

Non-blocking Property

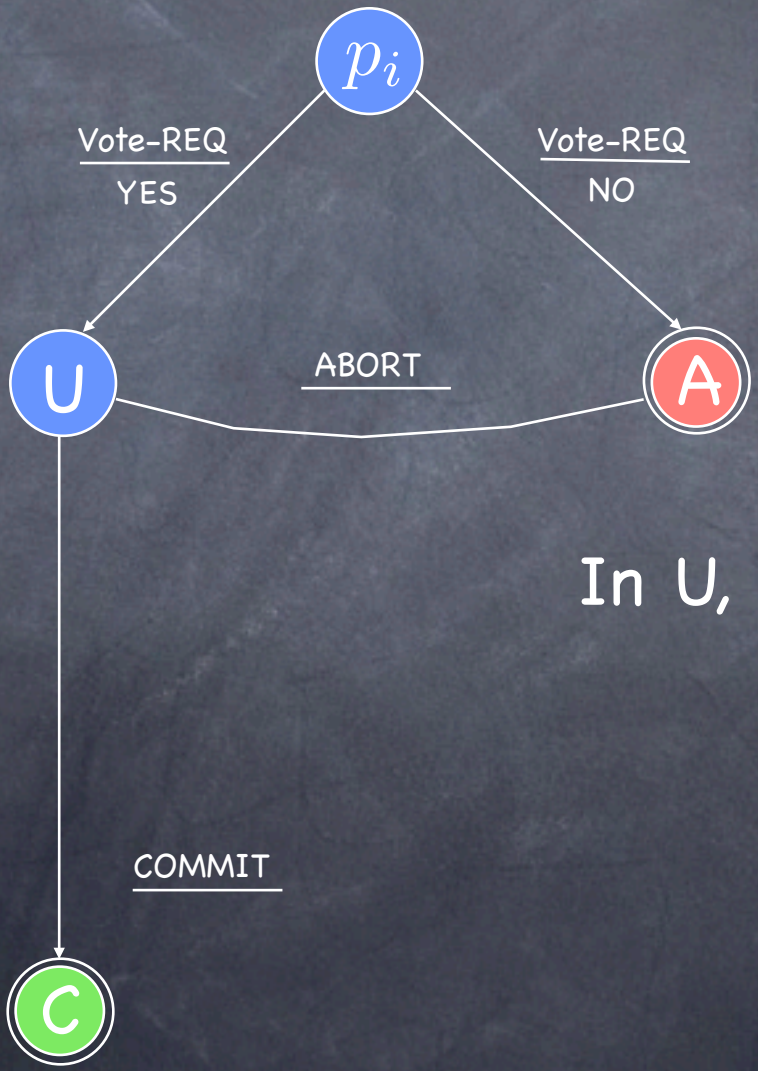
If any operational process is uncertain, then no process has decided COMMIT

2PC Revisited



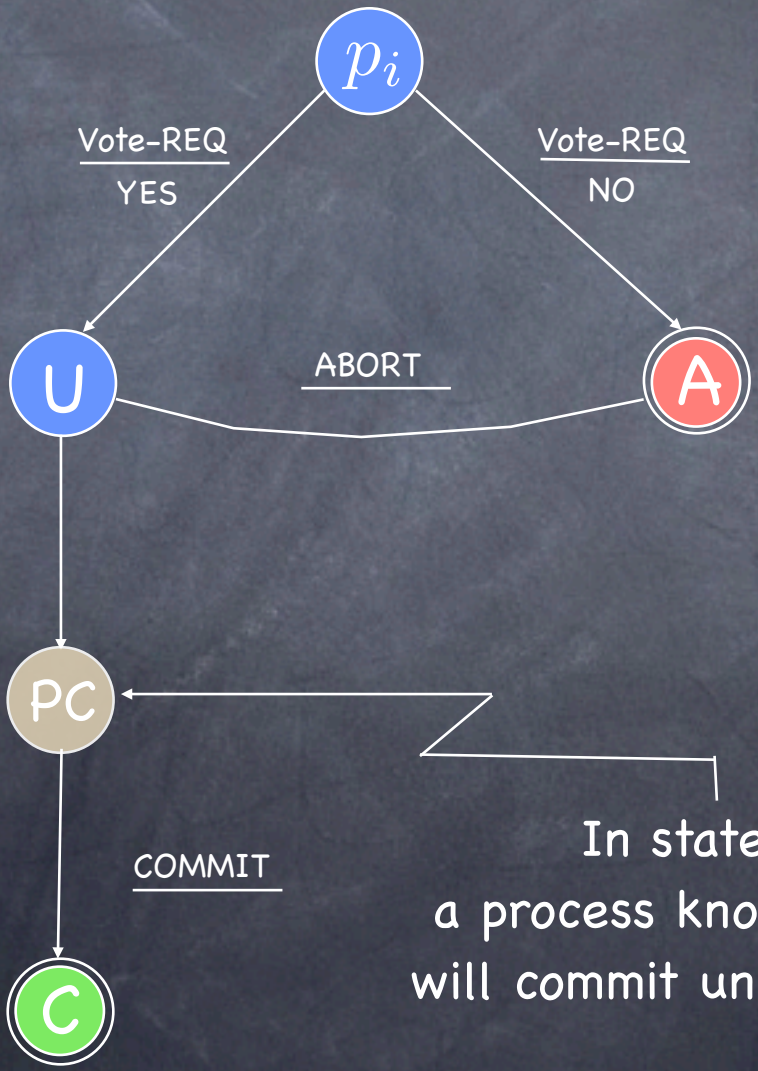
In U, both A and C are reachable!

2PC Revisited



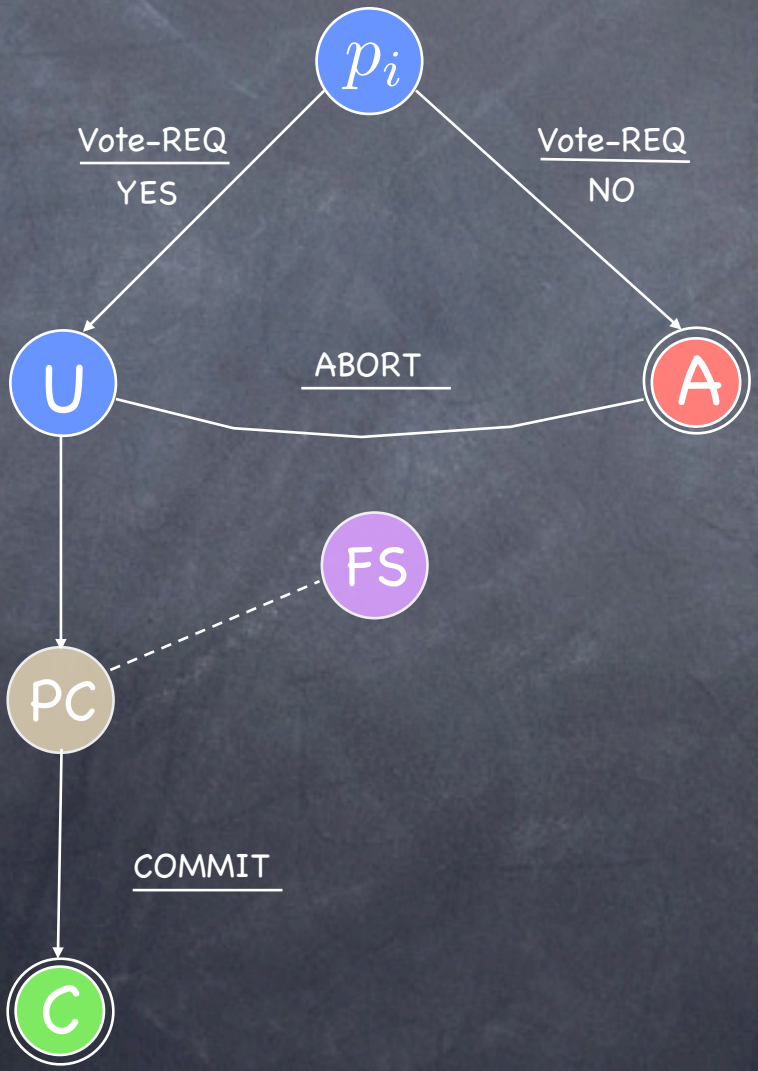
In U, both A and C are reachable!

2PC Revisited

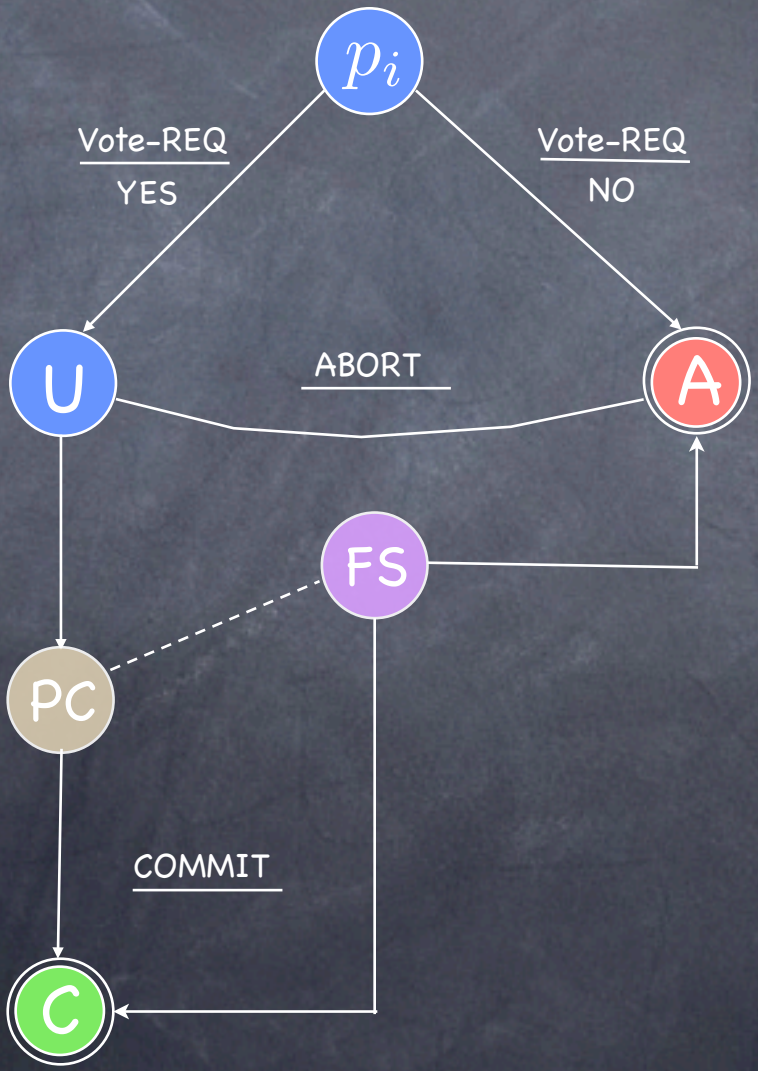


In state PC
a process knows that it
will commit unless it fails

2PC Revisited



2PC Revisited



3PC: The Protocol

Dale Skeen (1982)

I. c sends VOTE-REQ to all participants.

II. When p_i receives a VOTE-REQ, it responds by sending a vote to c
if $vote_i$ = No, then $decide_i$:= ABORT and p_i halts.

III. c collects votes from all (including itself).
if all votes are Yes, then c sends PRECOMMIT to all
else $decide_c$:= ABORT; sends ABORT to all who voted Yes halts

IV. if p_i receives PRECOMMIT then it sends ACK to c

V. c collects ACKs from all.
When all ACKs have been received, $decide_c$:= COMMIT;
c sends COMMIT to all.

VI. When p_i receives COMMIT, p_i sets $decide_i$:= COMMIT and halts.

Wait a minute!

1. c sends VOTE-REQ to all participants
2. When participant p_i receives a VOTE-REQ,
   it responds by sending a vote to c
   if $vote_i$ = No, then $decide_i$ = ABORT and p_i halts
3. c collects vote from all (including itself)
   if all votes are Yes, then c sends PRECOMMIT to all
   else $decide_c$ = ABORT; c sends ABORT to all who
   voted Yes
   c halts
4. if p_i receives PRECOMMIT then it sends ACK to c
5. c collects ACKs from all
   when all ACKs have been received, $decide_c$ = COMMIT
   c sends COMMIT to all
6. When p_i receives COMMIT, p_i sets $decide_i$ = COMMIT
   p_i halts

- Messages are known to the receiver before they are sent...so, why are they sent?

Wait a minute!

1. c sends VOTE-REQ to all participants
2. When participant p_i receives a VOTE-REQ,
   it responds by sending a vote to c
   if $vote_i$ = No, then $decide_i$ = ABORT and p_i halts
3. c collects vote from all (including itself).
   if all votes are Yes, then c sends PRECOMMIT to all
   else $decide_c$ = ABORT; c sends ABORT to all who
   voted Yes
   c halts
4. if p_i receives PRECOMMIT then it sends ACK to c
5. c collects ACKs from all
   when all ACKs have been received, $decide_c$ = COMMIT
   c sends COMMIT to all
6. When p_i receives COMMIT, p_i sets $decide_i$ = COMMIT
   p_i halts

- Messages are known to the receiver before they are sent...so, why are they sent?

They inform the recipient of the protocol's progress!

Wait a minute!

1. c sends VOTE-REQ to all participants
2. When participant p_i receives a VOTE-REQ,
   it responds by sending a vote to c
   if $vote_i$ = No, then $decide_i$ = ABORT and p_i halts
3. c collects vote from all (including itself).
   if all votes are Yes, then c sends PRECOMMIT to all
   else $decide_c$ = ABORT; c sends ABORT to all who
   voted Yes
   c halts
4. if p_i receives PRECOMMIT then it sends ACK to c
5. c collects ACKs from all
   when all ACKs have been received, $decide_c$ = COMMIT
   c sends COMMIT to all
6. When p_i receives COMMIT, p_i sets $decide_i$ = COMMIT
   p_i halts

- Messages are known to the receiver before they are sent...so, why are they sent?

They inform the recipient of the protocol's progress!

- When c receives ACK from p, it knows p is not uncertain

Wait a minute!

1. c sends VOTE-REQ to all participants
2. When participant p_i receives a VOTE-REQ,
   it responds by sending a vote to c
   if $vote_i$ = No, then $decide_i$ = ABORT and p_i halts
3. c collects vote from all (including itself).
   if all votes are Yes, then c sends PRECOMMIT to all
   else $decide_c$ = ABORT; c sends ABORT to all who
   voted Yes
   c halts
4. if p_i receives PRECOMMIT then it sends ACK to c
5. c collects ACKs from all
   when all ACKs have been received, $decide_c$ = COMMIT
   c sends COMMIT to all
6. When p_i receives COMMIT, p_i sets $decide_i$ = COMMIT
   p_i halts

- Messages are known to the receiver before they are sent...so, why are they sent?

They inform the recipient of the protocol's progress!

- When c receives ACK from p, it knows p is not uncertain
- When p receives COMMIT, it knows no participant is uncertain, so it can commit

Timeout Actions

Processes are waiting on steps 2, 3, 4, 5, and 6

Step 2 p_i is waiting for VOTE-REQ from coordinator	Step 3 Coordinator is waiting for vote from participants
Step 4 p_i waits for PRECOMMIT	Step 5 Coordinator waits for ACKs
Step 6 p_i waits for COMMIT	