Deterministic Process Groups in dOS

Tom Bergan
Nicholas Hunt, Luis Ceze, Steven D. Gribble

University of Washington

A Nondeterministic Program

```
global x=0
```

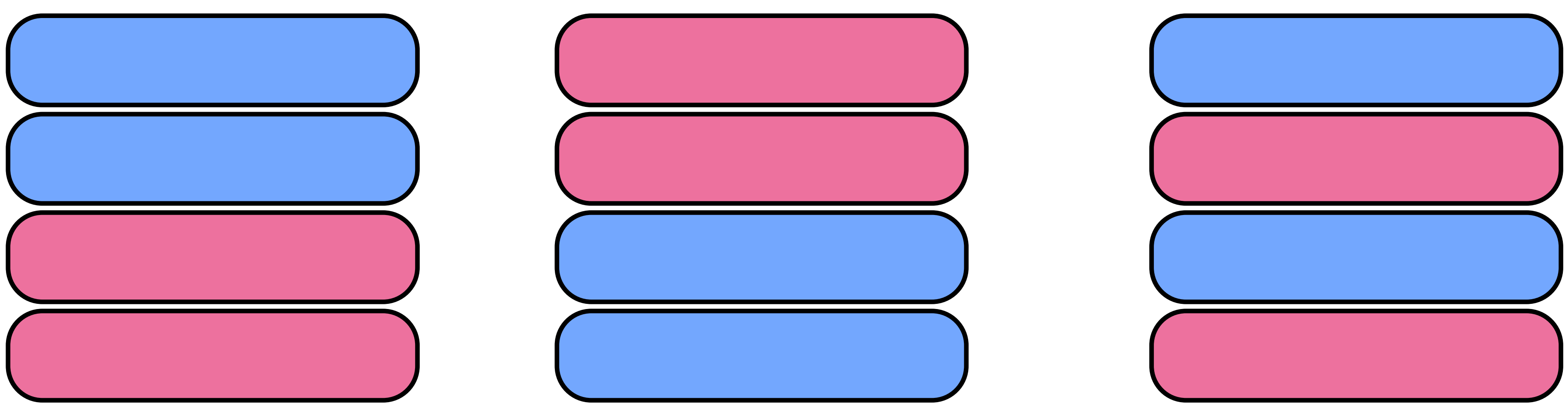
Thread 1

```
t := x
x := t + 1
```

Thread 2

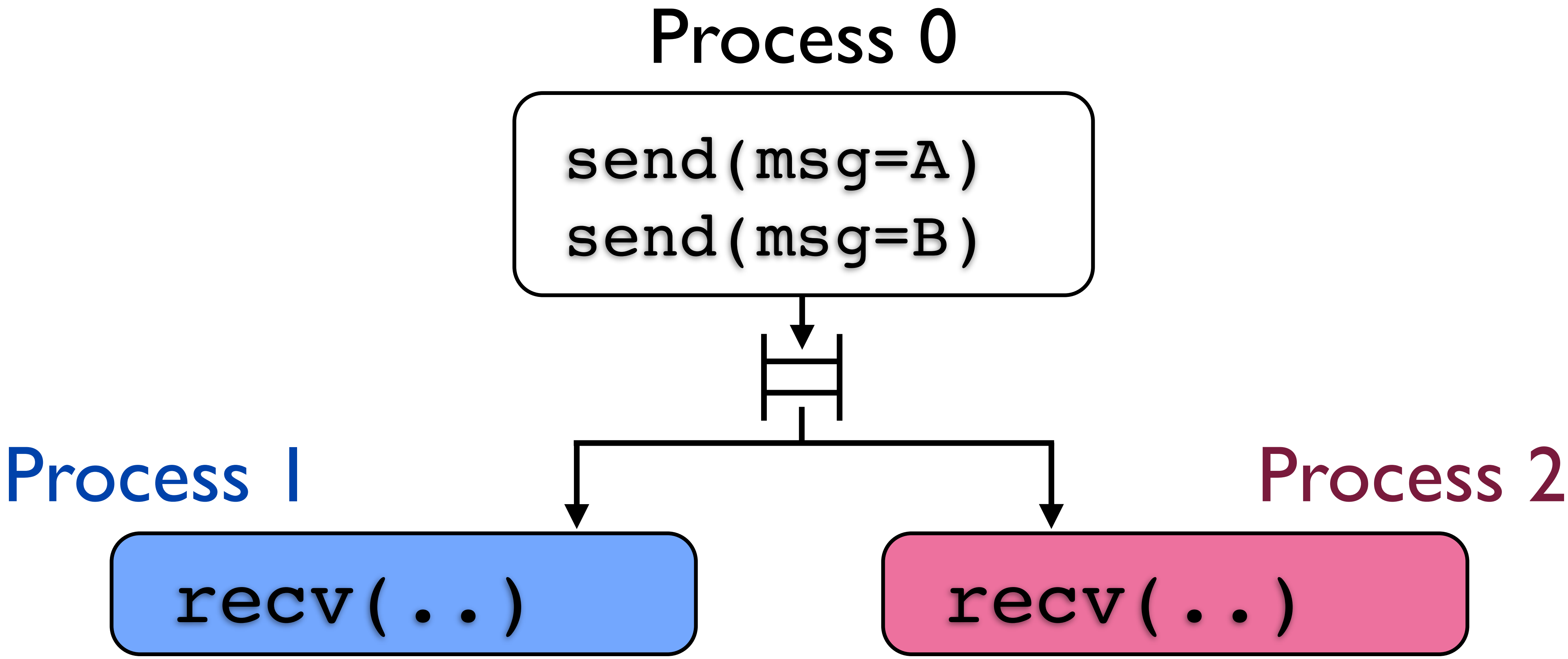
```
t := x
x := t + 1
```

What is x?

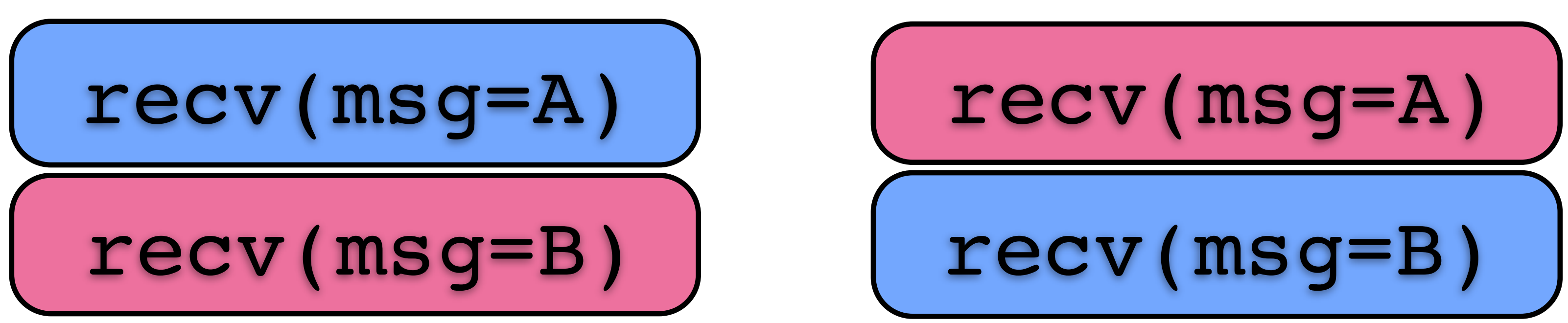


`x == 2` `x == 2` **`x == 1`**

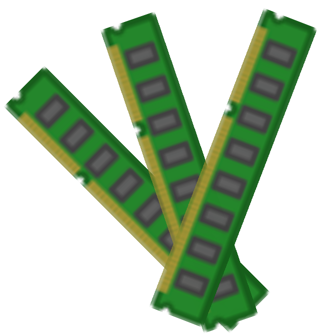
Nondeterministic IPC



Who gets msg A?



Nondeterminism In Real Systems

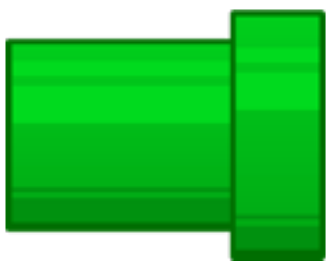


shared-memory

why nondeterministic:
multiprocessor hardware is unpredictable

disks

why nondeterministic:
drive latency is unpredictable

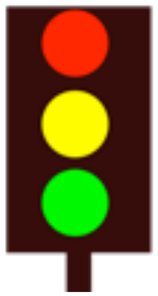


IPC (e.g. pipes)

why nondeterministic:
multiprocessor hardware is unpredictable

network

why nondeterministic:
packets arrive from external sources



posix signals

why nondeterministic:
unpredictable scheduling, also can be triggered by users

. . .

The Problem

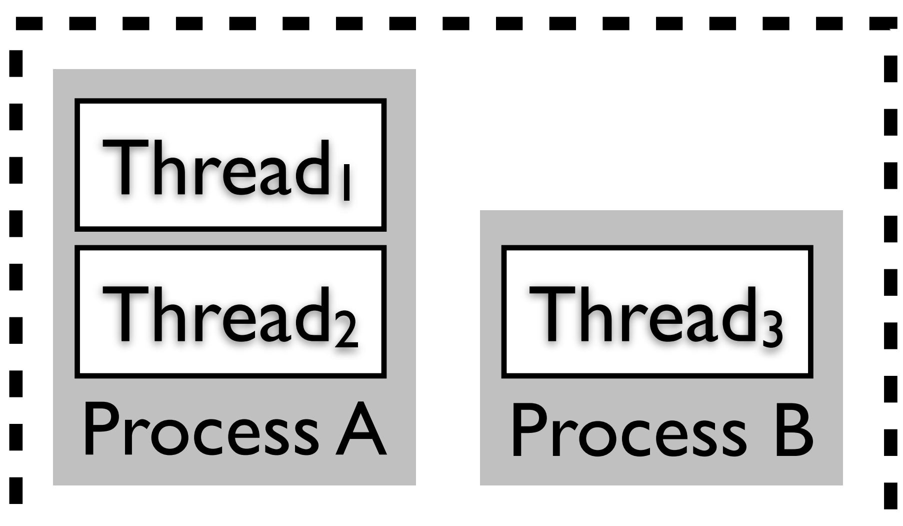
- Nondeterminism makes programs . . .
  - ➡ hard to test
    - same input, different outputs
  - ➡ hard to debug
    - leads to heisenbugs
  - ➡ hard to replicate for fault-tolerance
    - replicas get out of sync
- Multiprocessors make this problem much worse!

Our Solution

- OS support for deterministic execution
  - ➡ of arbitrary programs
  - ➡ attack *all* sources of nondeterminism (not just shared-memory)
  - ➡ even on multiprocessors

New OS abstraction:

Deterministic Process Group (DPG)



deterministic box

Key Questions

1. **What can be made deterministic?**
2. **What can we do about the remaining sources of nondeterminism?**

Key Questions

1. **What can be made deterministic?**
   - distinguish *internal* vs. *external* nondeterminism
2. **What can we do about the remaining sources of nondeterminism?**

Internal nondeterminism

- arises from scheduling artifacts (hw timing, etc)

NOT Fundamental

can be eliminated!

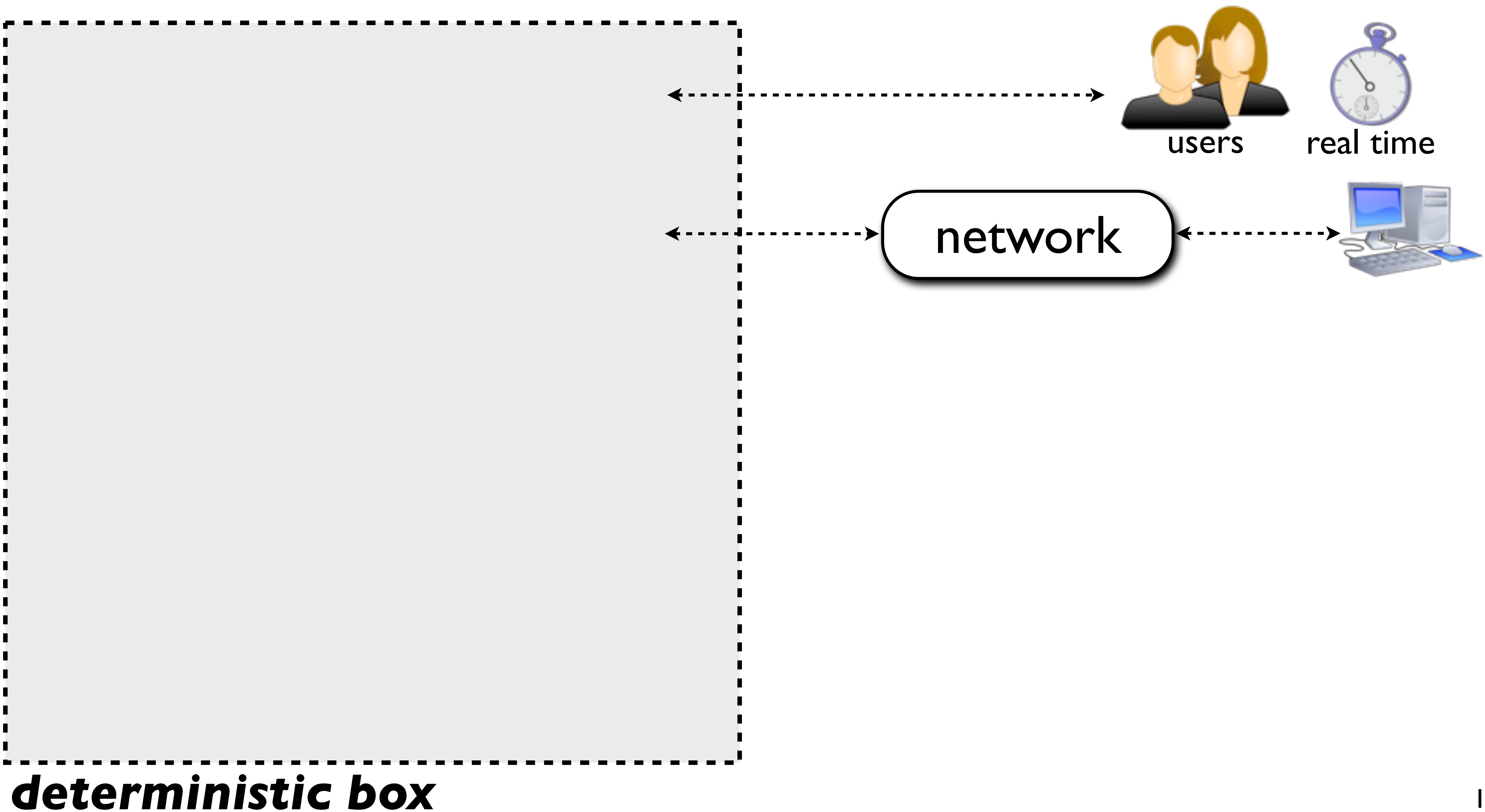
External nondeterminism

- arises from interactions with the external world (networks, users, etc)

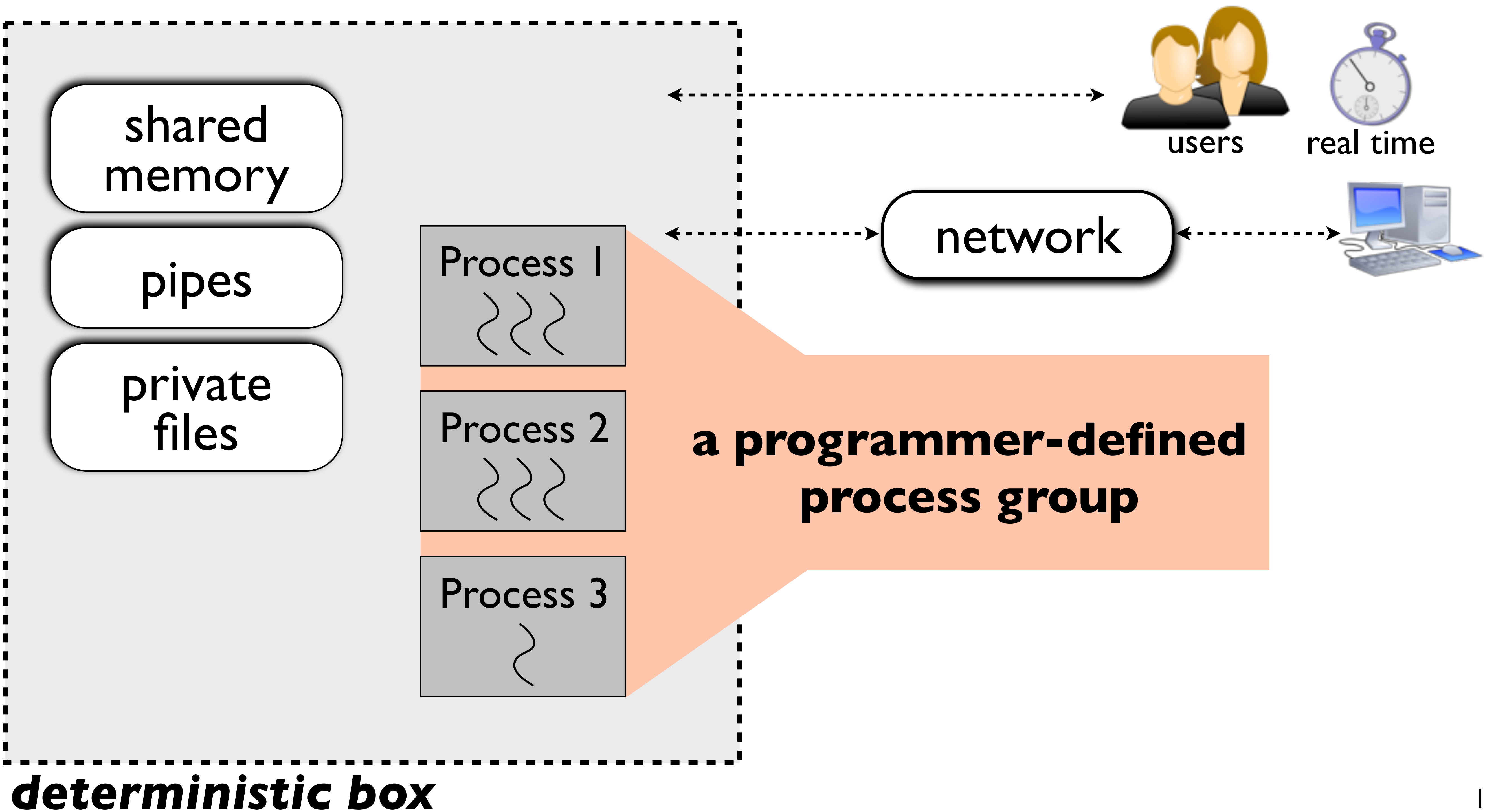
Fundamental

can not be eliminated

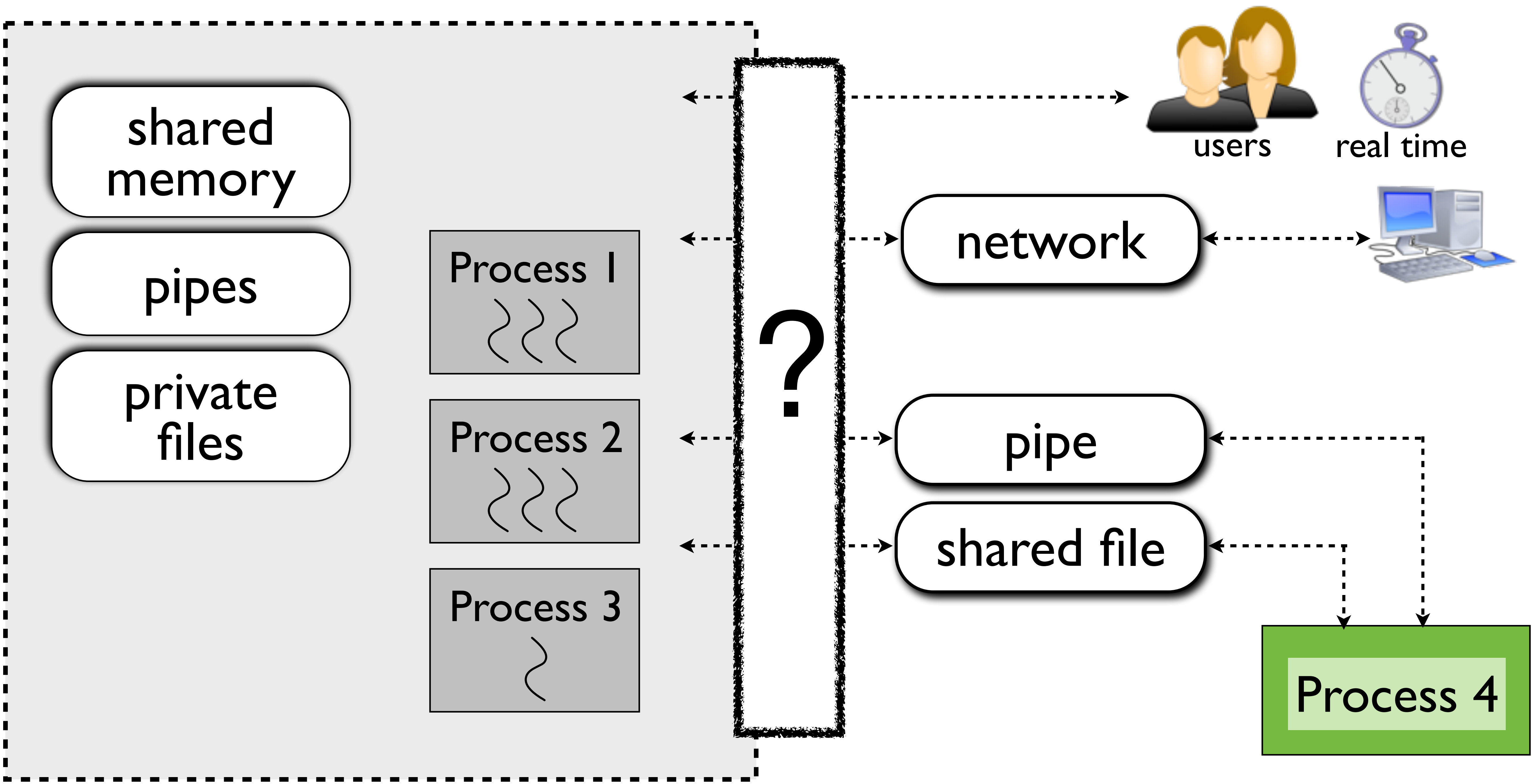
Internal Determinism	External Nondeterminism



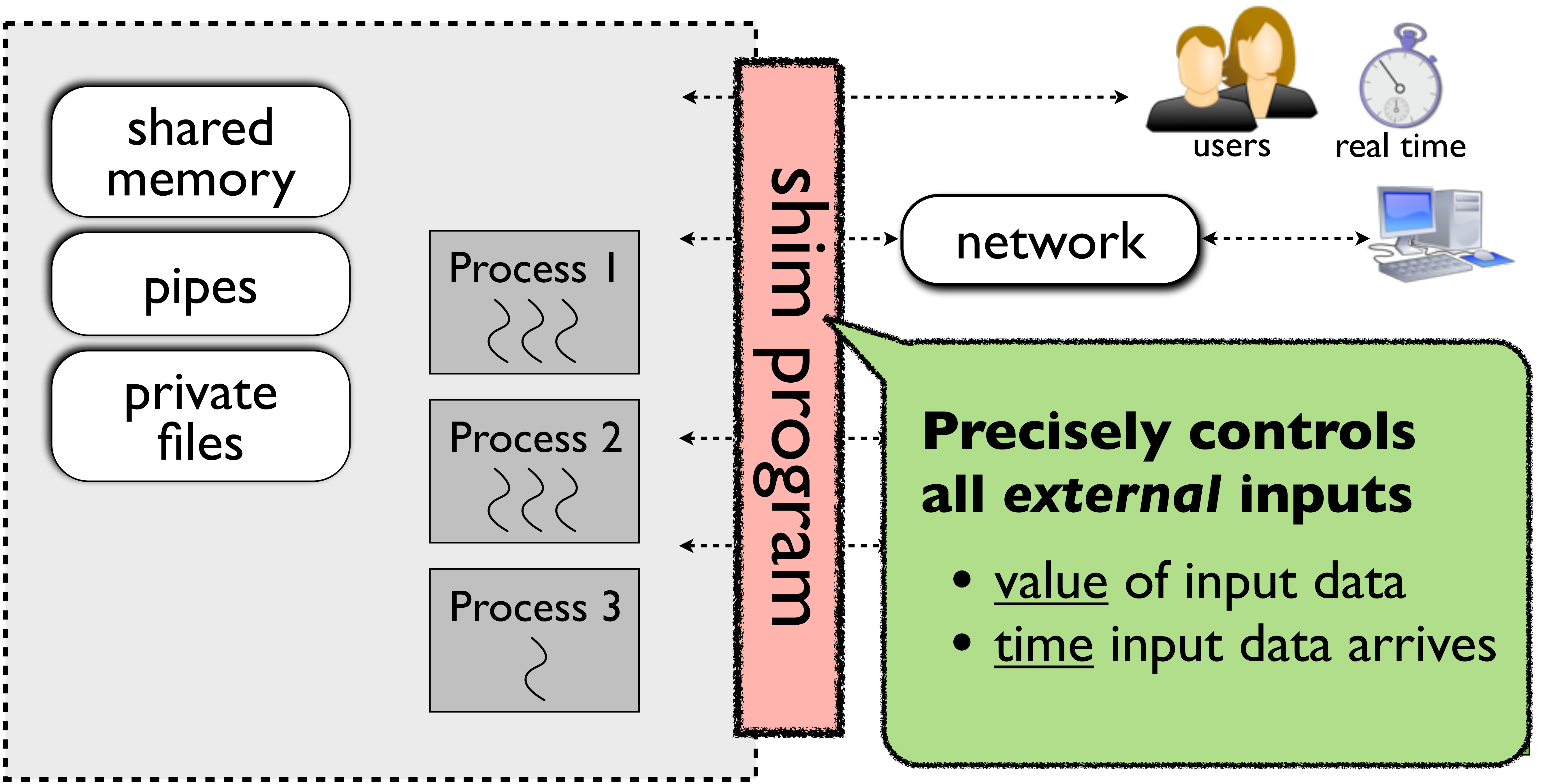
Internal Determinism | External Nondeterminism



Internal Determinism | External Nondeterminism

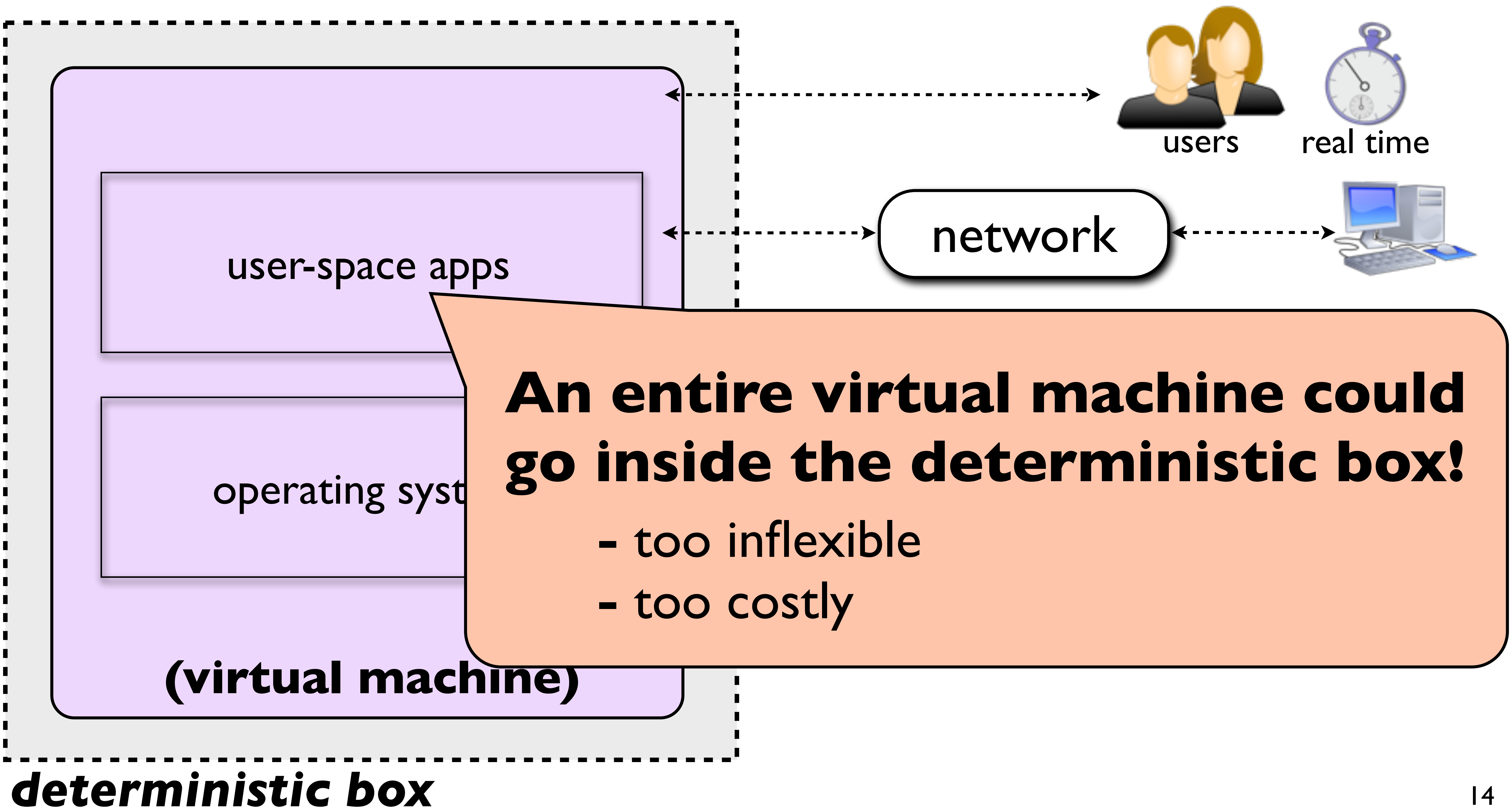


Internal Determinism | External Nondeterminism

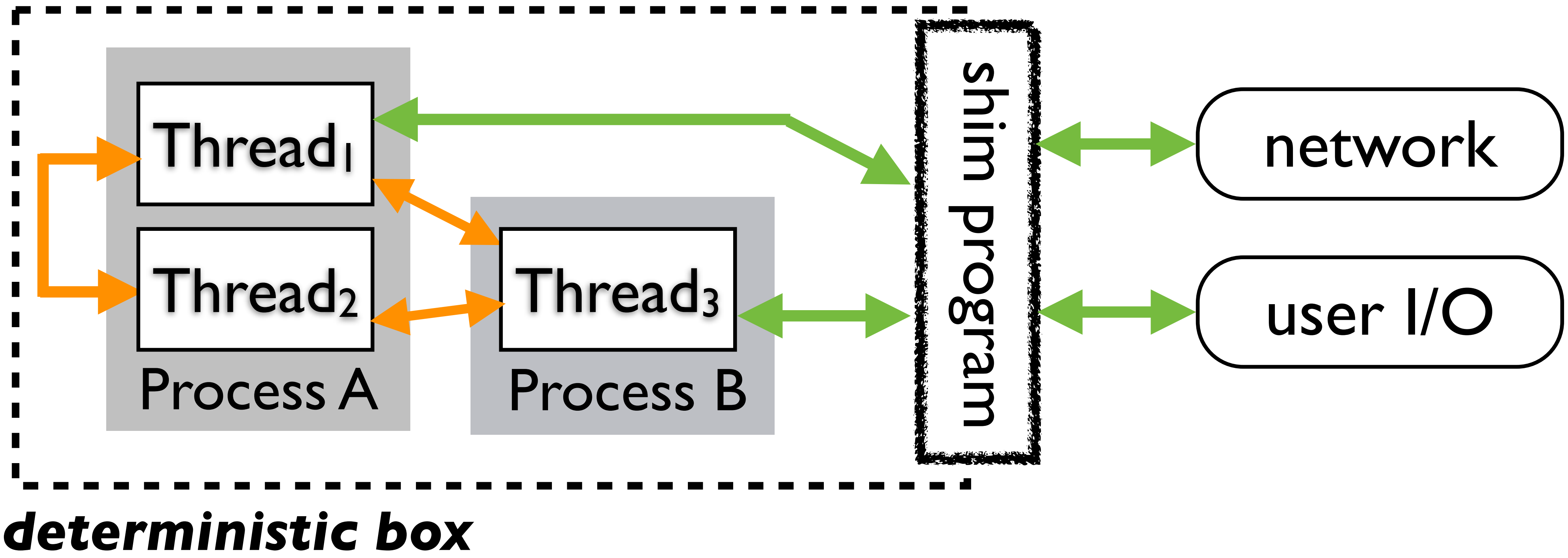


deterministic box

Internal Determinism | External Nondeterminism



Deterministic Process Groups



OS ensures:

- *internal* nondeterminism is eliminated
  (for shared-memory, pipes, signals, local files, ...)
- *external* nondeterminism funneled through *shim program*

Shim Program:

- user-space program that precisely *controls* all external nondeterministic inputs

Contributions

Conceptual:

- identify *internal* vs. *external* nondeterminism
- key: *internal* nondeterminism can be eliminated!

Abstraction:

- Deterministic Process Groups (DPGs)
- control *external* nondeterminism via a <u>shim program</u>

Implementation:

- dOS, a modified version of Linux
- supports arbitrary, unmodified binaries

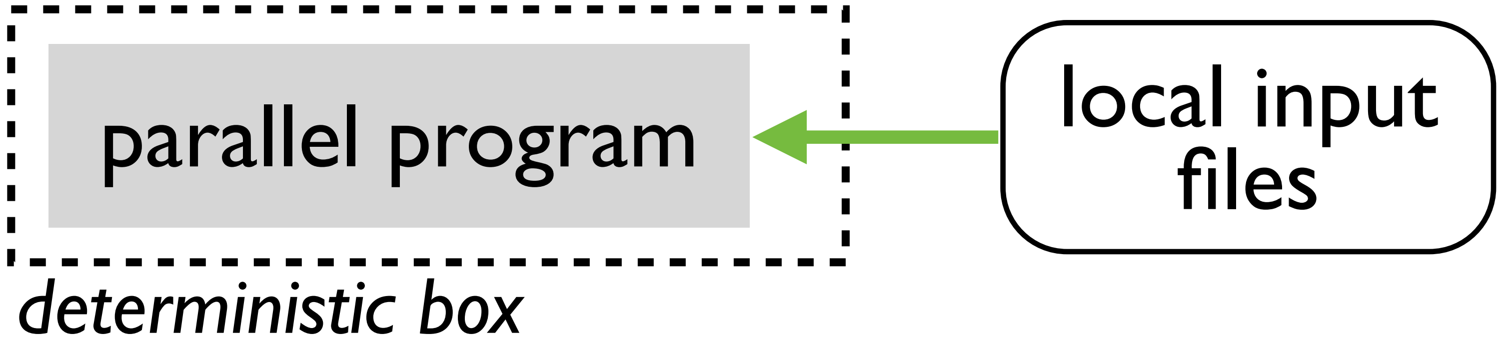
Applications:

- deterministic parallel execution
- record/replay
- replicated execution

Outline

- Example Uses
  - ➡ a parallel computation
  - ➡ a webserver
- Deterministic Process Groups
  - ➡ system interface
  - ➡ conceptual model
- dOS: our Linux-Based Implementation
- Evaluation

A Parallel Computation

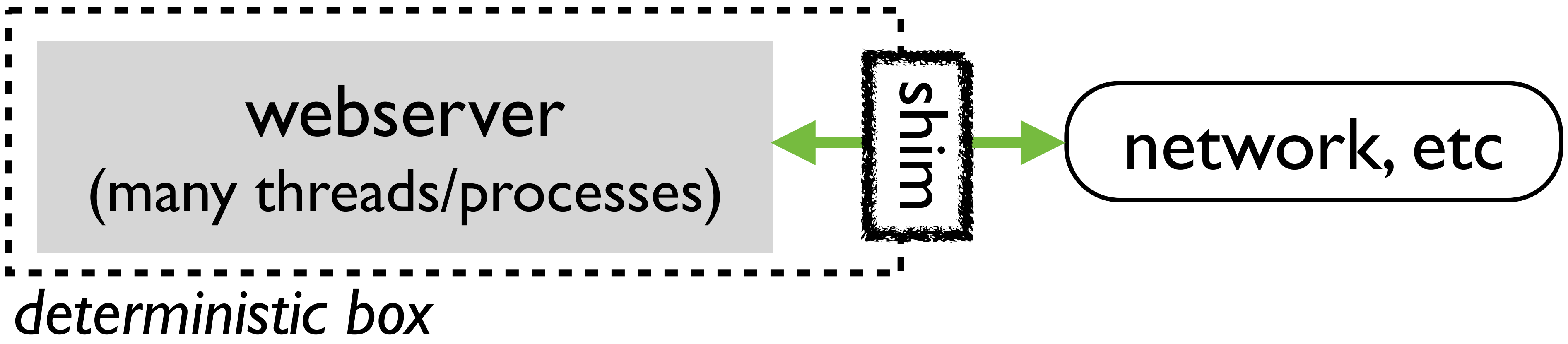


This program executes deterministically!

- even on a multiprocessor
- supports parallel programs written in *any* language

- no heisenbugs!
- test *input files*, not interleavings

A Webserver



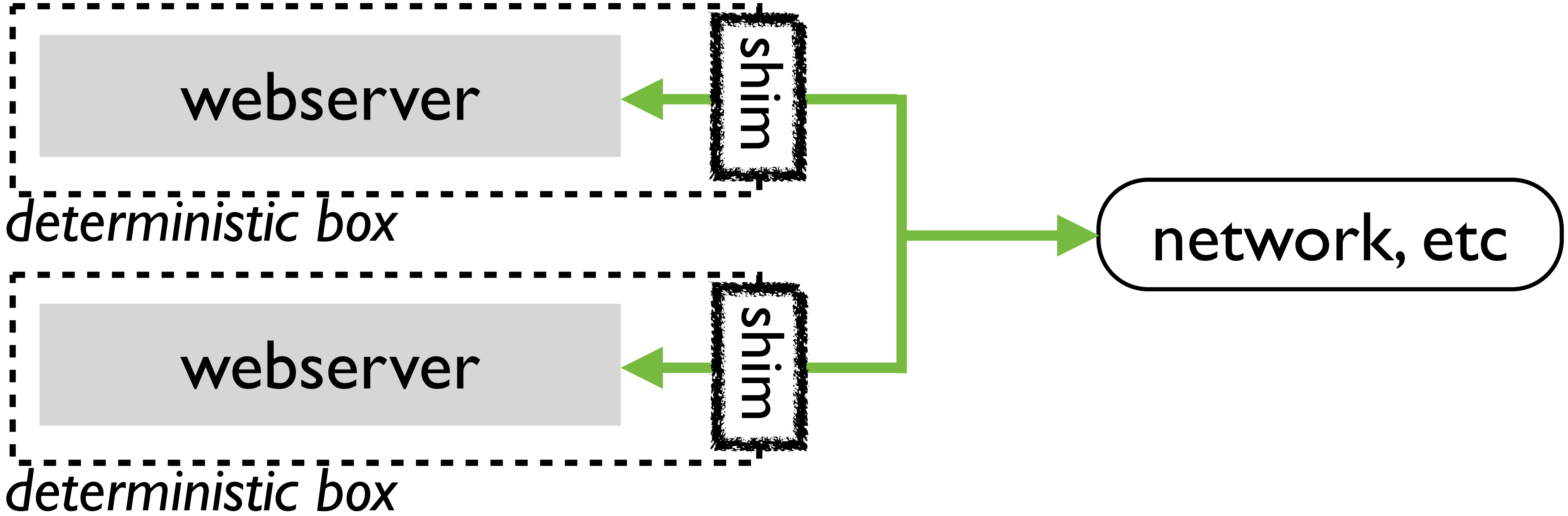
Deterministic Record/Replay

- implement in shim program
- requires no webserver modification

Advantages

- significantly less to log (*internal* nondeterminism is eliminated)
- log sizes 1,000x smaller!

A Webserver



Fault-tolerant Replication

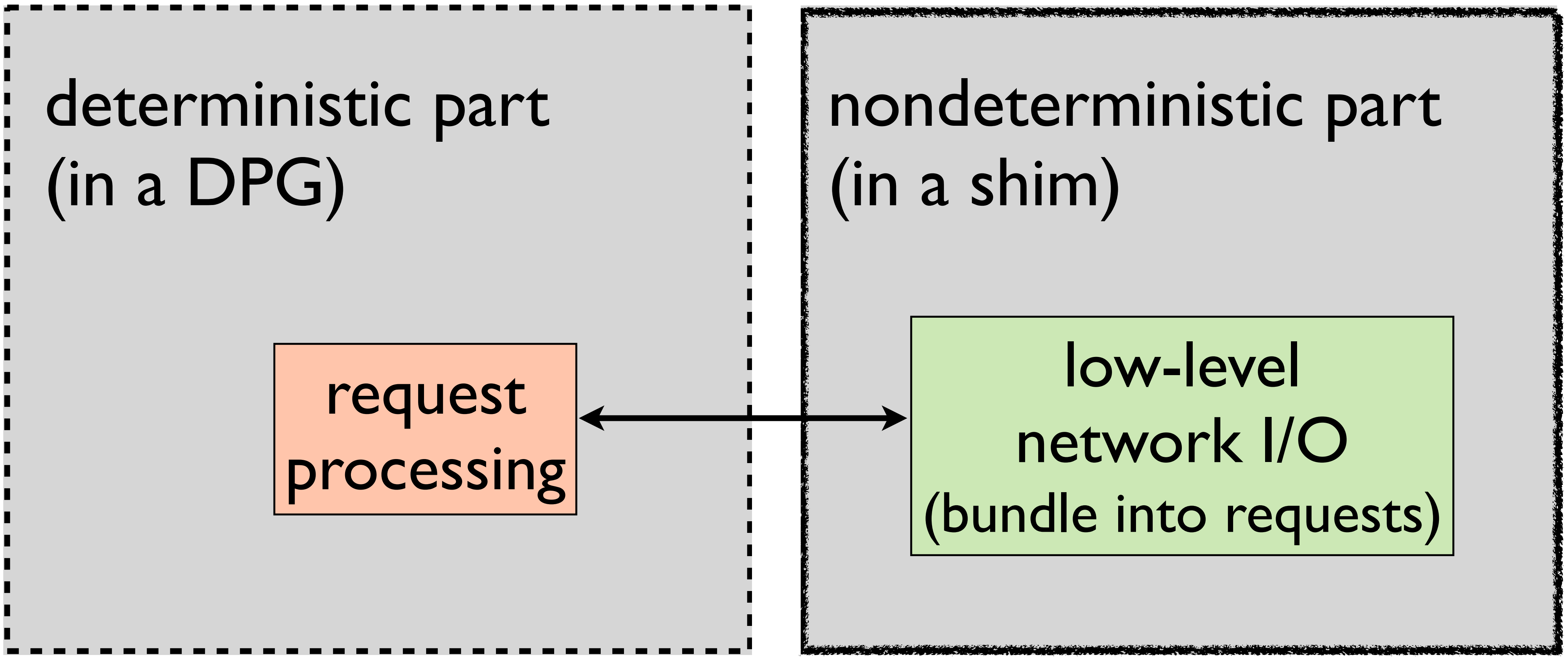
- implement replication protocol in shim programs (paxos, virtual synchrony, etc)

Advantage

- easy to replicate multithreaded servers (*internal* nondeterminism is eliminated)

A Webserver

Using DPGs to construct applications



webserver

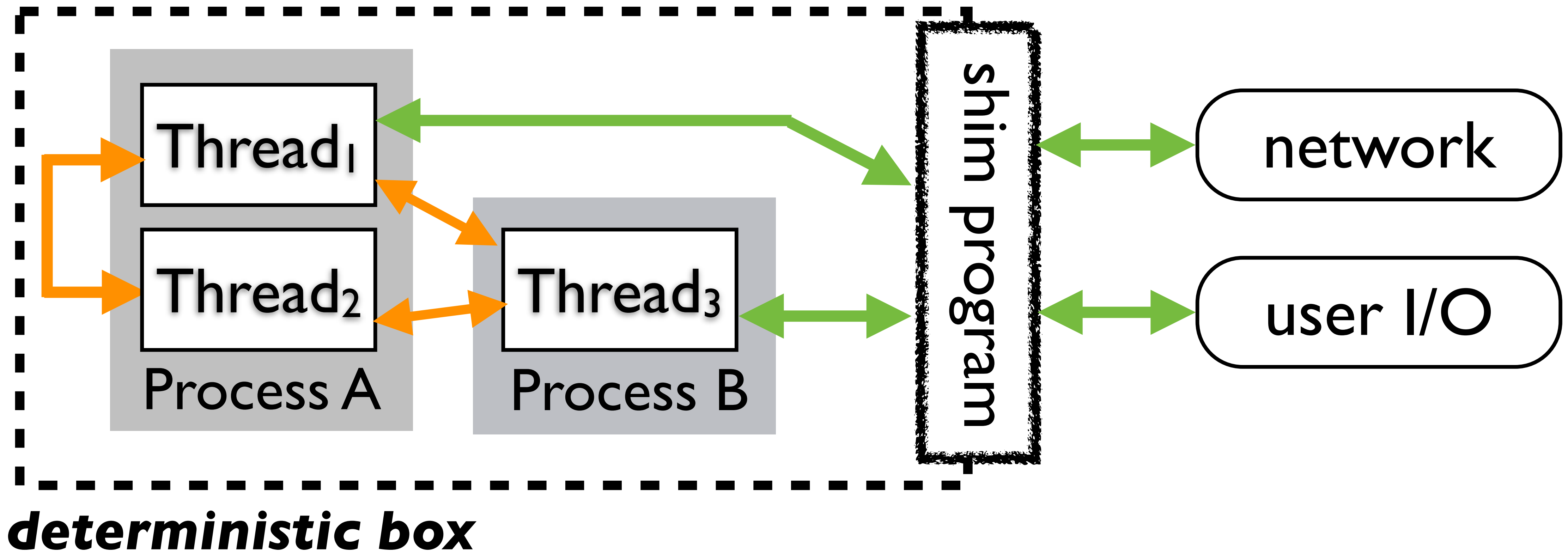
- behaves deterministically w.r.t. *requests* rather than *packets*

Shim program defines the nondeterministic interface

Outline

- Example Uses
  - ➡ a parallel computation
  - ➡ a webserver
- Deterministic Process Groups
  - ➡ system interface
  - ➡ conceptual model
- dOS: our Linux-Based Implementation
- Evaluation

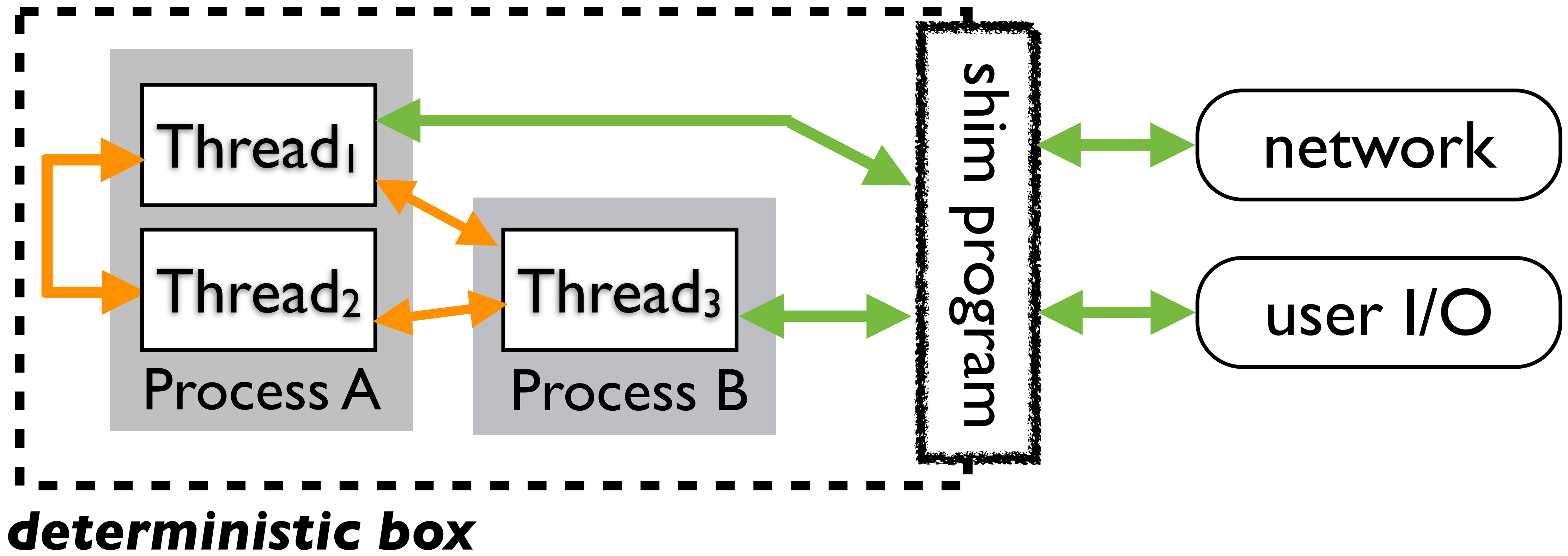
Deterministic Process Groups



System Interface

- New system call creates a new DPG: `sys_makedet()`
  - *DPG expands to include all child processes*
- Just like ordinary linux processes
  - *same system calls, signals, and hw instruction set*
  - *can be multithreaded*

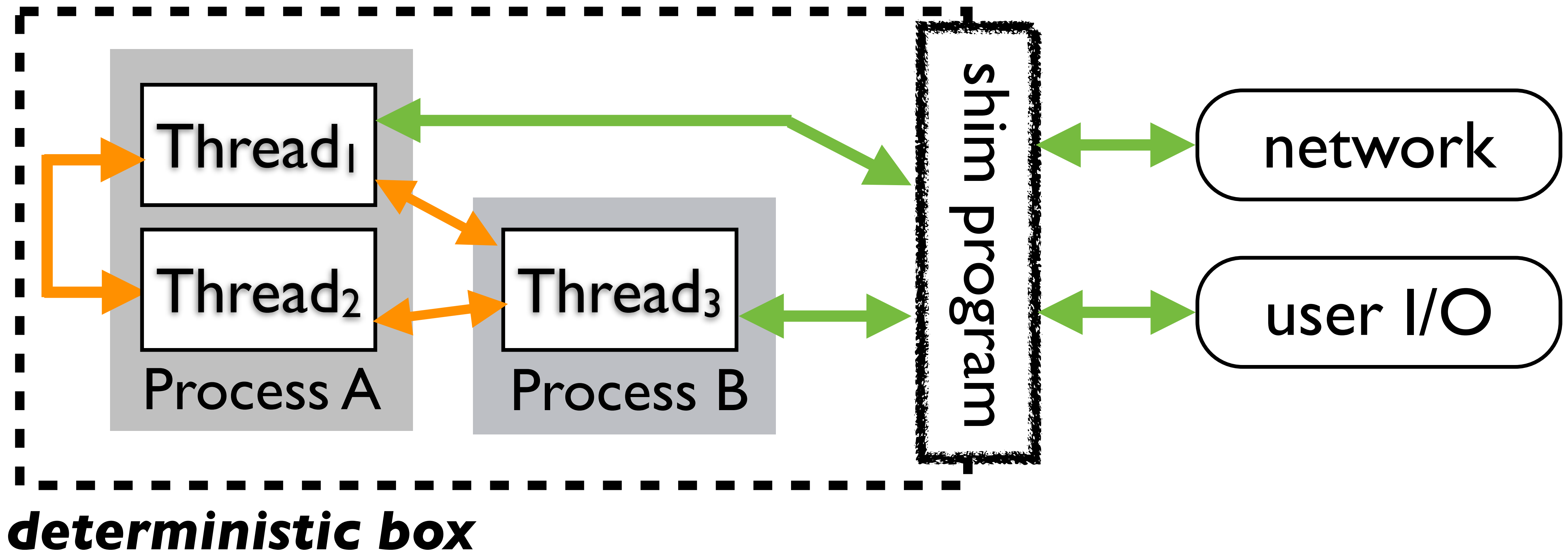
Deterministic Process Groups



Two questions:

- What are the semantics of *internal* determinism?
- How do shim programs work?

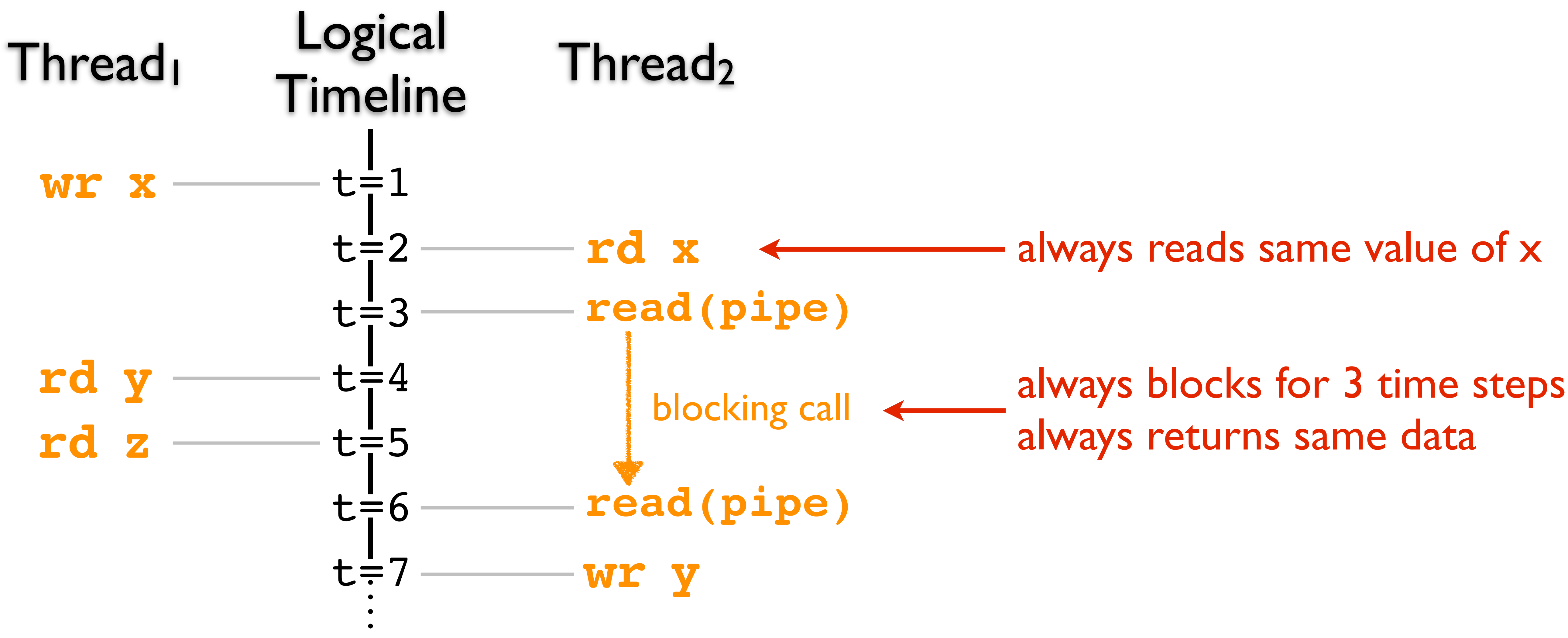
Deterministic Process Groups



Internal Determinism

- OS guarantees internal communication is scheduled *deterministically*
- Conceptually: executes as if serialized onto a *logical timeline*
  - *implementation is parallel*

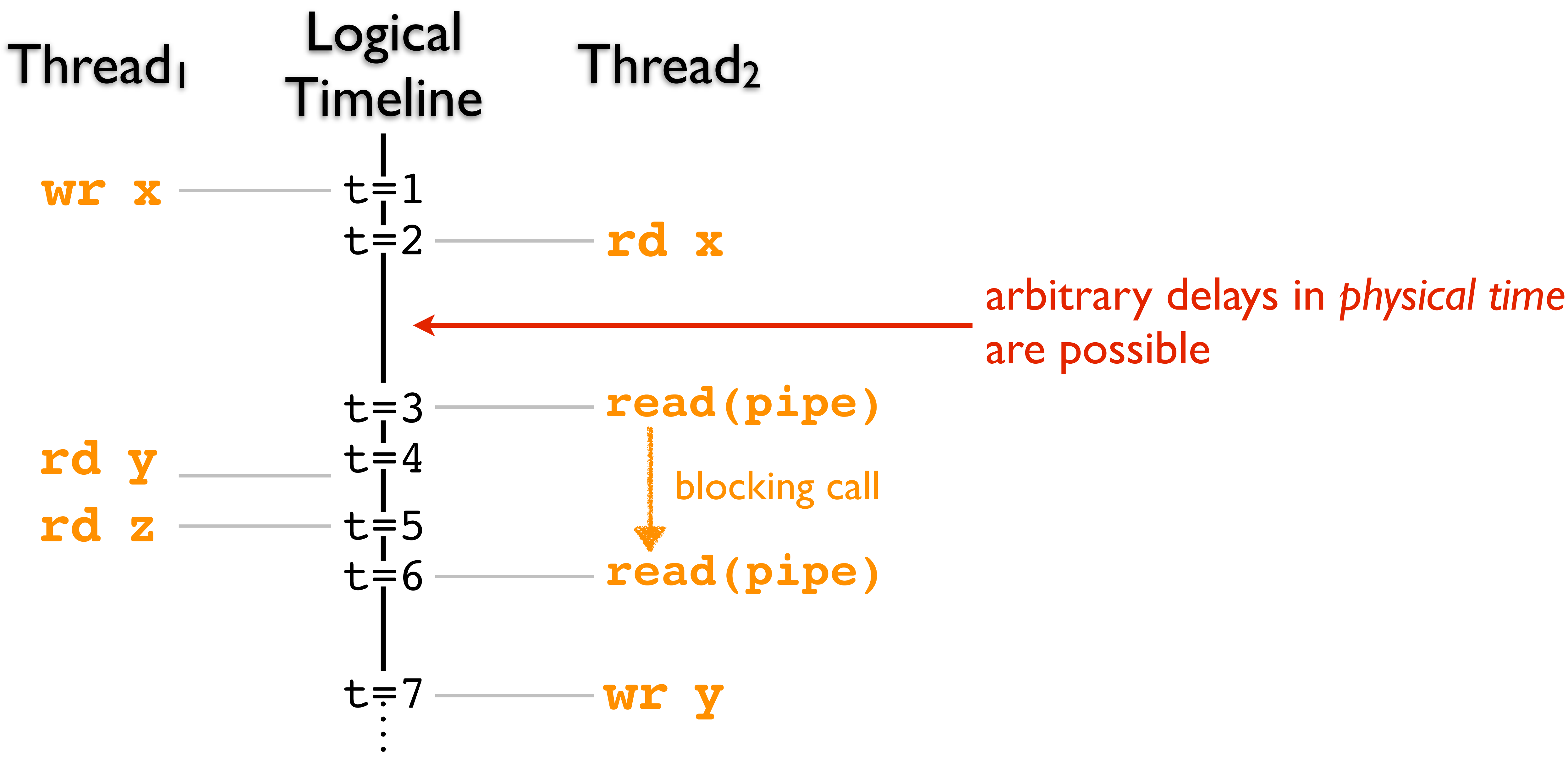
Internal Determinism



Each DPG has a *logical timeline*

- instructions execute as if serialized onto the logical timeline
- internal events are deterministic

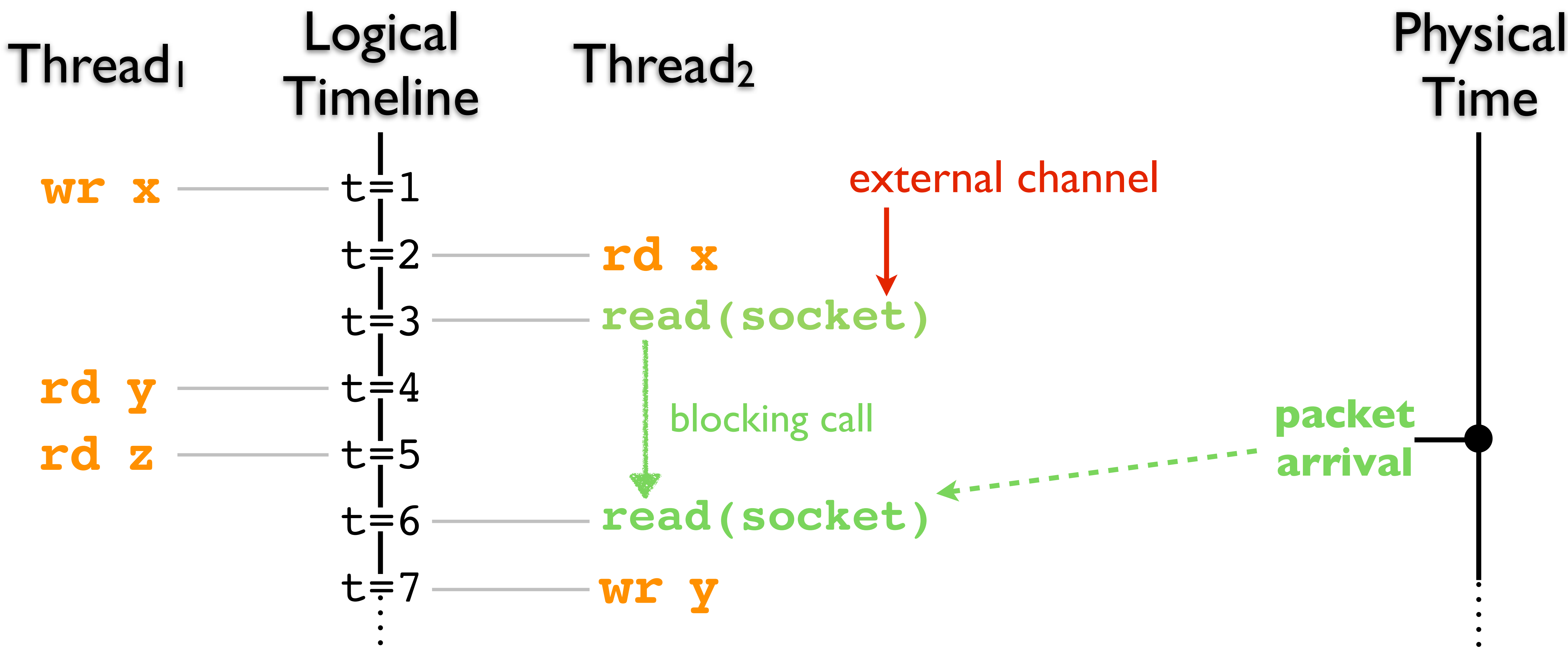
Internal Determinism



Physical time **is <u>not</u> deterministic**

- deterministic *results*, but not deterministic *performance*

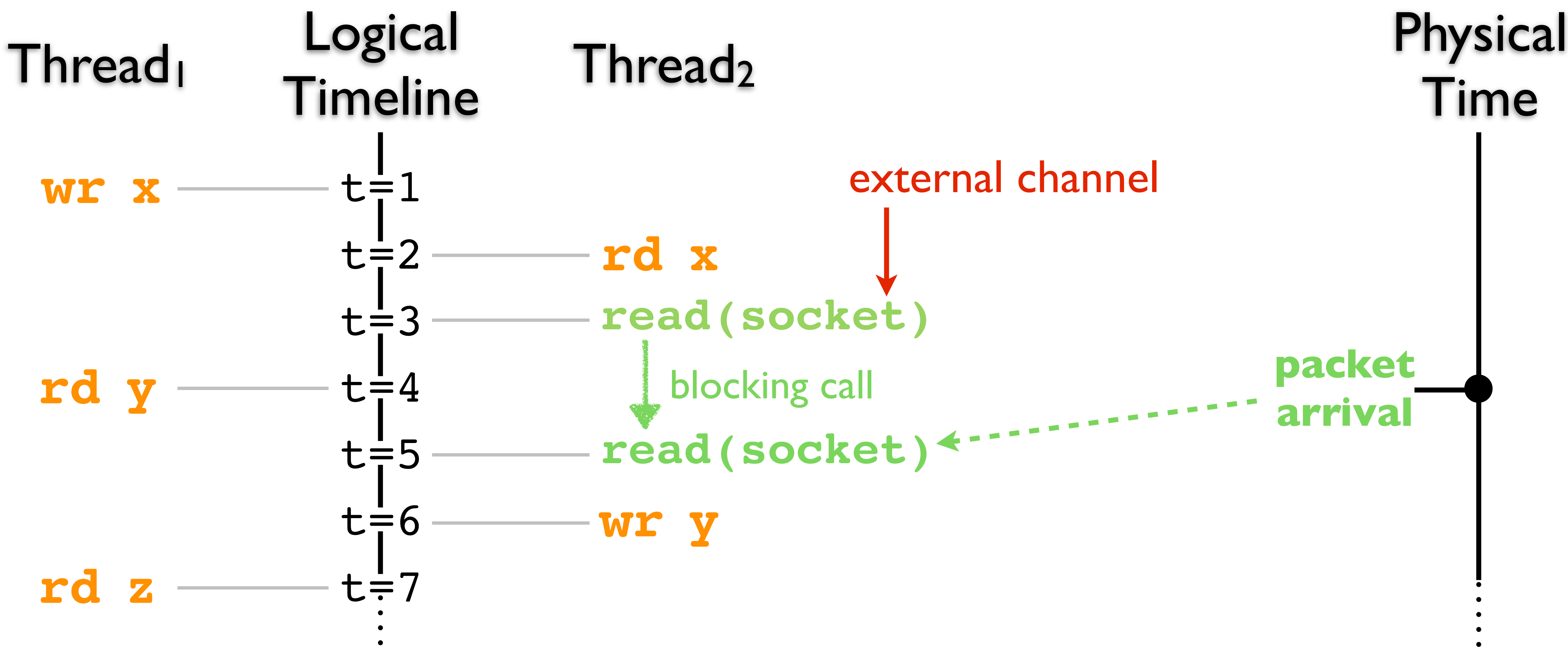
External Nondeterminism



Two sources of nondeterminism:

- data returned by `read()`
- blocking time of `read()`

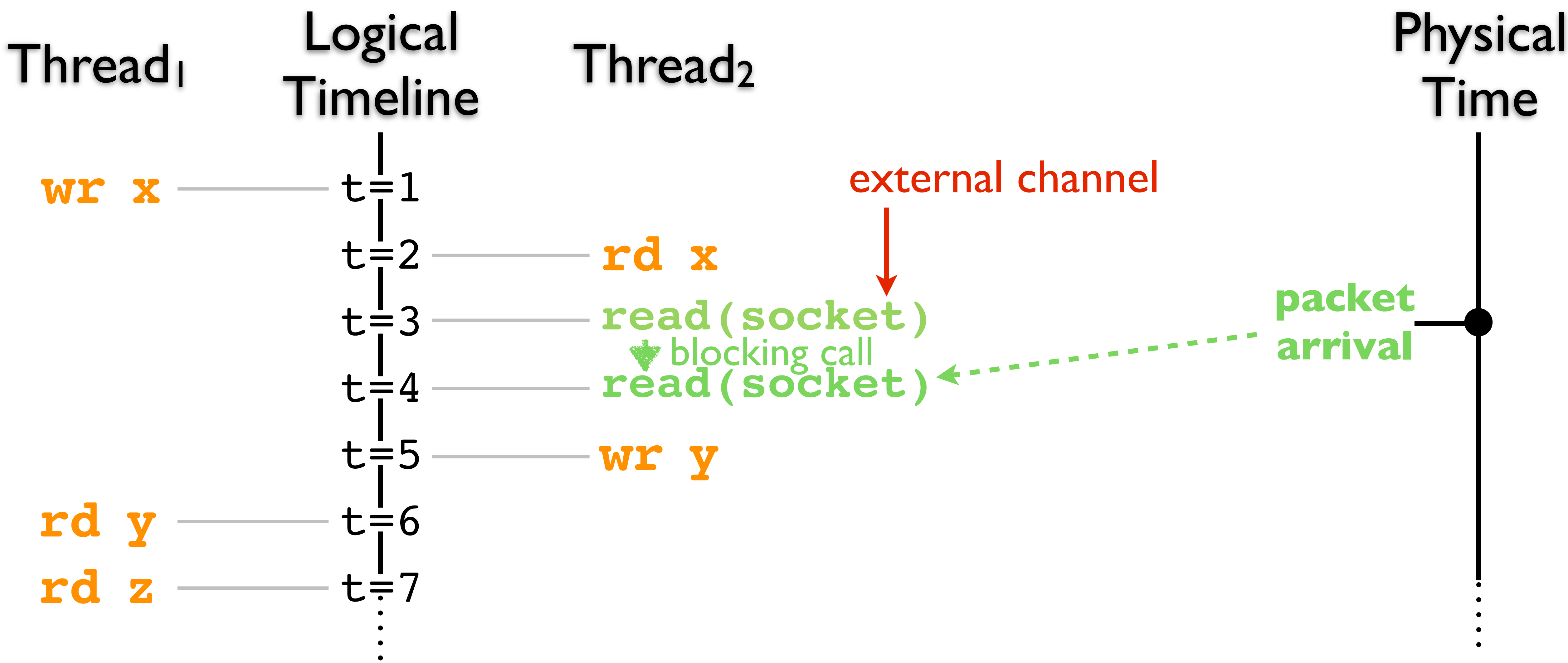
External Nondeterminism



Two sources of nondeterminism:

- data returned by `read()`
- blocking time of `read()`

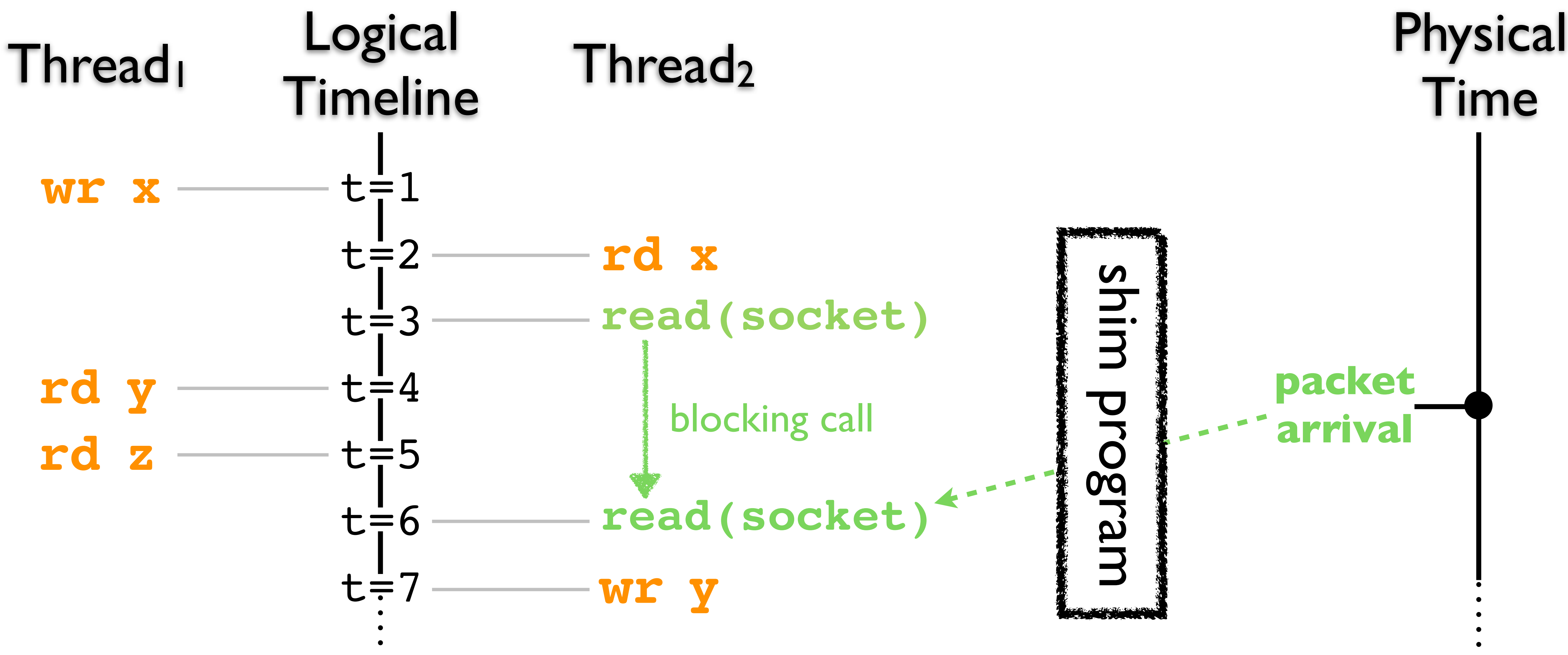
External Nondeterminism



Two sources of nondeterminism:

- data returned by `read()`
- blocking time of `read()`

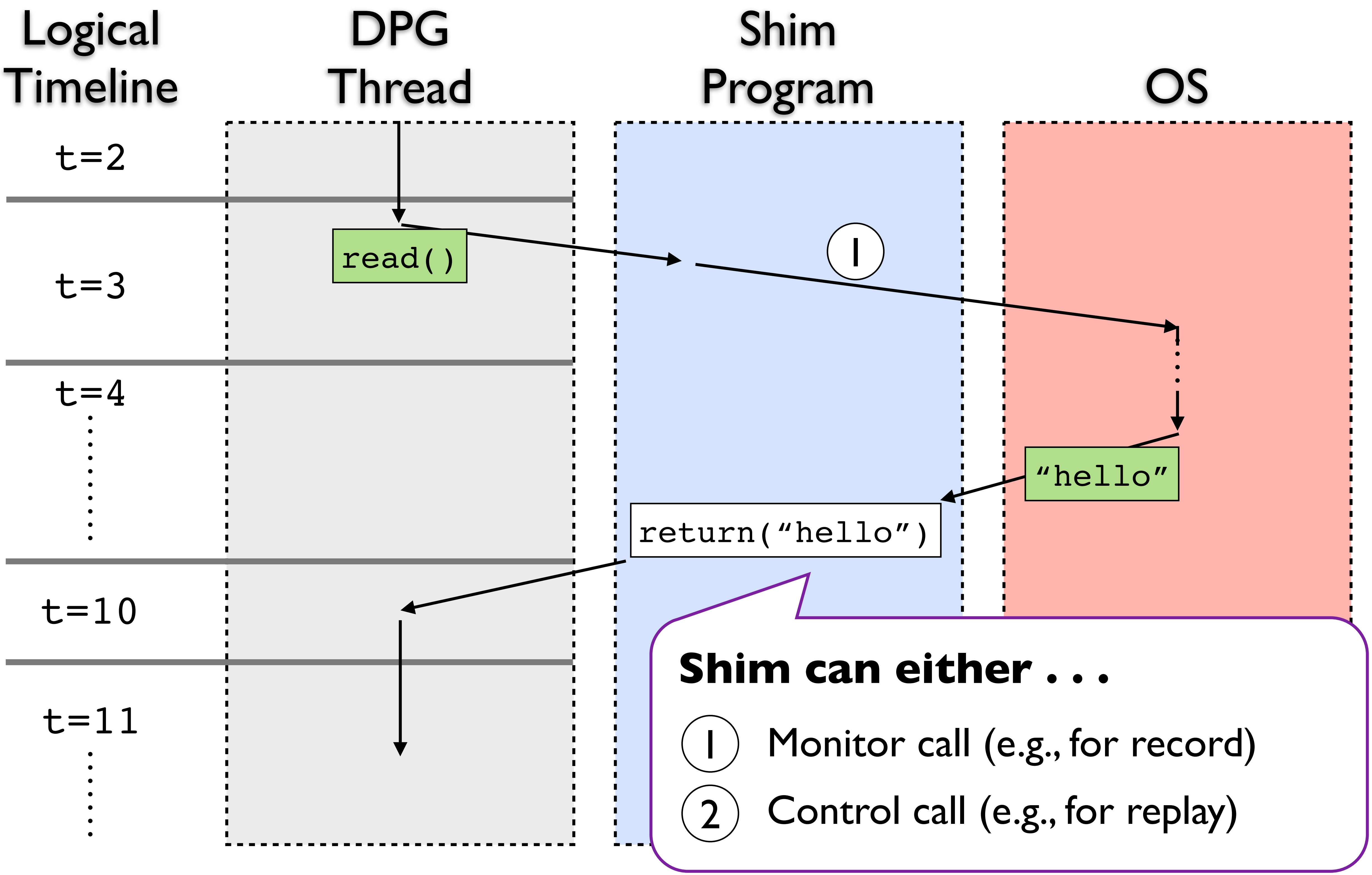
External Nondeterminism



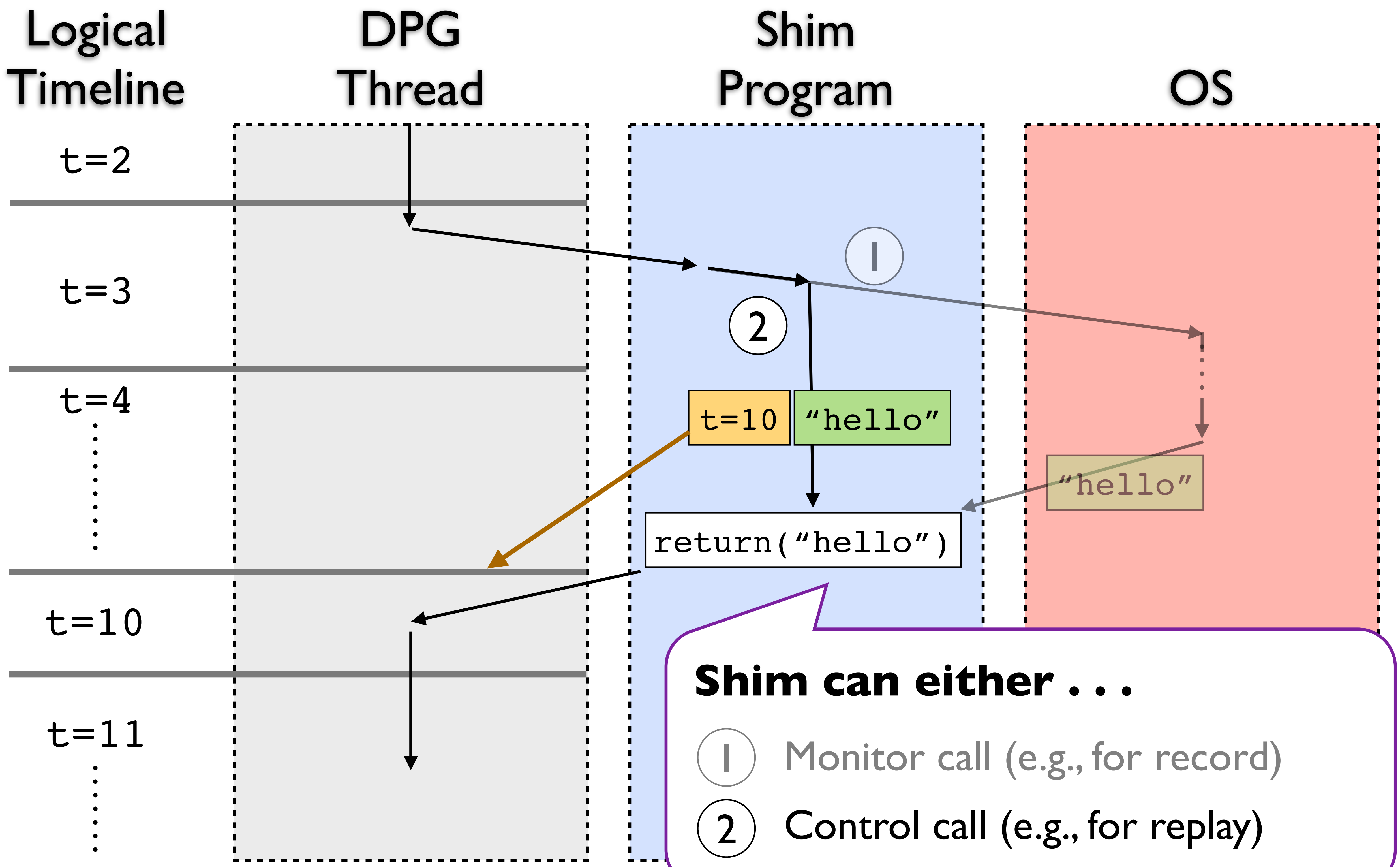
Two sources of nondeterminism:

- <u>data</u> returned by `read()` ‣ the **<u>what</u>**
- <u>blocking time</u> of `read()` ‣ the **<u>when</u>**

Shim Example: Read Syscall



Shim Example: Read Syscall



Shim can either . . .

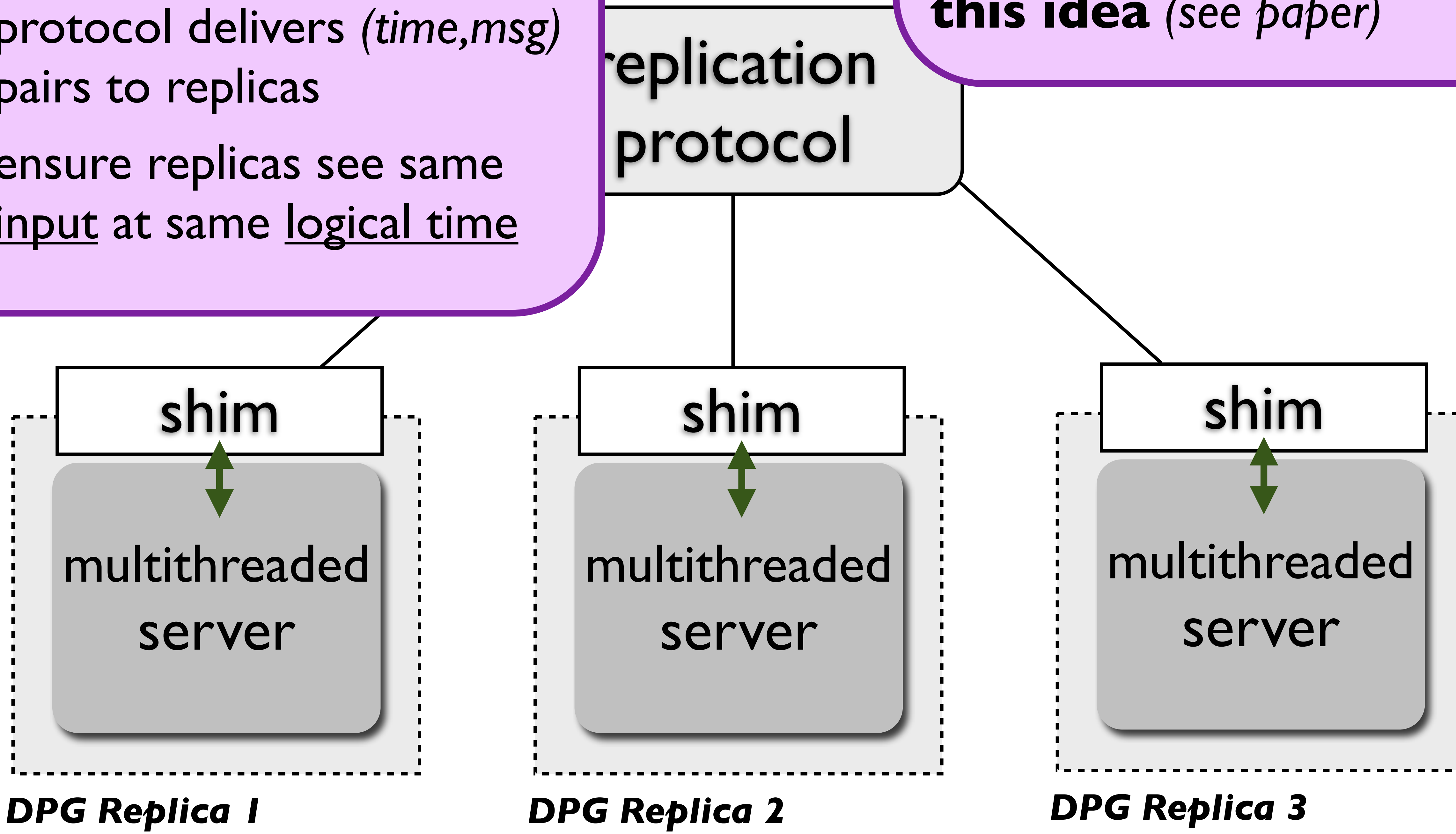
1. Monitor call (e.g., for record)
2. Control call (e.g., for replay)

Shim Example: Replication

Key idea:

- protocol delivers *(time,msg)* pairs to replicas
- ensure replicas see same <u>input</u> at same <u>logical time</u>

We have implemented this idea *(see paper)*



Outline

- Example Uses
    - ➡ a parallel computation
    - ➡ a webserver
- Deterministic Process Groups
    - ➡ system interface
    - ➡ conceptual model
- dOS: our Linux-Based Implementation
- Evaluation

dOS Overview

Modified version of Linux 2.6.24/x86_64

- ~8,000 lines of code added or modified
- ~50 files changed or modified
- transparently supports unmodified binaries

Support for DPGs:

- implement a deterministic scheduler
- implement an API for writing shim programs
- subsystems modified:
    - thread scheduling
    - virtual memory
    - system call entry/exit

talk focus

Paper describes challenges in depth

dOS: Deterministic Scheduler

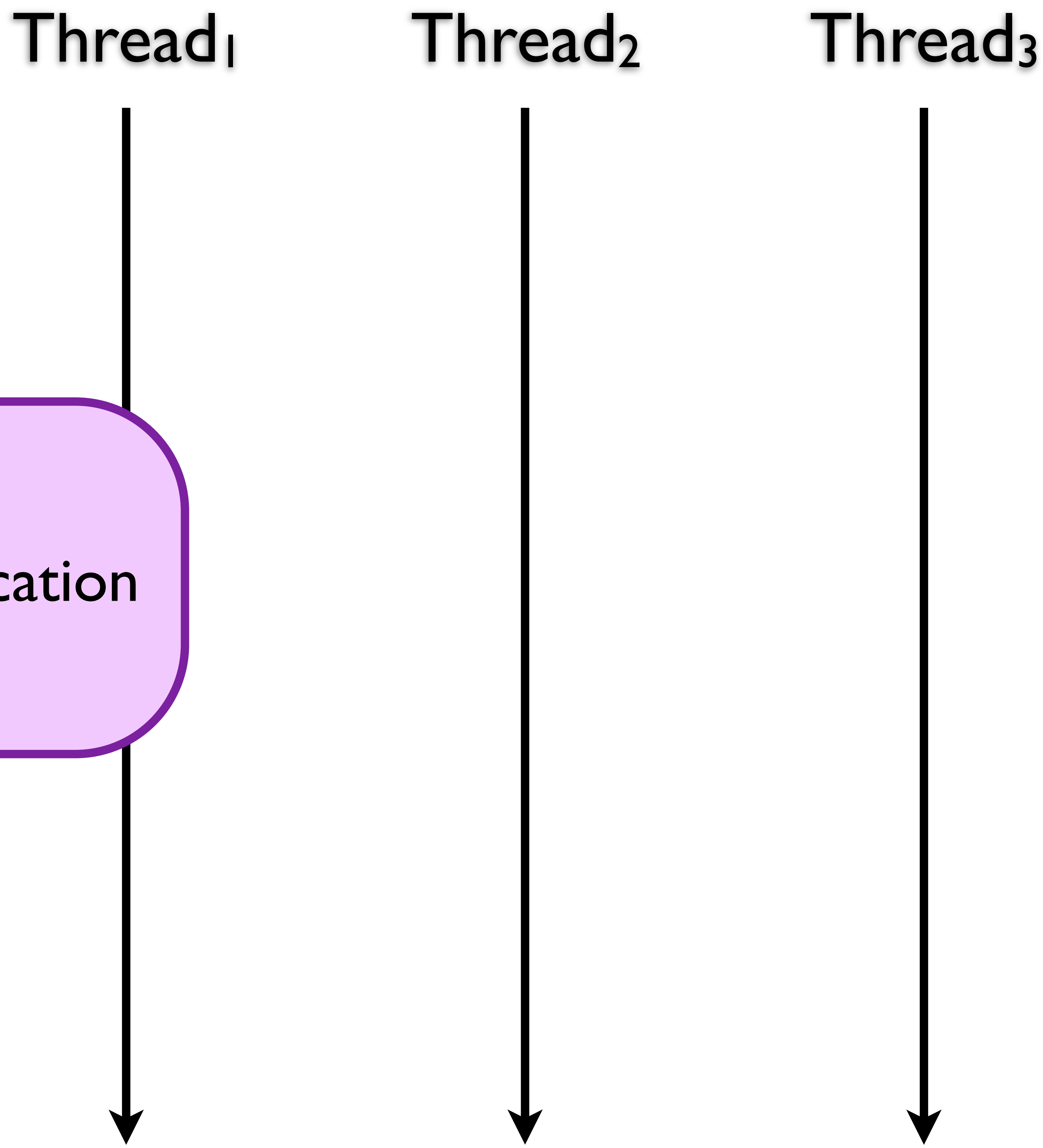
Which deterministic execution algorithm?

- DMP-O, from prior work [Asplos09, Asplos10]
    - other algorithms have better scalability, but
    - . . . Dmp-O is easiest to implement

How does DMP-O work?

How does dOS implement DMP-O?

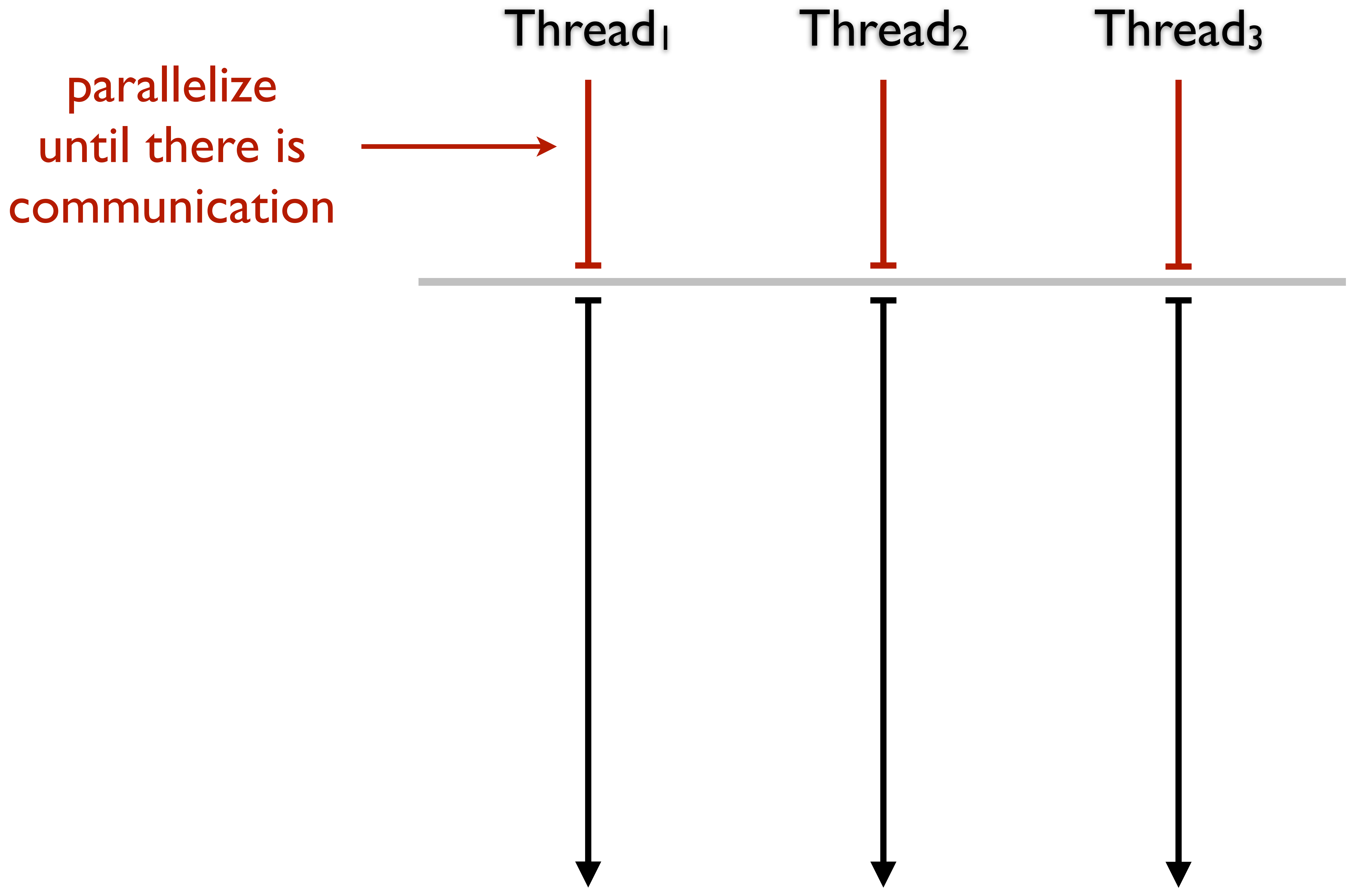
Deterministic Execution with DMP-O



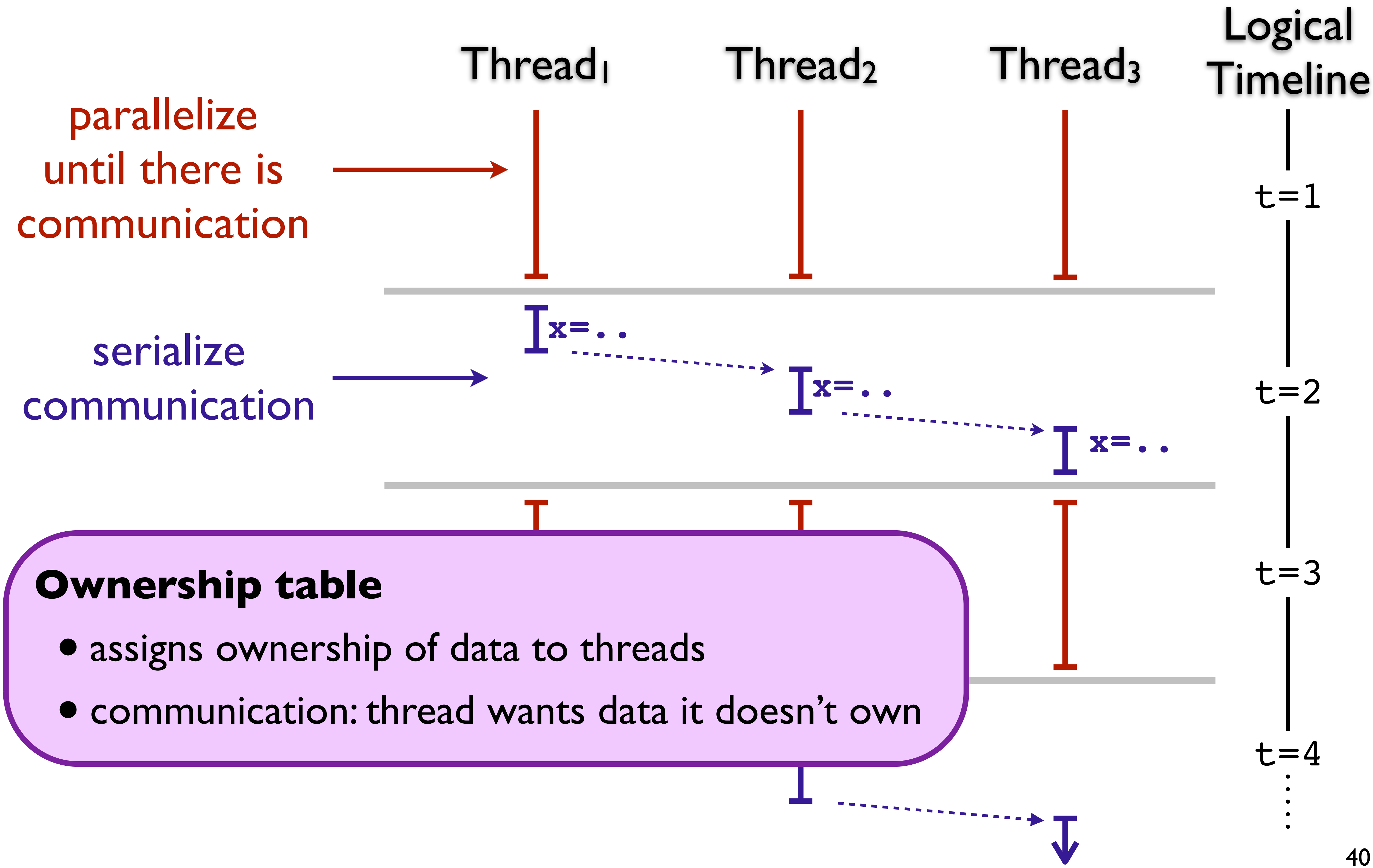
Key idea:

- *serialize* all communication deterministically

Deterministic Execution with DMP-O



Deterministic Execution with DMP-O



Ownership table

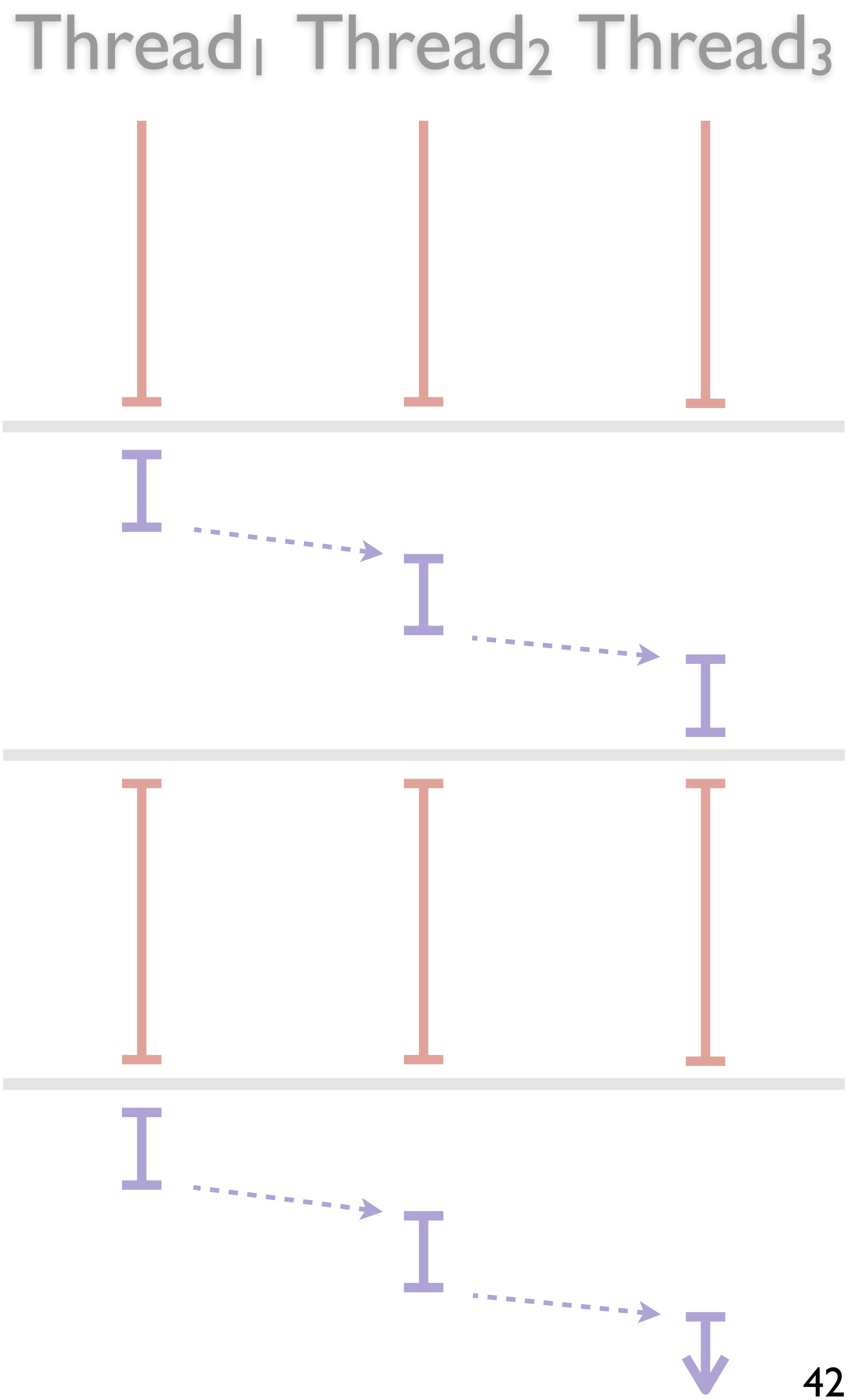
- assigns ownership of data to threads
- communication: thread wants data it doesn't own

dOS: Changes for DMP-O

Ownership Table

<u>must instrument the system interface</u>

- *loads/stores*
  - for shared-memory
- *system calls*
  - for in-kernel channels
  - *explicit*: pipes, files, signals, ...
  - *implicit*: address space, file descriptor table, ...

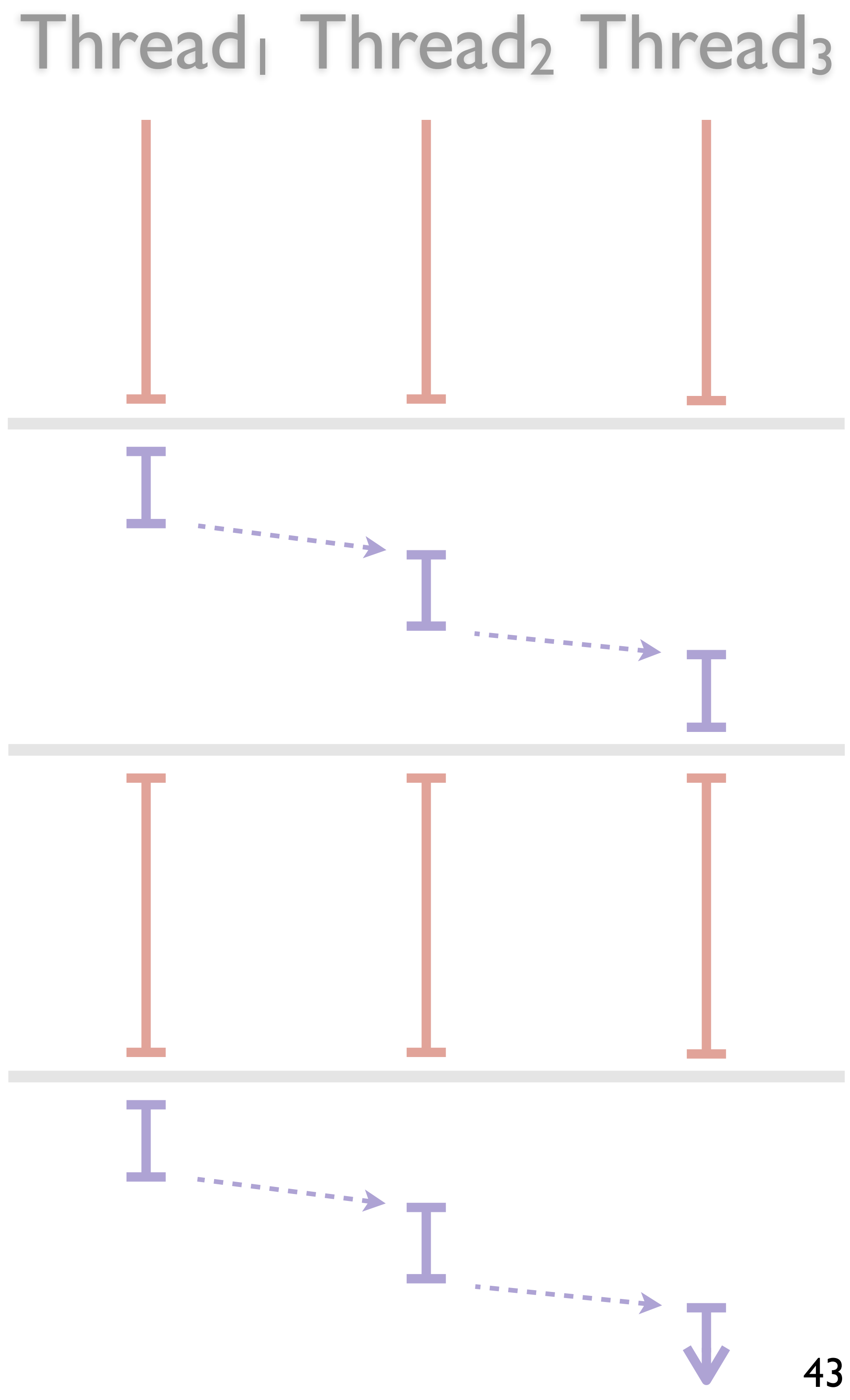


dOS: Changes for DMP-O

Ownership Table

for shared-memory

- must instrument loads/stores
  - use page-protection hw
- each thread has a *shadow page table*
  - permission bits denote ownership
  - page faults denote communication
  - *page granularity* ownership



dOS: Changes for DMP-O

Ownership Table

for in-kernel channels (pipes, etc.)

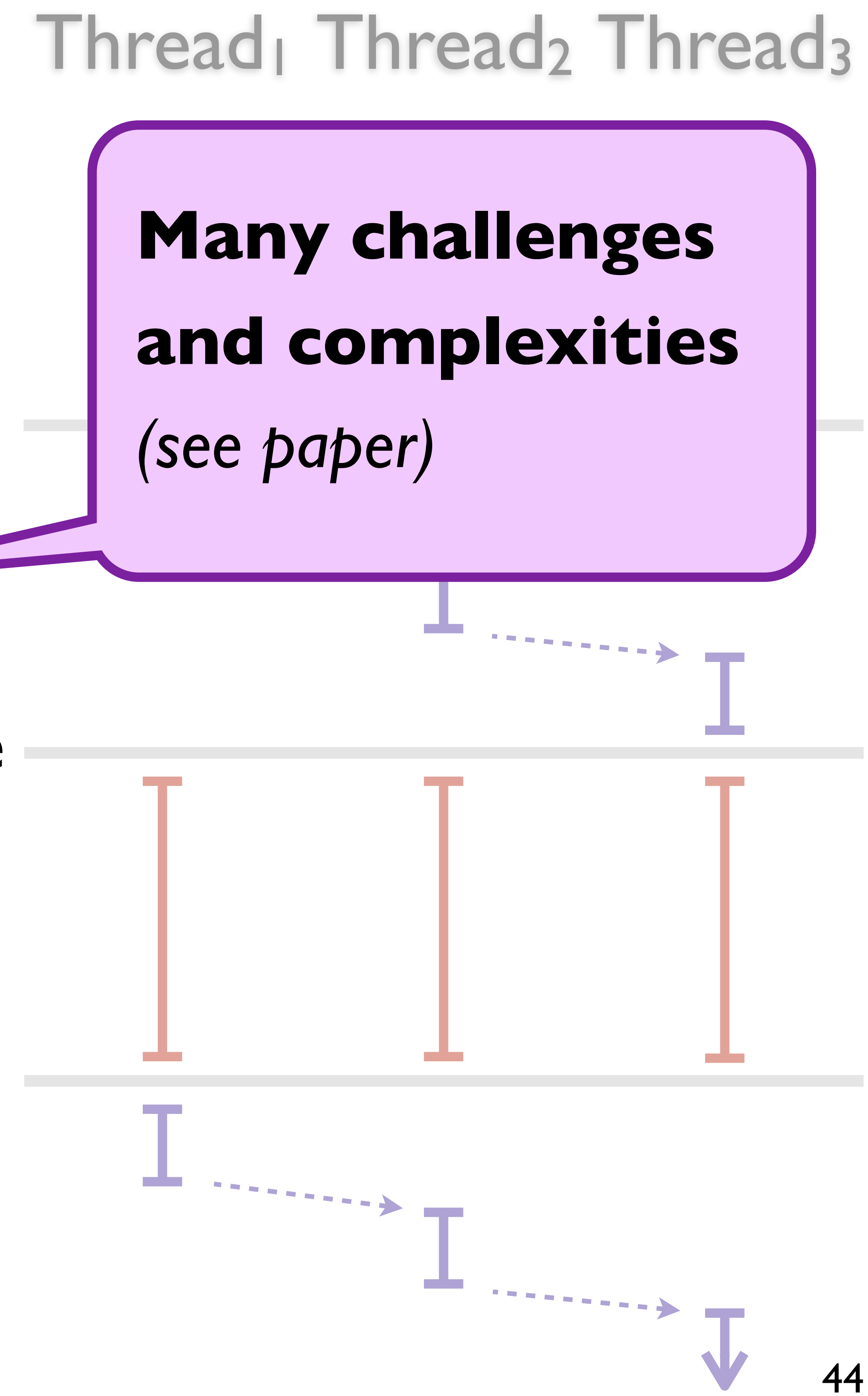
- must instrument *system calls*
- on syscall entry:
  - decide what channels are used

    `read()`: pipe or file being read

    `mmap()`: the thread's address space
  - acquire ownership

    ownership table is just a hash-table
  - any external channels?

    if yes: forward to shim program



Many challenges and complexities

(see paper)

Outline

- Example Uses
  - ➡ a parallel computation
  - ➡ a webserver
- Deterministic Process Groups
  - ➡ system interface
  - ➡ conceptual model
- dOS: our Linux-Based Implementation
- Evaluation

Evaluation Overview

Setup

- ➡ 8-core 2.8GHz Intel Xeon, 10GB RAM
- ➡ Each application ran in its own DPG

Verifying determinism

- ➡ used the racey deterministic stress test [ISCA02, MarkHill]

Key questions

- ➡ How much internal nondeterminism is eliminated? (log sizes for record/replay)
- ➡ How much overhead does dOS impose?
- ➡ How much does dOS affect parallel scalability?

Eval: Record Log Sizes

dOS

- implemented an "execution recorder" shim

SMP-ReVirt (a hypervisor) [VEE 08]

- also uses page-level ownership-tracking
- . . . but has to record *internal* nondeterminism

Log size comparison

	dOS	SMP-ReVirt	
`fmm`	1 MB	83 GB	*(log size per day)*
`lu`	11 MB	11 GB	
`ocean`	1 MB	28 GB	
`radix`	1 MB	88 GB	8,800x bigger!
`water`	5 MB	58 GB	

Eval: dOS Overheads

Possible sources of overhead

- deterministic scheduling
- shim program interposition

Ran each benchmark in three ways:

- without a DPG (ordinary, nondeterministic)

  ↕ scheduling overheads

- with a DPG only

  ↕ shim overheads

- with a DPG and an “execution recorder” shim program

Eval: dOS Overheads

Apache

- 16 worker threads
- serving 100KB static pages

  DPGs saturate 1 gigabit network

- serving 10 KB static pages

Nondet (no DPG)	*saturates 1 gigabit network*
DPG (no shim):	26% throughput drop
DPG (with record shim):	78% throughput drop (over Nondet)

Chromium

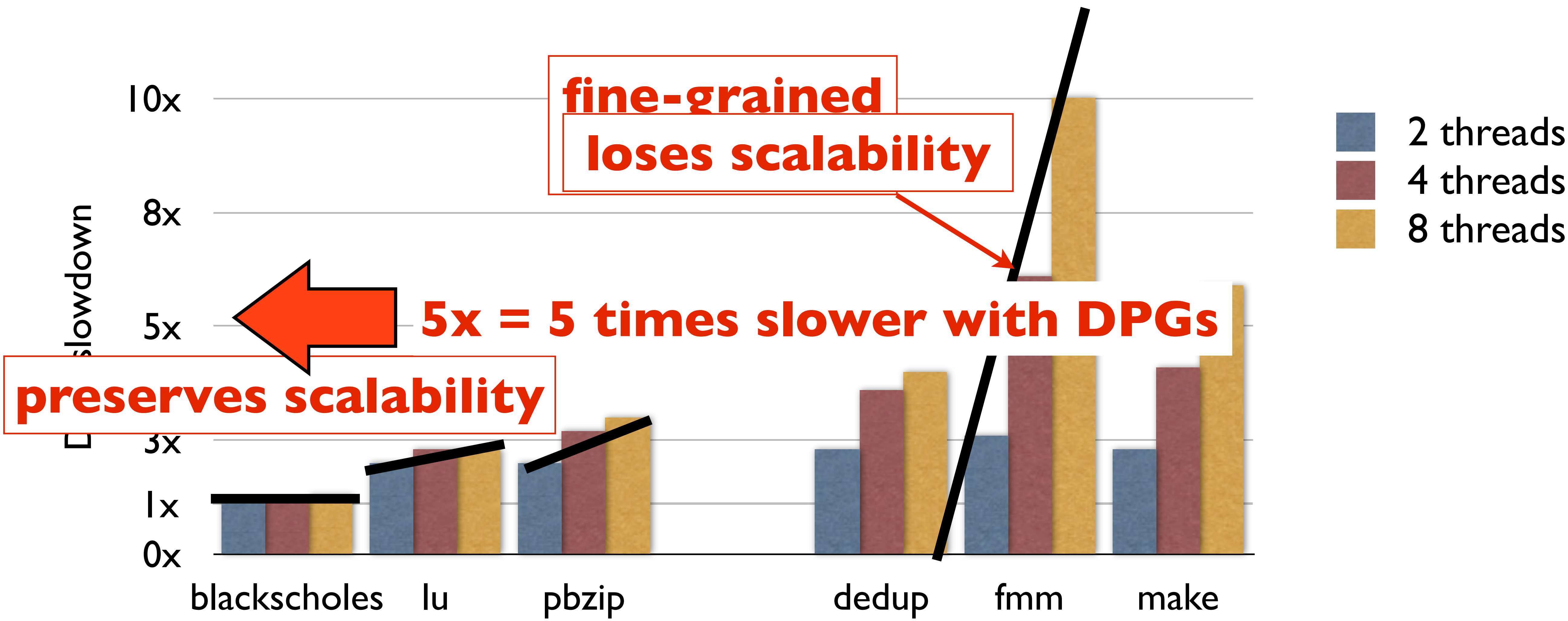
- process per tab
- scripted user session (5 tabs, 12 urls)

DPG (no shim):	1.7x slowdown
DPG (with record shim):	1.8x slowdown (over Nondet)

Eval: dOS Overheads

Parallel application slowdowns

- DPG only
- relative to nondeterministic execution



Wrap Up

Deterministic Process Groups

- new OS abstraction
- *eliminate* or *control* sources of nondeterminism

dOS

- Linux-Based implementation of DPGs
- use cases demonstrated: deterministic execution, record/replay, and replicated execution

Also in the paper . . .

- many more implementation details
- a more thorough evaluation
- thoughts on a "from scratch" implementation

Thank you!

Questions?

http://sampa.cs.washington.edu

```
C:\DOS
C:\DOS\RUN
C:\DOS\RUN\DETERM~1.EXE
```

(backup slides)

Performance?

Already good enough for some workloads!

- infrequent system calls
- infrequent fine-grained sharing
  - *examples:* Apache 100KB static pages, blackscholes, pbzip, etc.

Improvements possible:

- better scheduling algorithm (*DMP-TM, DMP-B*) [Asplos09, Asplos10]
- binary instrumentation (*to support arbitrary data granularity*)
- implement shims as kernel modules (*lower context switch overhead*)

Research question:

- how much does determinism <u>fundamentally</u> impact performance?

Overheads Breakdown

Deterministic scheduler

	% serialization	% single-stepping
Apache 100KB	26%	0%
Apache 10KB	60%	0%
Chromium	25%	13%
blackscholes	3%	27%
fmm	54%	18%
dedup	90%	12%

Shim context-switching

microbenchmark: **5x** overhead on system call traps

Why are DPGs awesome?

DPGs give you determinism, which helps:

- testing
- debugging
- fault-tolerant replication
- security
  - can eliminate internal timing channels [Aviram et al, CCSW10]

DPGs give you determinism flexibly:

- user-defined process group
  - keeps separate apps isolated in their own determinism domain
- shim programs can customize:
  - the interface to the nondeterministic external world
  - the set of deterministic services

    (more details in paper)

Internal Determinism Design Choices

A single thread
- current systems
- massively nondeterministic on multiprocessors

A single multithreaded process

DPGS →

A group of multithreaded processes
- our choice
- most flexible

A virtual machine
- too costly, too inflexible

A local area network cluster?

deterministic box

Right Place For Determinism?

Language?

- ✓ more robust determinism, enables static analysis (lower cost)
- ➡ must rewrite program with specialized constructs

Operating System?

- ✓ support arbitrary, unmodified binaries
- ➡ high overheads for some workloads

Compiler?

- ✓ lower overheads than OS for some workloads (finer-grained tracking)
- ➡ can't resolve communication via the kernel

Hardware?

- ✓ low-overhead shared-memory determinism
- ➡ must build custom hardware

SMP-ReVirt?

Advantages of SMP-ReVirt

✓ full-system record/replay

- includes OS code
- via a hypervisor implementation

Advantages of dOS

✓ process level

- cheaper than full-system?
- don't need to resolve kernel-level shared-memory (up to 50% of sharing for some benchmarks [VEE 08])

✓ no internal nondeterminism

- smaller logs (by 1,000x)

Prior Work: Record/Replay

Record internal nondeterminism

- ➡ in software [SMP-ReVirt, Scribe, DejaVu, ...]
- ➡ in hardware [FDR, DeLorean, ...]
  - big logs, high runtime overheads for software

Search execution space during replay

- ➡ record a few bits of internal nondeterminism [PRES, ODR]
- ➡ record nothing [ESD]
  - cannot guarantee replay (might fail to find an execution)

Advantages of dOS

- ✓ small logs (no internal nondeterminism)
- ✓ replay is guaranteed

Prior Work: Deterministic Execution

References

➡ DMP	[ASPLOS 09]	custom hardware
➡ Kendo	[ASPLOS 09]	custom runtime (race-free programs only)
➡ CoreDet	[ASPLOS 10]	custom compiler/runtime
➡ Grace	[OOPSLA 10]	custom runtime (fork-join programs only)

Advantages of dOS

✓ supports:
- multiple processes
- communication other than shared-memory (pipes, etc.)
- arbitrary binaries

✓ does not require:
- custom hardware
- recompilation

✓ **shims for external nondeterminism**