Memory Consistency & Synchronization

15-740 FALL'21

NATHAN BECKMANN

Today: Consistency & synchronization

Consistency

- What is it?
- Sequential consistency
- Weak consistency

Synchronization

- ISA support
- Locks
- Barriers

Memory Consistency

Review: Multiprocessor Types

Loosely coupled multiprocessors

- No shared global memory address space
- Multicomputer network
  - Network-based multiprocessors
- Usually programmed via message passing
  - Explicit calls (send, receive) for communication

Tightly coupled multiprocessors

- Shared global memory address space
- Traditional multiprocessing: symmetric multiprocessing (SMP)
  - Existing multi-core processors, multithreaded processors
- Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
  - Operations on shared data require synchronization

Difficulty in Parallel Programming

Little difficulty if parallelism is natural

- "Embarrassingly parallel" applications
- Multimedia, physical simulation, graphics
- Large web servers, databases?

Difficulty is in

- Getting parallel programs to work correctly
- Optimizing performance in the presence of bottlenecks

Much of **parallel computer architecture** is about

- Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
- Making programmer's job easier in writing correct and high-performance parallel programs

Ordering of Operations

Operations: A, B, C, D

- In what order should the hardware execute (and report the results of) these operations?

A contract between programmer and microarchitect

- Specified by the ISA

Preserving an “expected” (more accurately, “agreed upon”) order simplifies programmer’s life

- Ease of debugging; ease of state recovery, exception handling

Preserving an “expected” order usually makes the hardware designer’s life difficult

- Especially if the goal is to design a high-performance processor
- Processors want to do things out-of-order for performance!

Memory Ordering in a Single Processor

Specified by the von Neumann model

Sequential order

- Hardware executes the load and store operations in the order specified by the sequential program

Out-of-order execution does not change the semantics

- Hardware retires operations in program order

Advantages:

- Architectural state is precise within an execution.
- Architectural state is consistent across different runs of the program → Easier to debug programs

Disadvantage: Preserving order adds overhead, reduces performance, particularly with speculation

Memory Ordering in a Dataflow Processor

A memory operation executes when its operands are ready

Ordering specified only by data dependencies

Two operations can be executed and retired in any order if they have no dependency

Advantage: Lots of parallelism → high performance

Disadvantage: Order can change across runs of the same program → Very hard to debug

Memory Ordering in a MIMD Processor

Each processor's memory operations are in sequential order with respect to the "thread" running on that processor (assume each processor obeys the von Neumann model)

Multiple processors execute memory operations concurrently

How does the memory see the order of operations from all processors?

- In other words, what is the ordering of operations across different processors?

Why Does This Even Matter?

Ease of debugging

- It is nice to have the same execution done at different times have the same order of memory operations

Correctness

- Can we have incorrect execution if the order of memory operations is different from the point of view of different processors?

Performance and overhead

- Enforcing a strict “sequential ordering” can make life harder for the hardware designer in implementing performance enhancement techniques (e.g., OoO execution, caches)

Protecting Shared Data

Threads are not allowed to update shared data concurrently

- For correctness purposes

Accesses to shared data are encapsulated inside *critical sections* or protected via *synchronization constructs (locks, semaphores, condition variables)*

Only one thread can execute a critical section at a given time

- Mutual exclusion principle

A multiprocessor should provide the *correct* execution of synchronization primitives to enable the programmer to protect shared data

Sequential Consistency [Lamport, ToC'79]

A multiprocessor system is sequentially consistent if:

- The result of any execution is the same as if the operations of all the processors were executed in some sequential order

AND

- The operations of each individual processor appear in this sequence in the order specified by its program

SC: Programmer's Abstraction

Memory is a switch that services one load or store at a time from any processor

All processors see the currently serviced load or store at the same time

Each processor's operations are serviced in program order

Sequentially Consistent Operation Orders

Potential correct global orders (all are correct):

A B X Y

A X B Y

A X Y B

X A B Y

X A Y B

X Y A B

Thread 1:

```
ld X
ld Y
```

Thread 2:

```
ld A
ld B
```

Which order (interleaving) is observed depends on implementation and dynamic latencies

Implementing SC in hardware

Only a few commercial systems implemented SC

- Neither x86 nor ARM are SC

Requires either severe performance penalty

- Wait for stores to complete before issuing new store

Or, complex hardware (MIPS R10K)

- Issue loads speculatively
- Detect inconsistency with later store
- Squash speculative load

Software reorders too!

```
//Producer code
*datap = x/y;
*flagp = 1;
```

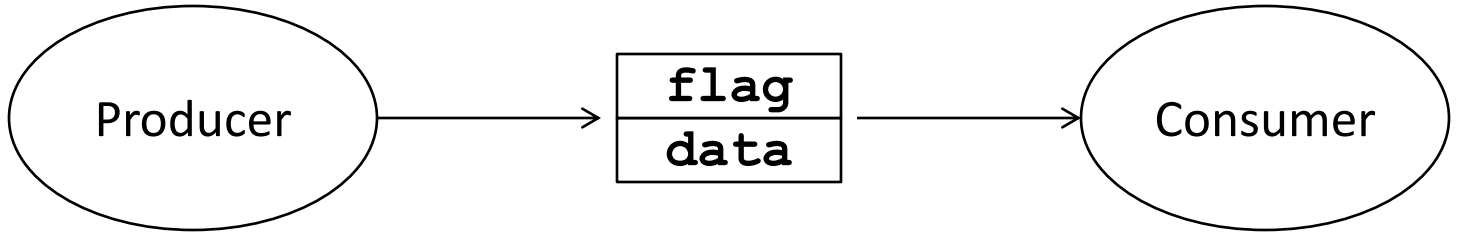
```
//Consumer code
while (!*flagp)
    ;
d = *datap;
```

Compiler can reorder/remove memory operations unless made aware of memory model

- Instruction scheduling, move loads before stores if to different address
- Register allocation, cache load value in register, don't check memory

Prohibiting these optimizations would result in very poor performance

Simple Producer-Consumer Example



Initially **flag=0**

```
st xdata, (xdatap)
li xflag, #1
st xflag, (xflagp)
```

```
spin: ld xflag, (xflagp)
      beqz xflag, spin
      ld xdata, (xdatap)
```

Is this correct?

Can consumer read **flag=1** before **data** written by producer?

Simple Producer-Consumer Example

Initially **flag =0**

```
st xdata, (xdatap)
li xflag, #1
st xflag, (xflagp)
```

```
spin: ld xflag, (xflagp)
      beqz xflag, spin
      ld xdata, (xdatap)
```

Dependencies from sequential ISA

Dependencies added by sequentially consistent memory model

Consequences of Sequential Consistency

Corollaries

1. Within the same execution, all processors see the same global order of operations to memory

   → No correctness issue

   → Satisfies the “happened before” intuition

2. Across different executions, different global orders can be observed (each of which is sequentially consistent)

   → Debugging is still difficult (as order changes across runs)

Issues with Sequential Consistency?

Nice abstraction for programming, but two issues:

- Too conservative
- Limits the aggressiveness of performance enhancement techniques

Is the total global order requirement too strong?

- Do we need a global order across all operations and all processors?
- How about a global order only across all stores?
  - Total store order memory model; unique store order model
- How about a enforcing a global order only at the boundaries of synchronization?
  - Relaxed memory models
  - Acquire-release consistency model

Weaker Memory Consistency

The ordering of operations is important when the order affects operations on shared data → i.e., when processors need to synchronize to execute a “program region”

Weak consistency

- Idea: Programmer specifies regions in which memory operations do not need to be ordered
- “Memory fence” instructions delineate those regions
  - All memory operations before a fence must complete before the fence is executed
  - All memory operations after the fence must wait for the fence to complete
  - Fences complete in program order
- All synchronization operations act like a fence

Fences in producer-consumer

Initially **flag =0**

```
sd xdata, (xdatap)
li xflag, 1
fence.w.w     // Write-Write
              // fence
sd xflag, (xflagp)
```

```
spin: ld xflag, (xflagp)
      beqz xflag, spin
      fence.r.r     // Read-Read
                    // fence
      ld xdata, (xdatap)
```

Tradeoffs: Weaker Consistency

Advantage

- No need to guarantee a very strict order of memory operations
  → Simplifies high-performance hardware implementation
  → Can be higher performance than stricter ordering

Disadvantage

- More burden on the programmer or software (need to get the “fences” correct)

Another example of the programmer-microarchitect tradeoff

Processor Consistency & Total Store Order

Approximately what's implemented in x86

Sequential consistency respects all memory orderings:

- W → W: Writes finish before later writes
- W → R: Writes finish before later reads
- R → W: Reads finish before later writes
- R → R: Reads finish before later reads

PC & TSO scrap the W→R ordering. Why?

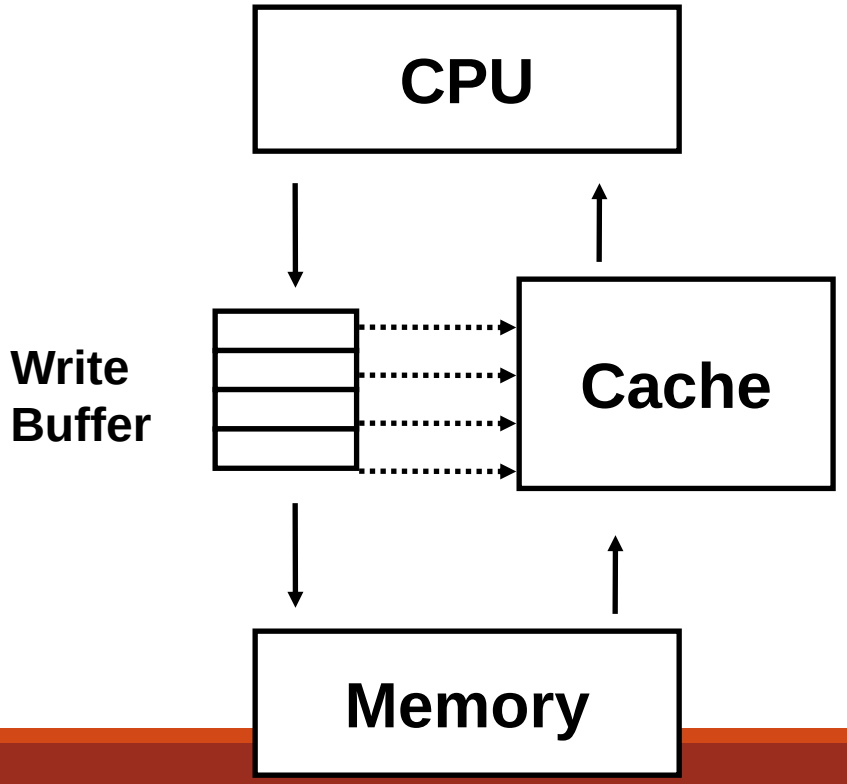
- Move writes off the critical path

TSO (not PC) enforces that all processors will observe a single global order of writes

Write buffering

Write Buffer

- Common optimization for all caches
- Overlaps memory updates with processor execution
- Read operation must check write buffer for matching address



SC vs PC vs TSO examples

Thread 1:	Thread 2:
A = 1	while (flag == 0)
flag = 1	print A

SC == PC == TSO

Thread 1:	Thread 2:
A = 1	print B
B = 1	print A

SC == PC == TSO

Thread 1:	Thread 2:	Thread 3:
A = 1	while (A == 0)	while (B == 0)
	B = 1	print A

SC == TSO != PC

Thread 1:	Thread 2:
A = 1	B = 1
print B	print A

SC != TSO == PC

Data-race-free & relaxed atomics

Data-race-free-0 (DRF0) [ISCA'90]

- Must label all racing memory accesses as "synch" / "atomics" in C++
- All atomics order data accesses
- All atomics order atomics
- ➔ Ensures SC semantics if no data races

Data-race-free-1 (DRF1) [TPDS'93]

- "Unpaired atomics" do not order data accesses
- Atomics *do* order other atomics
- ➔ Ensures SC semsntics if no data races

"Relaxed atomics" [PLDI'08]

- Do not order data accesses or atomics
- But has no formal spec, can produce "values from nowhere" that violate SC

Chasing away RAts. ISCA'19
Matt Sinclair, Jonathan Alsop, Sarita Adve.

Data-race-free & relaxed atomics

"Chasing Away RAts" paper analyzes how relaxed atomics are used in real programs & provides semantics

E.g., commutative updates, like updating an event counter

- *Commutative:* Order that atomics happen doesn't matter (unpaired atomics too strong)
- *Intermediates not visible* to threads
- ➔ Final result will match SC semantics (but not necessarily intermediate states)

Several other categories

Synchronization

Review: Caveats of Parallelism

Amdahl's Law

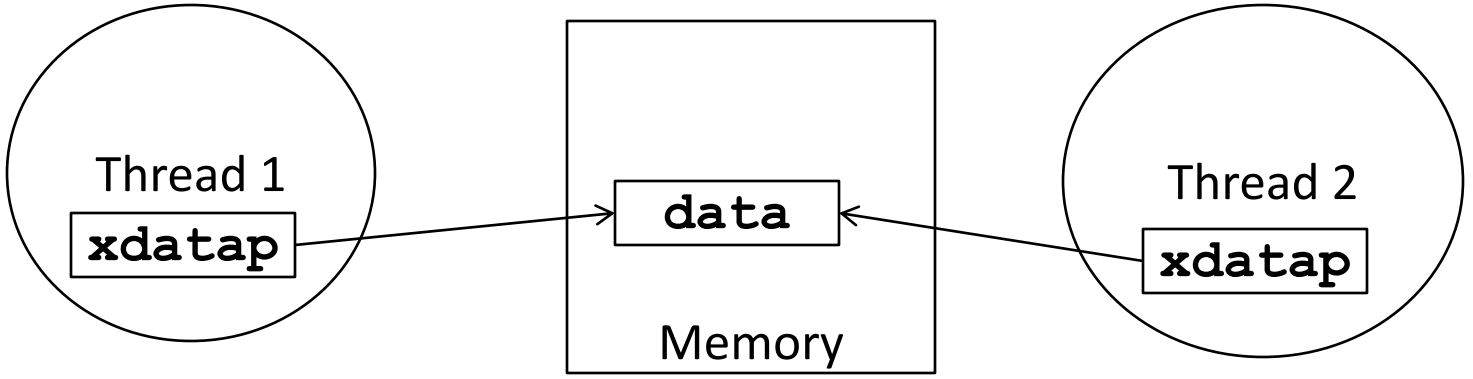
- f: Parallelizable fraction of a program
- N: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

Parallel portion is usually not perfectly parallel

- **Synchronization overhead (e.g., updates to shared data)**
- Load imbalance overhead (imperfect parallelization)
- Resource sharing overhead (contention among N processors)

Simple mutual-exclusion example




```
// Both threads execute:
ld xdata, (xdatap)
add xdata, 1
st xdata, (xdatap)
```

Is this correct? *No! Both threads can enter crit sec*

MutEx with LD/ST in SC

A protocol based on two shared variables c1 and c2.

Initially, both c1 and c2 are 0 (not busy)

Process 1

```
    ...
    c1=1;
L:  if c2=1 then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
L:  if c1=1 then go to L
      < critical section>
    c2=0;
```

What is wrong?

Deadlock!

MutEx with LD/ST in SC (2nd attempt)

To avoid *deadlock*, let a process give up the reservation

(i.e. Process 1 sets c1 to 0) while waiting.

Process 1

```
    ...
L:  c1=1;
    if c2=1 then
    { c1=0; go to L}
      < critical section>
    c1=0
```

Process 2

```
    ...
L:  c2=1;
    if c1=1 then
    { c2=0; go to L}
      < critical section>
    c2=0
```

1. Deadlock impossible, but *livelock* may occur (low probability)

2. Unlucky processes never get lock (*starvation*)

A Protocol for Mutual Exclusion (+ SC)

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and **turn**.

Initially, both c1 and c2 are 0 (not busy)

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
            then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
        then go to L
      < critical section>
    c2=0;
```

turn = i ensures that only process i can wait

Variables c1 and c2 ensure mutual exclusion

Solution for n processes was given by Dijkstra and is quite tricky!

Components of Mutual Exclusion

Acquire

- How to get into critical section

Wait algorithm

- What to do if acquire fails

Release algorithm

- How to let next thread into critical section

Can be implemented using LD/ST, but…

- Need fences in weaker models
- Doesn't scale + **complex**

Busy Waiting vs. Blocking

Threads spin in above algorithm if acquire fails

Busy-waiting is preferable when:

- Scheduling overhead is larger than expected wait time
- Schedule-based blocking is inappropriate (eg, OS)

Blocking is preferable when:

- Long wait time & other useful work to be done
- Especially if core is needed to release the lock!

Hybrid *spin-then-block* often used

Need atomic primitive!

Many choices…

Test&Set – set to 1 and return old value

Swap – atomic swap of register + memory location

Fetch&Op

- E.g., Fetch&Increment, Fetch&Add, …

Compare&Swap – “if *mem == A then *mem == B”

Load-linked/Store-Conditional (LL/SC)

Mutual Exclusion with Atomic Swap

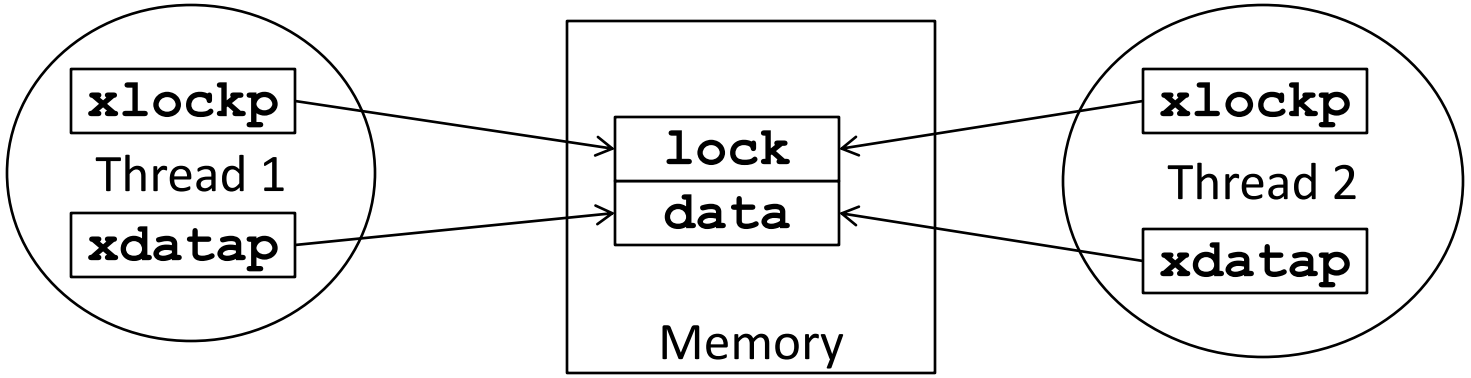
Atomic swap: `amoswap x, y, (z)`

- Semantics:
  ```
  x = Mem[z]
  Mem[z] = y
  ```

```
lock:       li r1, #1
spin:       amoswap r2, r1, (lockaddr)
            bnez r2, spin
            ret

unlock:     st (lockaddr), #0
            ret
```

Much simpler than LD/ST with SC!

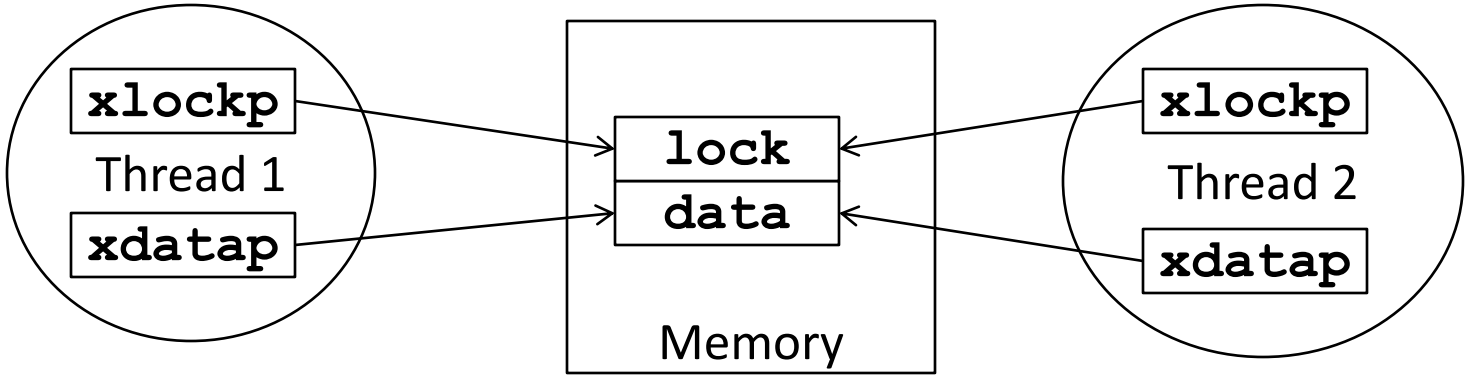
Mutual Exclusion with Atomic Swap




```
    li xone, 1
spin: amoswap xlock, xone, (xlockp)   # Acquire Lock
      bnez xlock, spin
      ld xdata, (xdatap)              # Critical Section
      add xdata, 1
      st xdata, (xdatap)
      st x0, (xlockp)                 # Release Lock
```

Assumes SC memory model

Mutual Exclusion with Relaxed Consistency




```
        li xone, 1
spin:   amoswap xlock, xone, (xlockp)
        bnez xlock, spin
        fence.r.r
        ld xdata, (xdatap)
        add xdata, 1
        sd xdata, (xdatap)
        fence.w.w
        sd x0, (xlockp)
```

Acquire Lock

Critical Section

Release Lock

Mutual Exclusion with Test & Set

Test & set: `t&s y, (x)`

- Semantics:

```
y = Mem[x]
If y == 0 then Mem[x] = 1
```

```
lock:           t&s   r1, (lockaddr)
                bnez  r1, lock
                ret

unlock:         st    (lockaddr), #0
                ret
```

Load-linked / store-conditional

Load-linked/Store-Conditional (LL/SC)

- `LL y, (x):`

```
y = Mem[x]
```

- `SC y, z, (x):`

```
if (x is unchanged since LL) then
    Mem[x] = y
    z = 1
else
    z = 0
endif
```

Useful to efficiently implement many atomic primitives

Fits nicely in 2-source reg, 1-destination reg instruction formats

Typically implemented as *weak* LL/SC: intervening loads/stores result in SC failure

Mutual Exclusion with LL/SC

```
lock:           ll r1, (lockaddr)
                bnez r1, lock
                add r1, r1, #1
                sc r1, r2, (lockaddr)
                beqz r2, lock
                ret

unlock:         st (lockaddr), #0
                ret
```

Implementing fetch&op with LL/SC

```
f&op:   ll    r1, (location)
        op    r2, r1, value
        sc    r2, r3, (location)
        beqz  r3, f&op
        ret
```

Implementing Atomics

Lock cache line or entire cache:

1. Get exclusive permissions
2. Don’t respond to invalidates
3. Perform operation (e.g., add in fetch&add)
4. Resume normal operation

Implementing LL/SC

Invalidation-based directory protocol

- SC requests exclusive permissions
- If requestor is still sharer & no incoming invalidations ➔ Success
  - **Snoopy:** Check signals on bus
  - **Directory:** Check if still a sharer at the directory
- SC does **not** send invalidations when it fails. Why? Avoid livelock

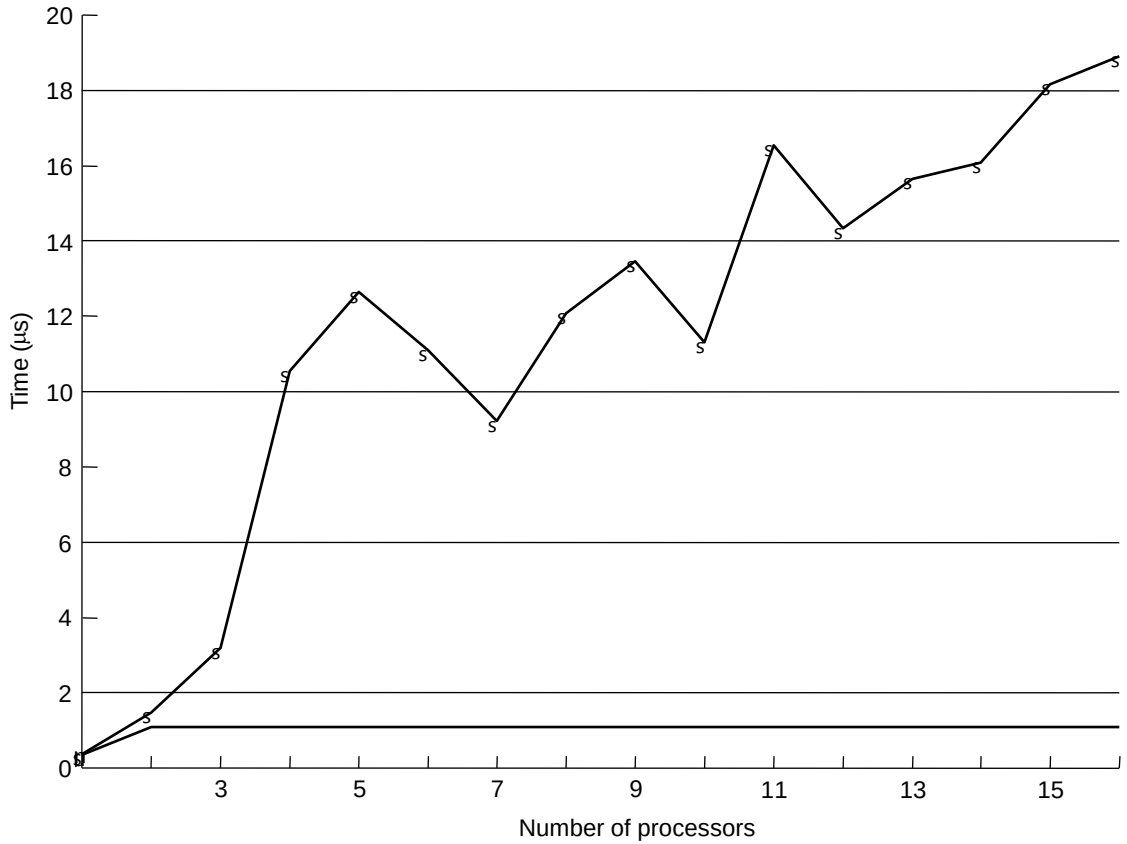
Add *link register* to store address of LL

- Invalidated upon coherence / eviction
- Only safe to use register-register instructions between LL/SC

T&S Lock Performance

Code: `for (i=0;i<N;i++) { lock; delay(c); unlock; }`

Same total no. of lock calls as P increases; measure time per transfer



WHY?

Test and Test&Set

```
A:    while (lock != 0);

      if (test&set(lock) == 0) {

          /* critical section */;
          lock = 0;

      } else {
          goto A;
      }
```

+ Spinning happens *in cache*

– Bursts of traffic when lock released

Test&Set with Backoff

Upon failure, delay for a while before retrying

- either constant delay or exponential backoff

Tradeoffs:

(+) much less network traffic

(-) exponential backoff can cause starvation for high-contention locks

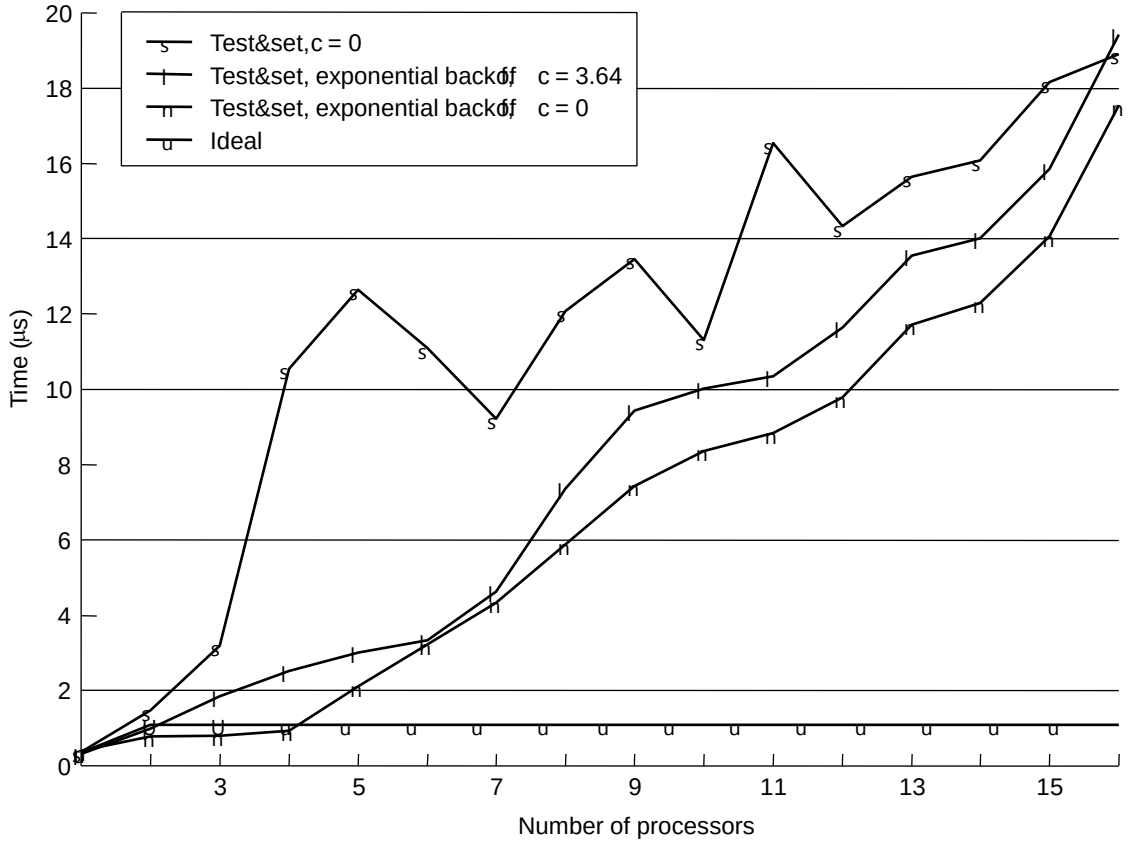
- new requestors back off for shorter times

But exponential found to work best in practice

T&S Lock Performance

Code: `for (i=0;i<N;i++) { lock; delay(c); unlock; }`

Same total no. of lock calls as P increases; measure time per transfer



Test&Set: Lingering issues

Test&Set sends updates to processors that cache the lock

Tradeoffs:

(+) good for bus-based machines

(-) still lots of traffic on distributed networks

Main problem with test&set-based schemes:

- a lock release causes all waiters to try to get the lock, using a test&set to try to get it.

Ticket Lock (fetch&incr based)

Two counters:

- `next_ticket` (number of requests)
- `now_serving` (number of releases that have happened)

Algorithm:

`ticket = fetch&increment(next_ticket)` `while (ticket != now_serving) delay(x)`	Acquire Lock
`/* mutex */`	Critical Section
`now_serving++`	Release Lock

What delay to use?

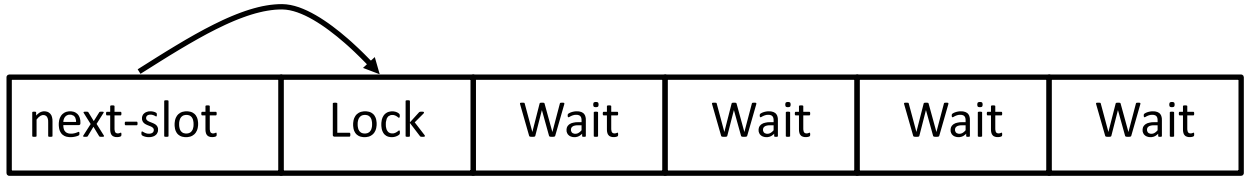
Not exponential! Why?

Instead: `ticket – now_serving`

+ Guaranteed FIFO order ➔ no starvation
+ Latency can be low (f&i cacheable)
+ Traffic can be low, but…
– Polling ➔ no guarantee of low traffic

Array-Based Queueing Locks

Every process spins on a unique location, rather than on a single `now_serving` counter




```
my-slot = F&I(next-slot)
my-slot = my-slot % num_procs
while (slots[my-slot] == 0);
slots[my-slot] = 0;
```
Acquire Lock

```
// mutex
```
Critical Section

```
slots[(my-slot+1)%num_procs] = 1;
```
Release Lock

List-Base Queueing Locks (MCS)

All other good things + O(1) traffic even without coherent caches (spin locally)

Uses compare&swap to build linked lists in software

Locally-allocated flag per list node to spin on

Can work with fetch&store, but loses FIFO guarantee

Tradeoffs:

(+) less storage than array-based locks

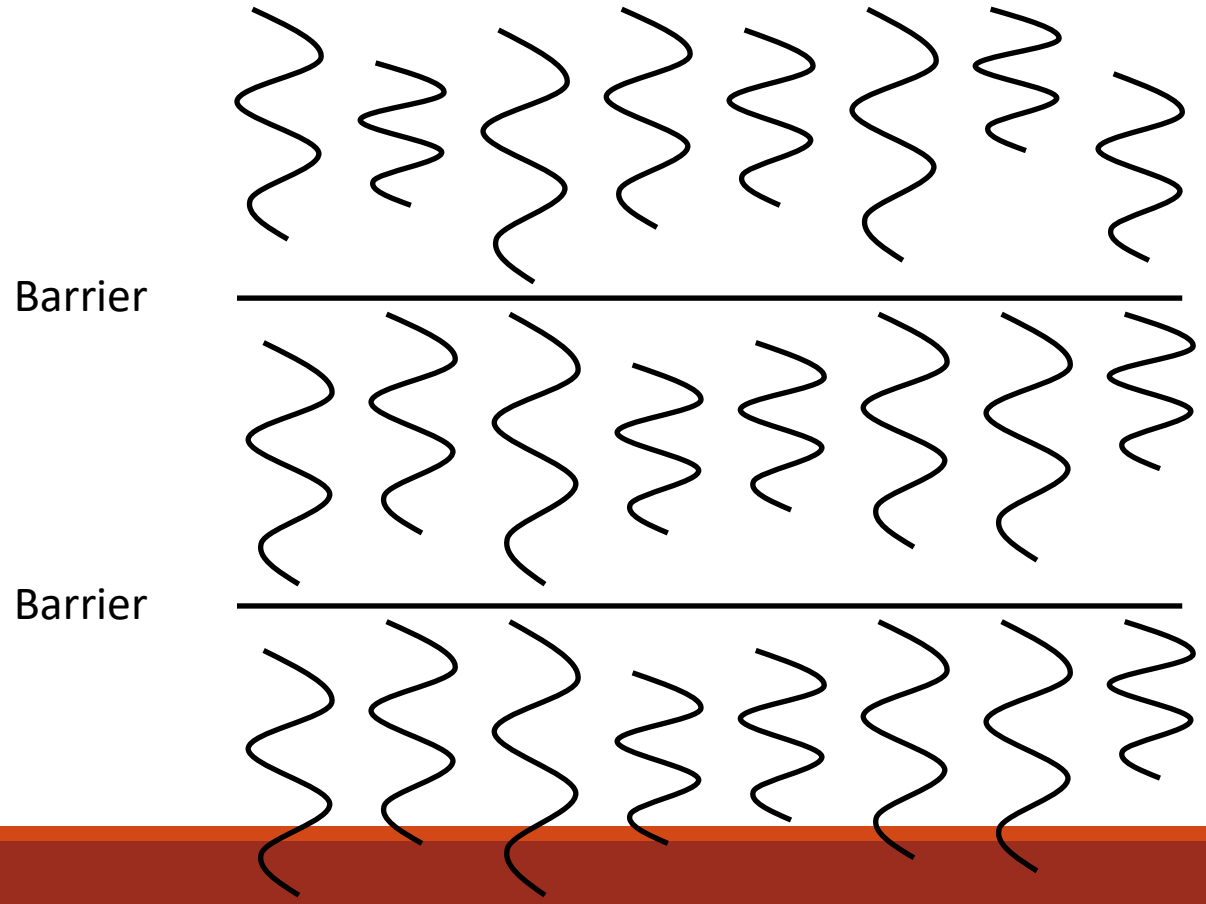
(+) O(1) traffic even without coherent caches

(-) compare&swap not easy to implement (three read-register operands)

Barriers

Barrier

Single operation: wait until P threads all reach synchronization point



Barriers

We will discuss five barriers:

- centralized
- software combining tree
- dissemination barrier
- tournament barrier
- MCS tree-based barrier

Barrier Criteria

Length of critical path

- Determines performance on scalable network

Total network communication

- Determines performance on non-scalable network (e.g., bus)

Storage requirements

Implementation requirements (e.g., atomic ops)

Critical Path Length

Analysis assumes independent parallel network paths available

May not apply in some systems

- Eg, communication serializes on bus
- In this case, total communication dominates critical path

More generally, network contention can lengthen critical path

Centralized Barrier

Basic idea:

- Notify a single shared counter when you arrive
- Poll that shared location until all have arrived
- Implemented using atomic fetch & op on counter

Centralized Barrier – 1st attempt

```
int counter = 1;

void barrier(P) {
  if (fetch_and_increment(&counter) == P) {
    counter = 1;
  } else {
    while (counter != 1) { /* spin */ }
  }
}
```

Is this implementation correct?

No! What one thread sleeps until another thread hits next barrier?

Centralized Barrier

Basic idea:

- Notify a single shared counter when you arrive
- Poll that shared location until all have arrived
- Implemented using atomic fetch & decrement on counter

Simple solution requires polling/spinning **twice**:

- First to ensure that all procs have left previous barrier
- Second to ensure that all procs have arrived at current barrier

Centralized Barrier – 2nd attempt

```
int enter = 1; // allocate on diff cache lines
int exit = 1;
void barrier(P) {
  if (fetch_and_increment(&enter) == P) {  // enter barrier
    enter = 1;
  } else {
    while (enter != 1) { /* spin */ }
  }
  if (fetch_and_increment(&exit) == P) {  // exit barrier
    exit = 1;
  } else {
    while (exit != 1) { /* spin */ }
  }
}
```

Do we need to count to P twice?

Centralized Barrier

Basic idea:

- Notify a single shared counter when you arrive
- Poll that shared location until all have arrived
- Implemented using atomic fetch & decrement on counter

Simple solution requires polling/spinning twice:

- First to ensure that all procs have left previous barrier
- Second to ensure that all procs have arrived at current barrier

Avoid spinning with sense reversal

Centralized Barrier – Final version

```
int counter = 1;
bool sense = false;
void barrier(P) {
  bool local_sense = ! sense;
  if (fetch_and_increment(&counter) == P) {
    counter = 1;
    sense = local_sense;
  } else {
    while (sense != local_sense) { /* spin */ }
  }
}
```

Centralized Barrier Analysis

Remote spinning ☹ on single shared location

$O(P)$ operations on critical path

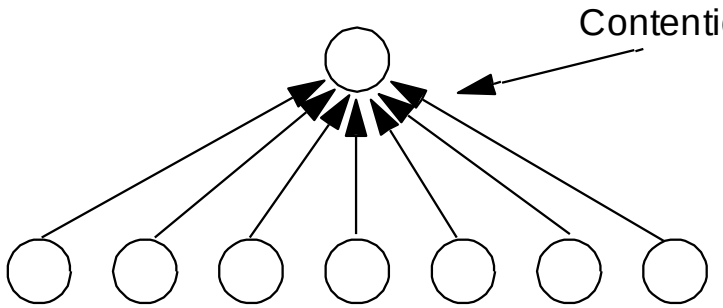
$O(1)$ space

But $O(P)$ best-case traffic, but $O(P^2)$ or even unbounded in practice *(why?)*

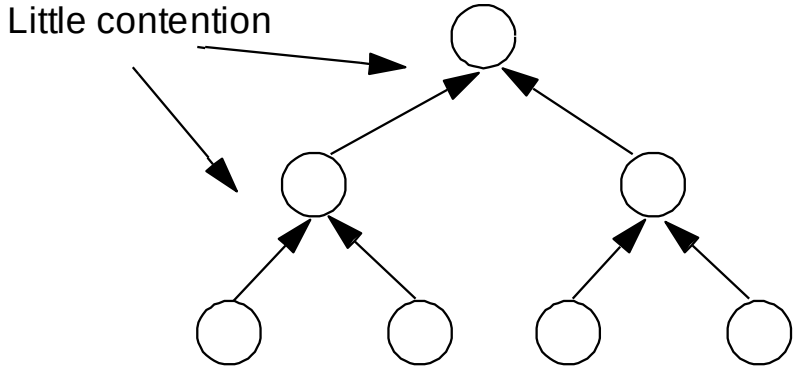
- ➔ $O(P^2)$: Atomic fetch&increment
- ➔ Unbounded: Non-coherent systems

How about exponential backoff? Bad idea! Exacerbates straggler problem.

Software Combining-Tree Barrier



Flat



Tree structured

Writes into one tree for barrier arrival

Reads from another tree to allow procs to continue

Sense reversal to distinguish consecutive barriers

Combining Barrier – Why binary?

With branching factor $\boldsymbol{k}$ what is critical path?

Depth of barrier tree is $\log_{\boldsymbol{k}} \boldsymbol{P}$

Each barrier notifies $\boldsymbol{k}$ children

➔ Critical path is $\boldsymbol{k} \log_{\boldsymbol{k}} \boldsymbol{P}$

Optimizing: $\min_{\boldsymbol{k}} \boldsymbol{k} \log_{\boldsymbol{k}} \boldsymbol{P}$ ➔ Choose $\boldsymbol{k} = 2$

Software Combining-Tree Analysis

Remote spinning ☹

$O(\log P)$ critical path

$O(P)$ space

$O(P)$ total network communication

- Unbounded without coherence

Needs atomic fetch & increment

Dissemination Barrier

$\log P$ rounds of synchronization

In round k, proc i synchronizes with proc $(i + 2^k) \bmod P$

Threads signal each other by writing flags

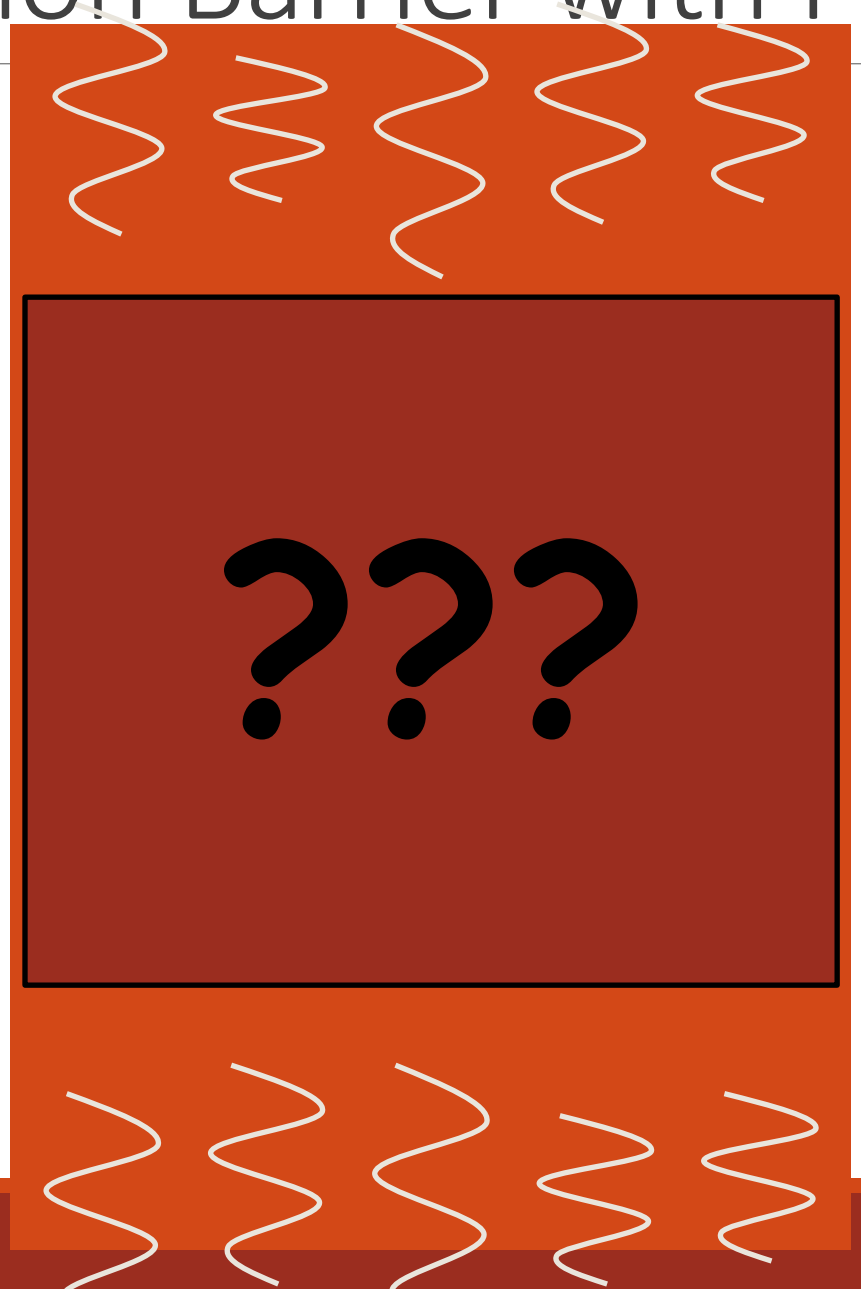
- One flag per round ➔ $\log P$ flags per thread

Advantage:

- Can statically allocate flags to avoid remote spinning
- Exactly $\log P$ critical path

Dissemination Barrier with P=5

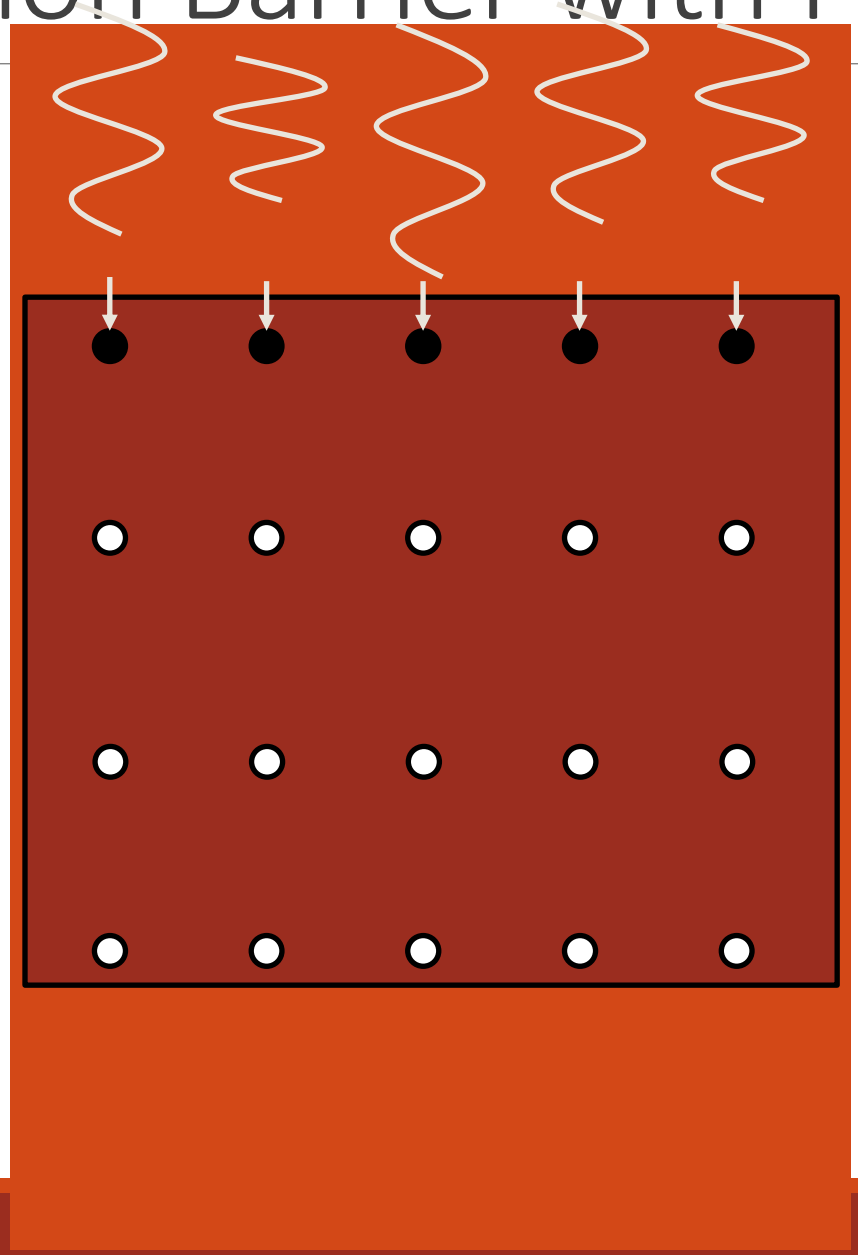
Barrier



Dissemination Barrier with P=5

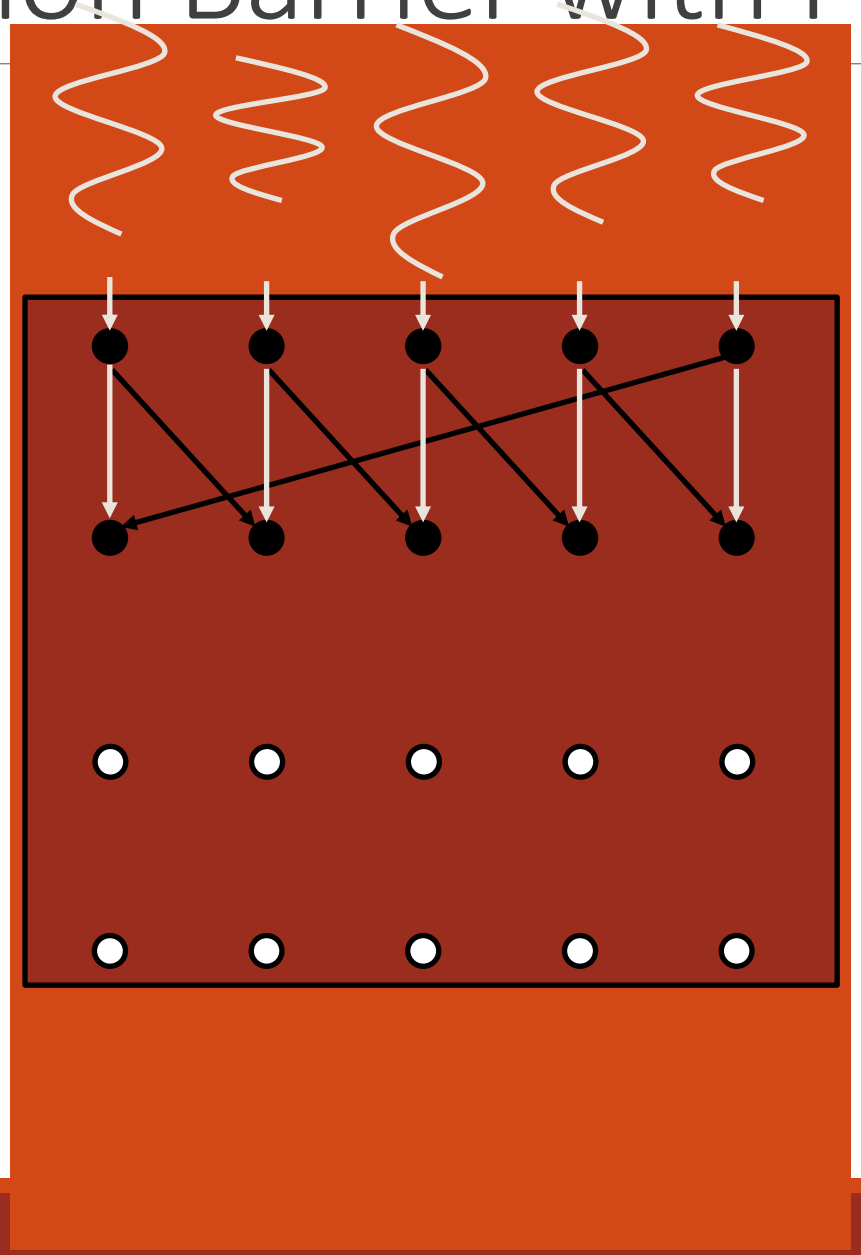
$3 = \log_2 P$ rounds

Barrier



Dissemination Barrier with P=5

$3 = \log_2 P$ rounds

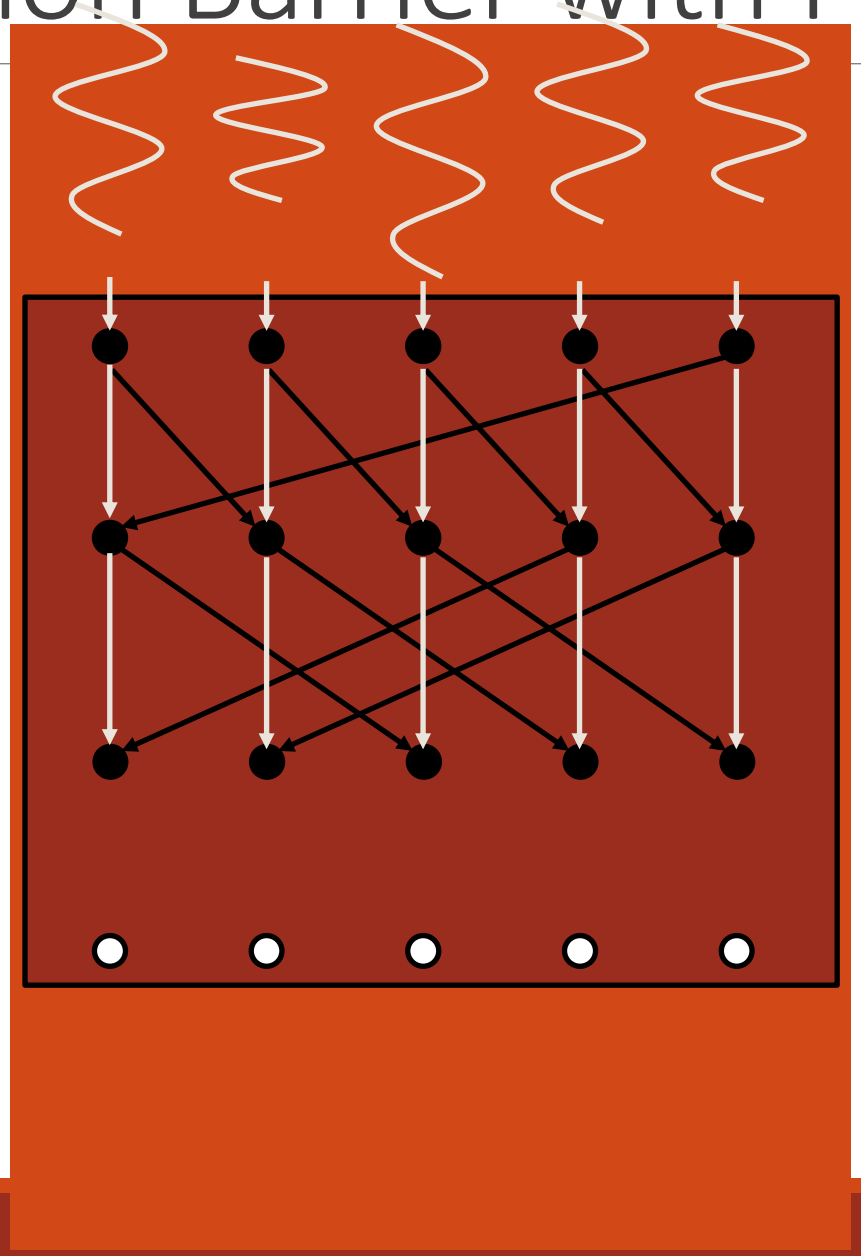


Round 1: offset $2^0 = 1$

Barrier

Dissemination Barrier with P=5

$3 = \log_2 P$ rounds



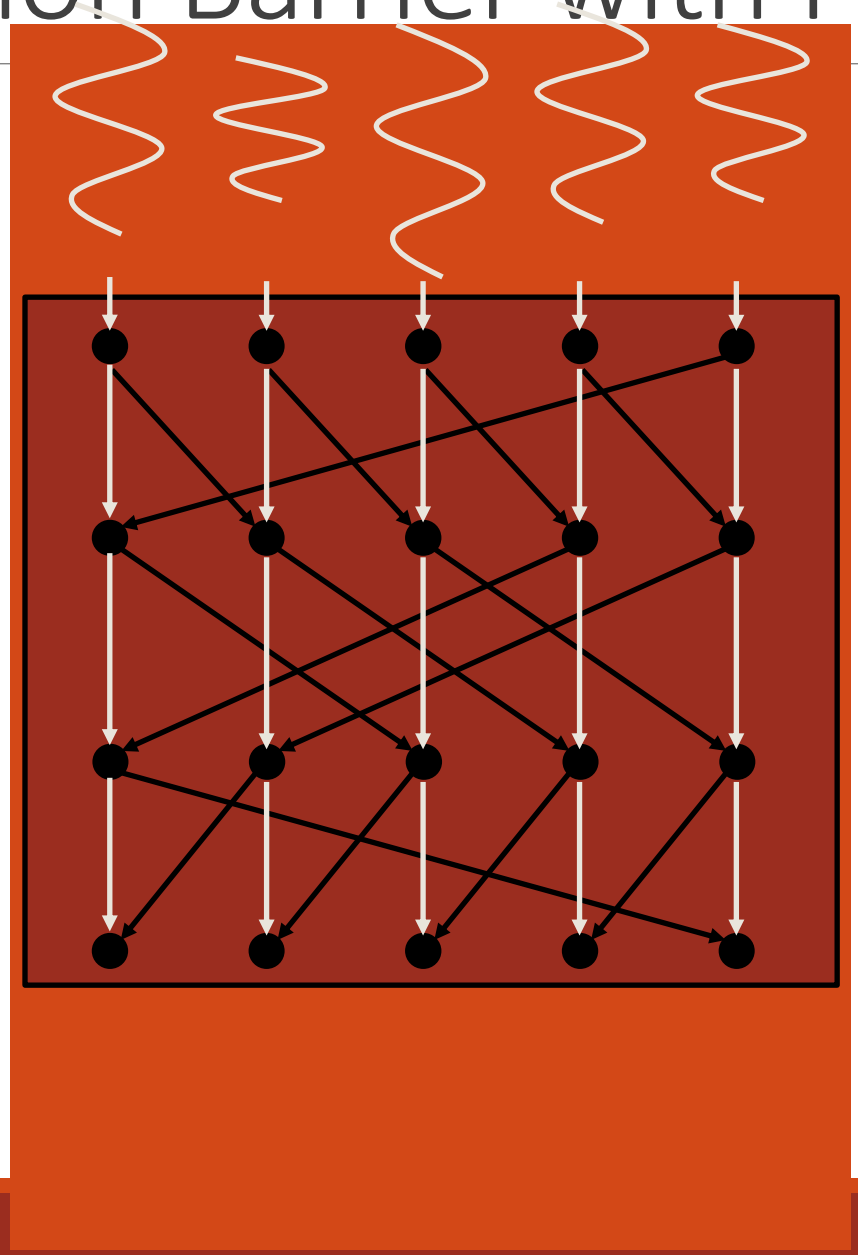
Barrier

Round 1: offset $2^0 = 1$

Round 2: offset $2^1 = 2$

Dissemination Barrier with P=5

$3 = \log_2 P$ rounds



Barrier

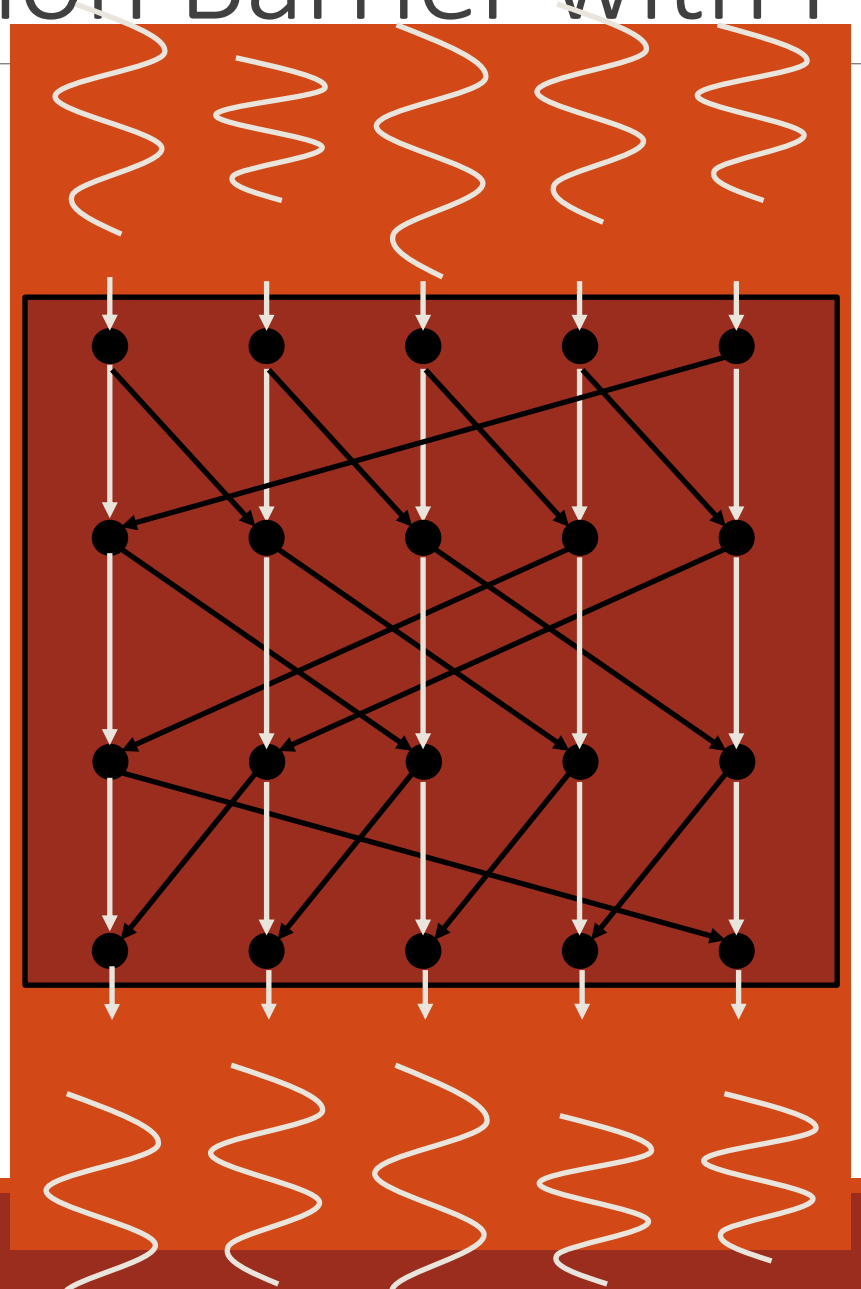
Round 1: offset $2^0 = 1$

Round 2: offset $2^1 = 2$

Round 3: offset $2^2 = 4$

Dissemination Barrier with P=5

$3 = \log_2 P$ rounds



Barrier

Round 1: offset $2^0 = 1$

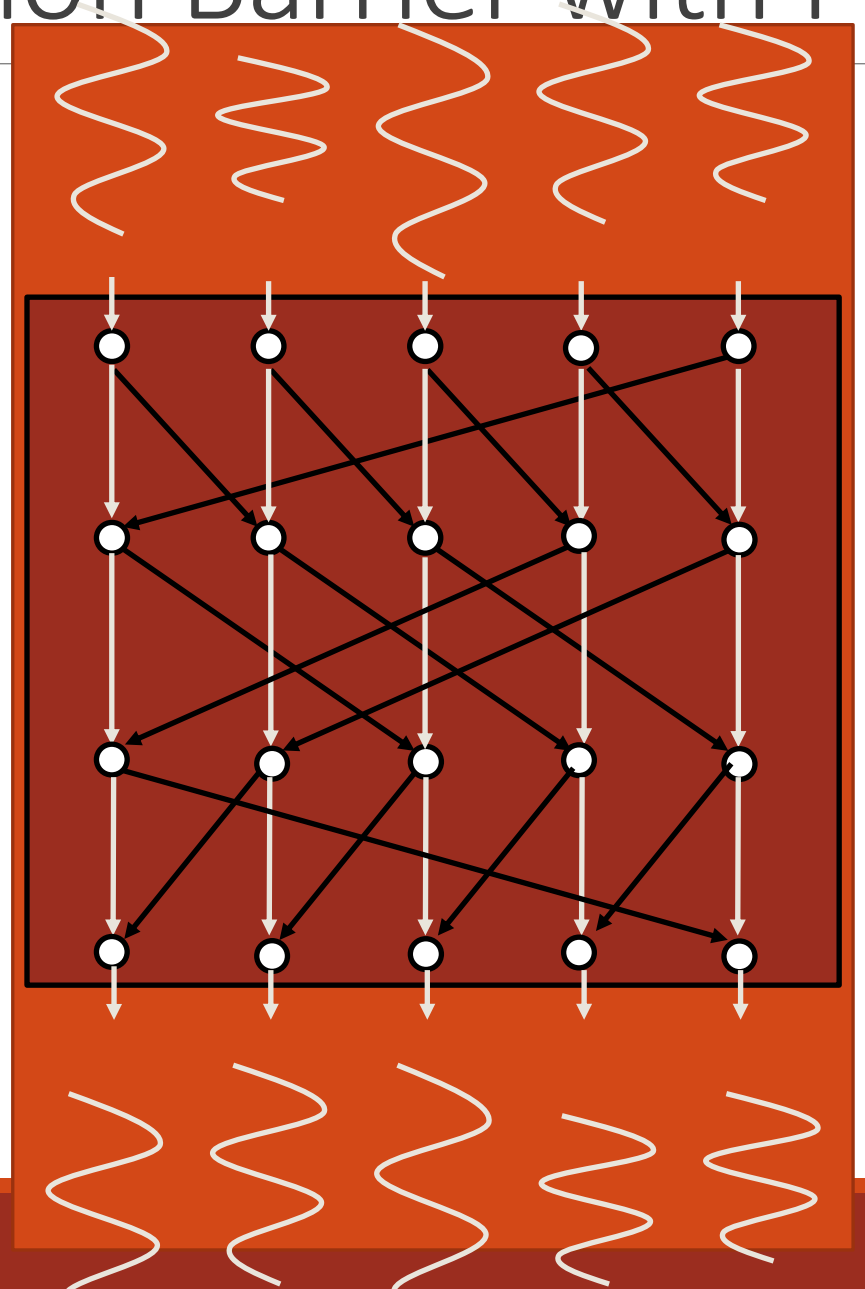
Round 2: offset $2^1 = 2$

Round 3: offset $2^2 = 4$

Dissemination Barrier with P=5

Threads can progress unevenly through barrier

But none will exit until all arrive



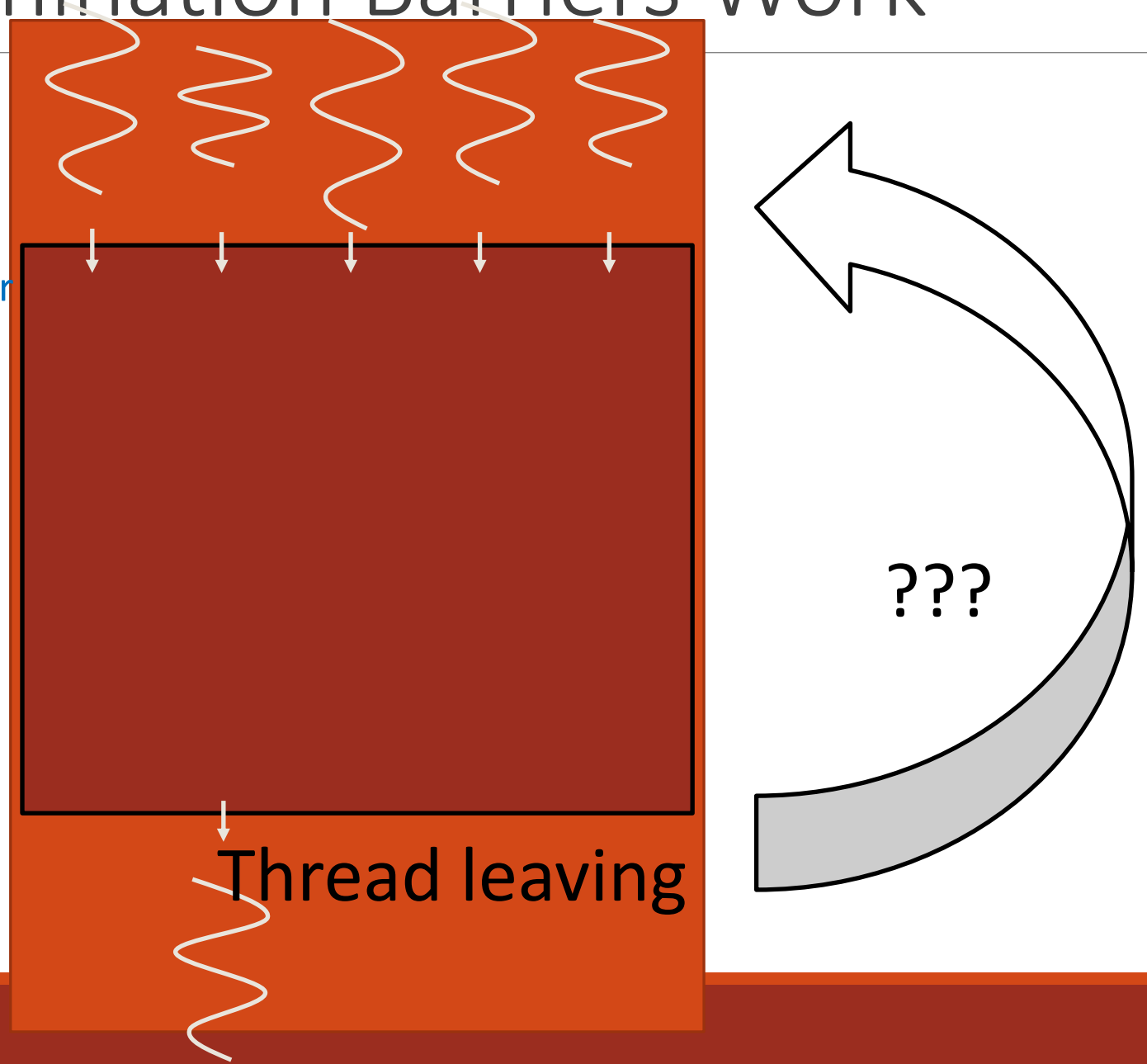
Why Dissemination Barriers Work

Prove that:

Any thread leaves barrier

➔

All threads entered barrier



Why Dissemination Barriers Work

Prove that:

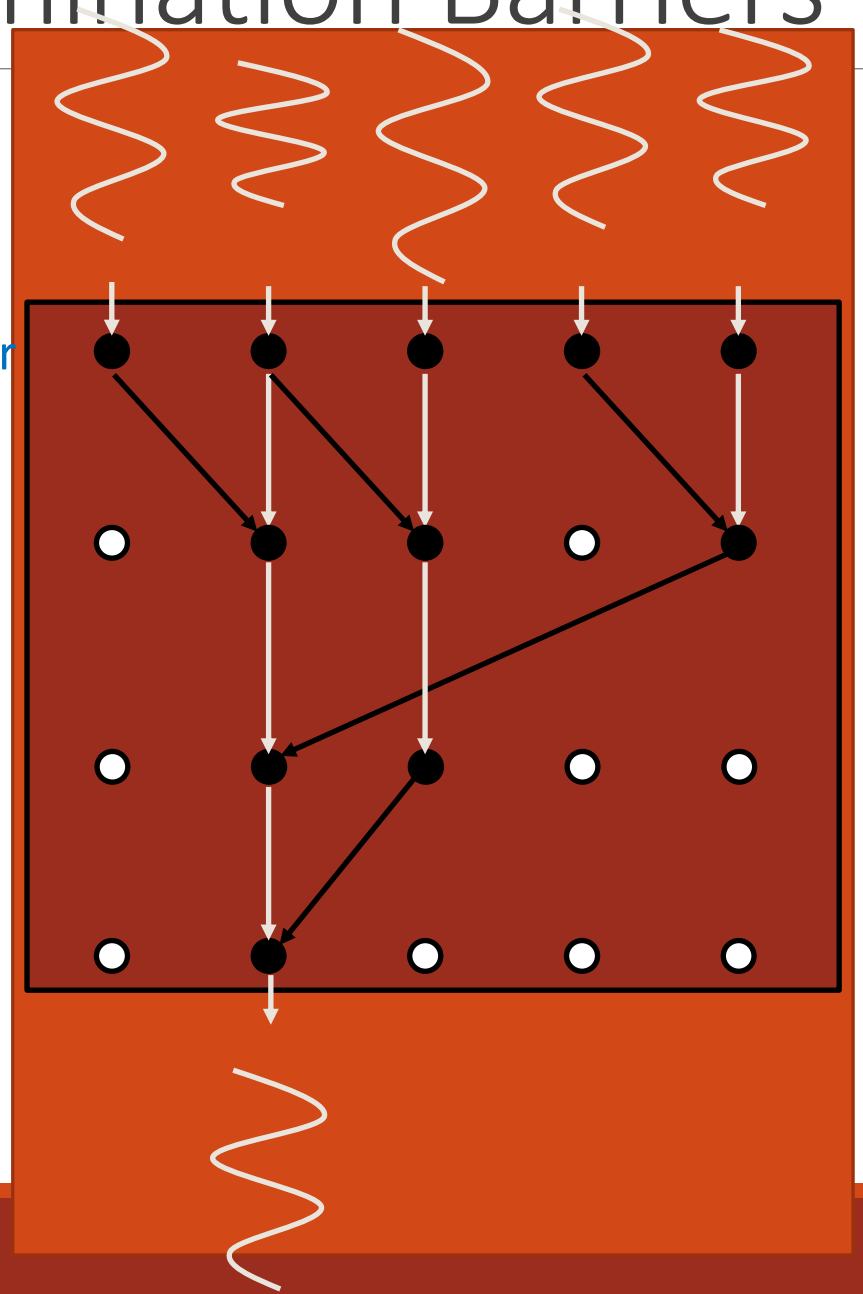
Any thread exits barrier

➔

All threads entered barrier

Forward propagation proves:

All threads exit barrier



Just follow dependence graph backwards!

- Each exiting thread is the root of a binary tree with all entering threads as leaves (requires log P rounds)

Proof is symmetric (mod P) for all threads

Dissemination Implementation #1

```
const int rounds = log(P);
bool flags[P][rounds]; // allocated in local storage per thread


void barrier() {
  for (round = 0 to rounds – 1) {
    partner = (tid + 2^round) mod P;
    flags[partner][round] = 1;
    while (flags[tid][round] == 0) { /* spin */ }
    flags[tid][round] = 0;
  }
}
```

What'd we forget?

Dissemination Implementation #2

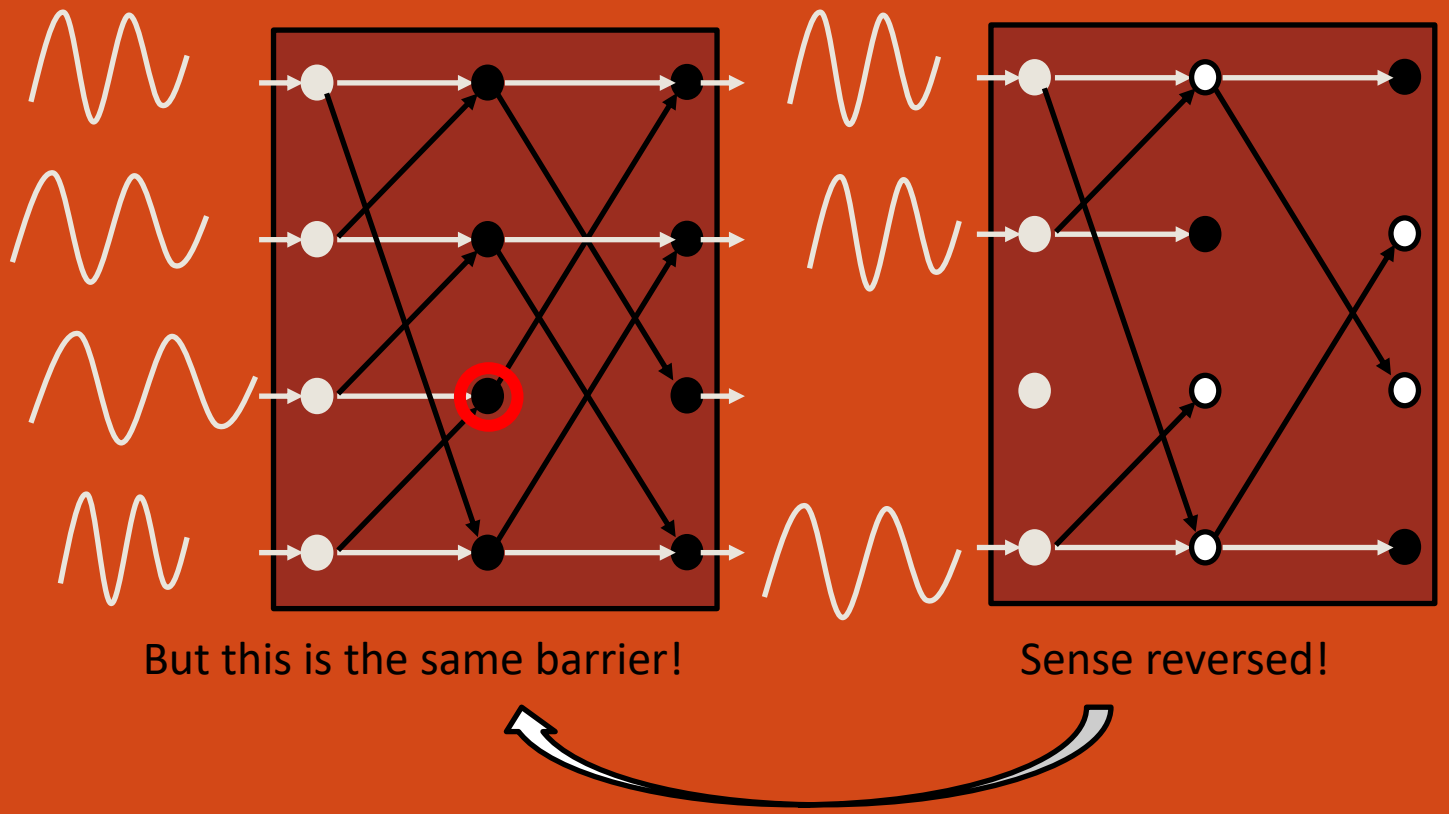
```
const int rounds = log(P);
bool flags[P][rounds]; // allocated in local storage per thread
local bool sense = false;

void barrier() {
  for (round = 0 to rounds - 1) {
    partner = (tid + 2^round) mod P;
    flags[partner][round] = !sense;
    while (flags[tid][round] == sense) { /* spin */ }
  }
  sense = !sense;
}
```

Good?

Sense Reversal in Dissemination

Thread 2 isn't scheduled for a while…

Thread 2 blocks waiting on old sense



Dissemination Implementation #3

```
const int rounds = log(P);

bool flags[P][2][rounds]; // allocated in local storage per thread

local bool sense = false;

local int parity = 0;


void barrier() {

  for (round = 0 to rounds – 1) {

    partner = (tid + 2^round) mod P;

    flags[partner][parity][round] = !sense;

    while (flags[tid][parity][round] == sense)

                        { /* spin */ }

  }

  if (parity == 1) {

    sense = !sense;

  }

  parity = 1 – parity;

}
```

Allocate 2 barriers, alternate between them via ‘parity’.

Reverse sense every other barrier.

Dissemination Barrier Analysis

Local spinning only

$O(\log P)$ messages on critical path

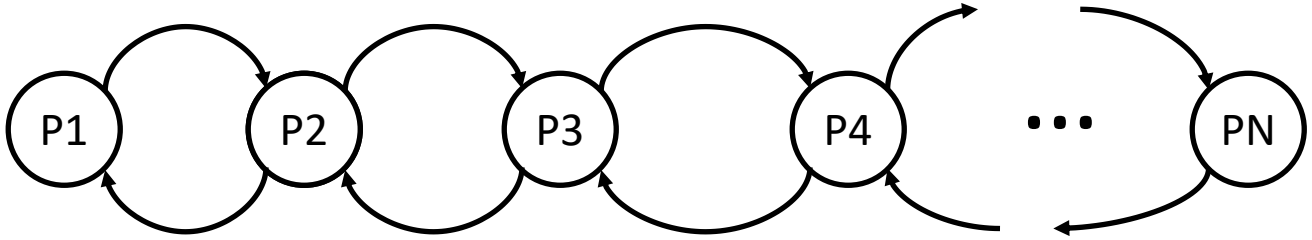
$O(P \log P)$ space – $\log P$ variables per processor

$O(P \log P)$ total messages on network

Only uses loads & stores

Minimum Barrier Traffic

What is the minimum number of messages needed to implement a barrier with N processors?



P-1 to notify everyone arrives

P-1 to wakeup

➔ 2P – 2 total messages minimum

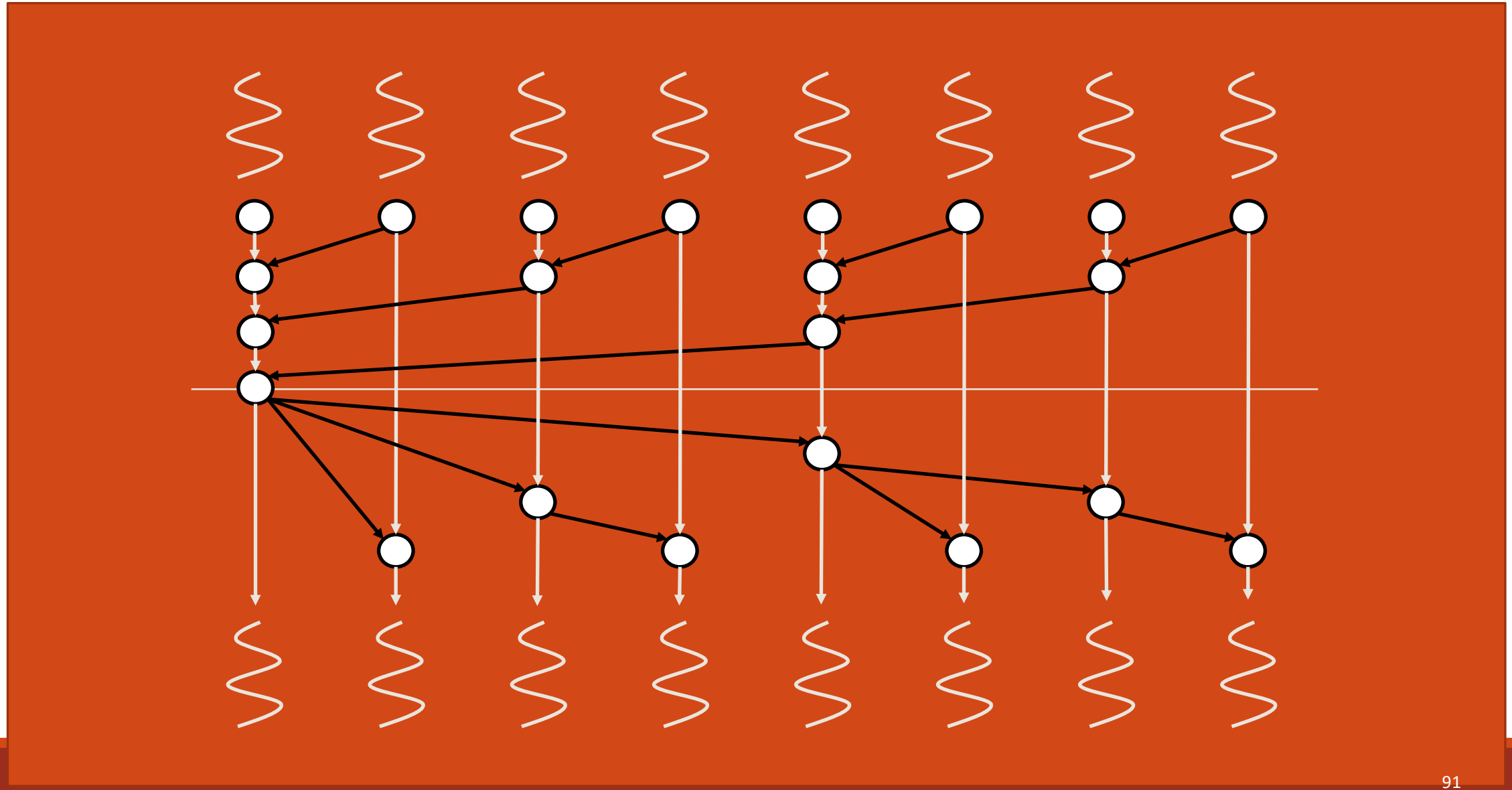
Tournament Barrier

Binary combining tree

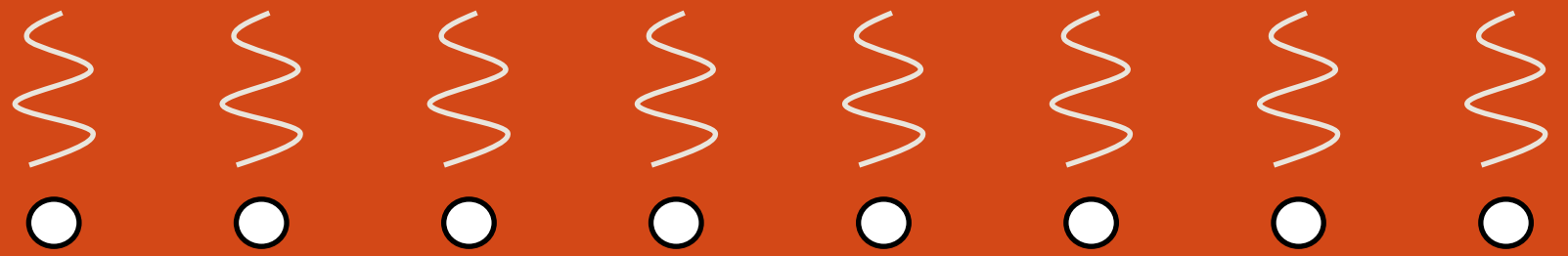
Representative processor at a node is statically chosen
- No fetch&op needed

In round k, proc $i = 2^k$ sets a flag for proc $j = i - 2^k$
- i then drops out of tournament and j proceeds in next round
- i waits for signal from partner to wakeup
  - Or, on coherent machines, can wait for global flag

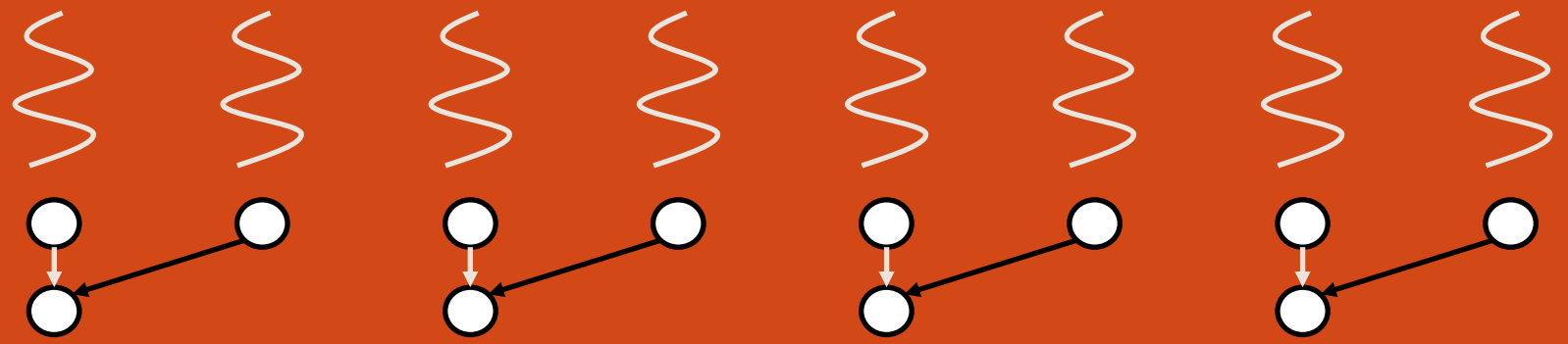
Tournament Barrier with P=8



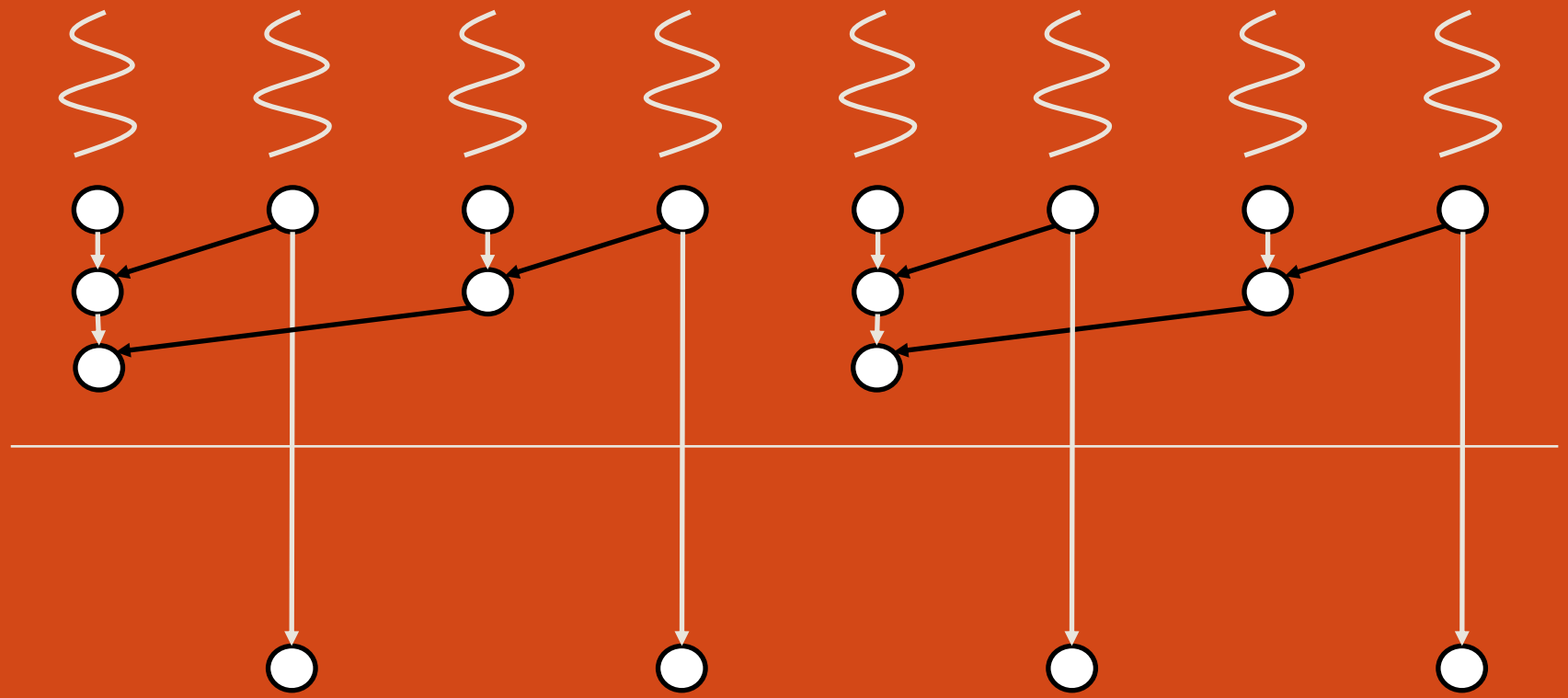
Tournament Barrier with P=8



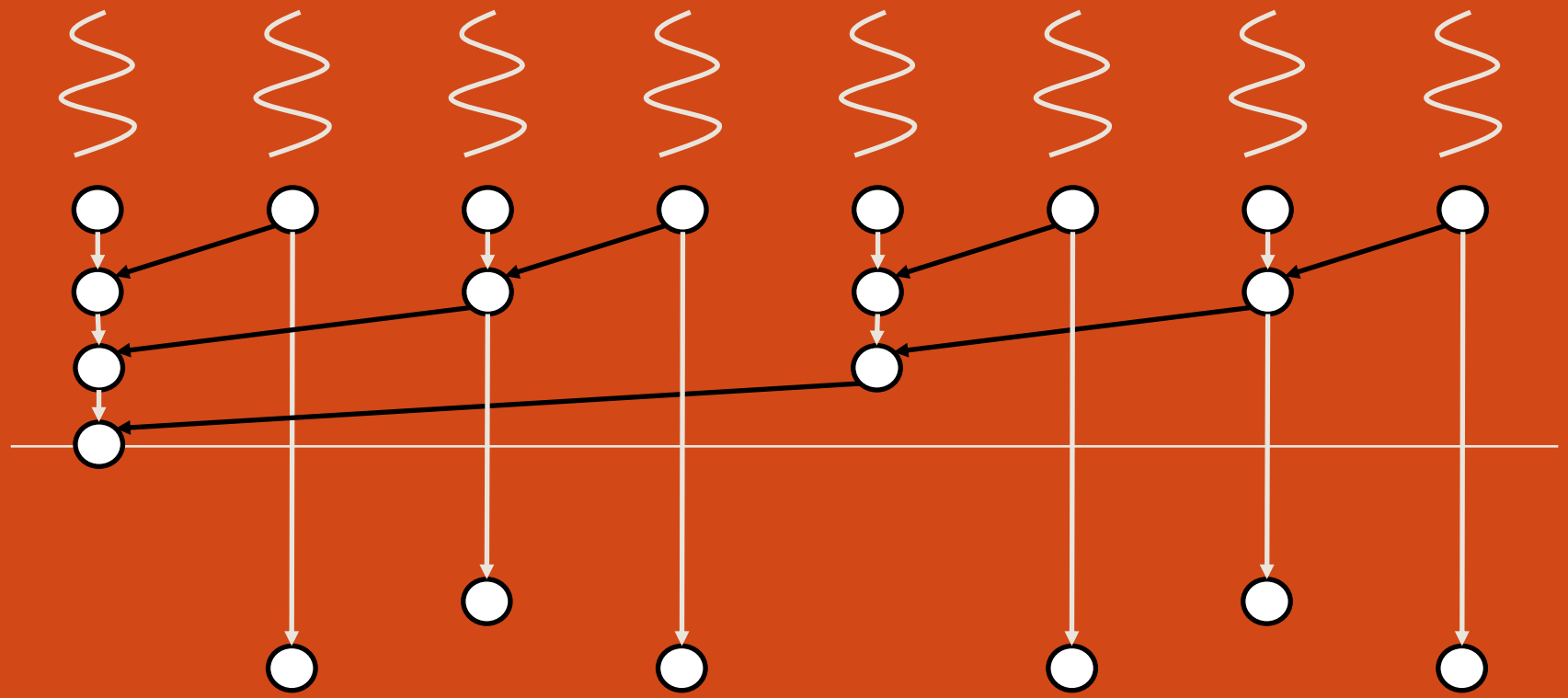
Tournament Barrier with P=8



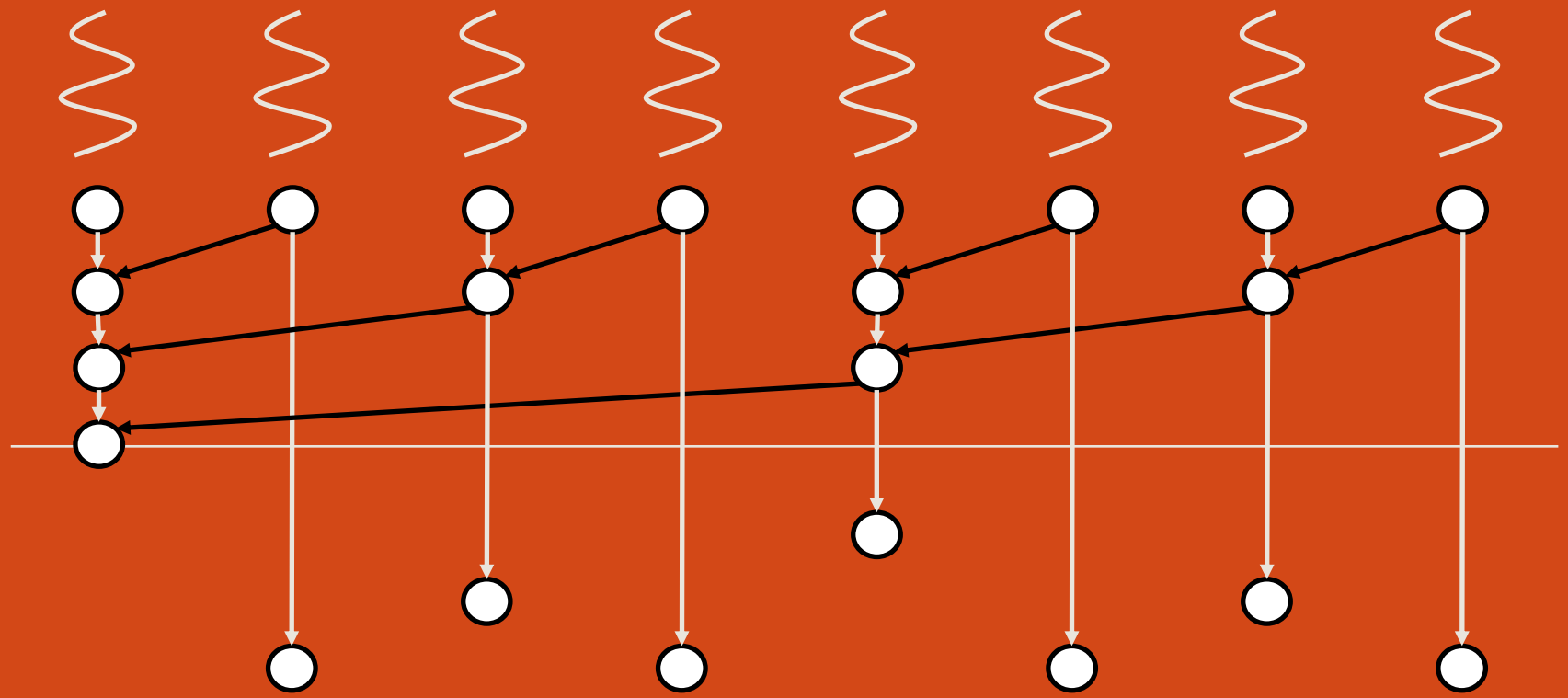
Tournament Barrier with P=8



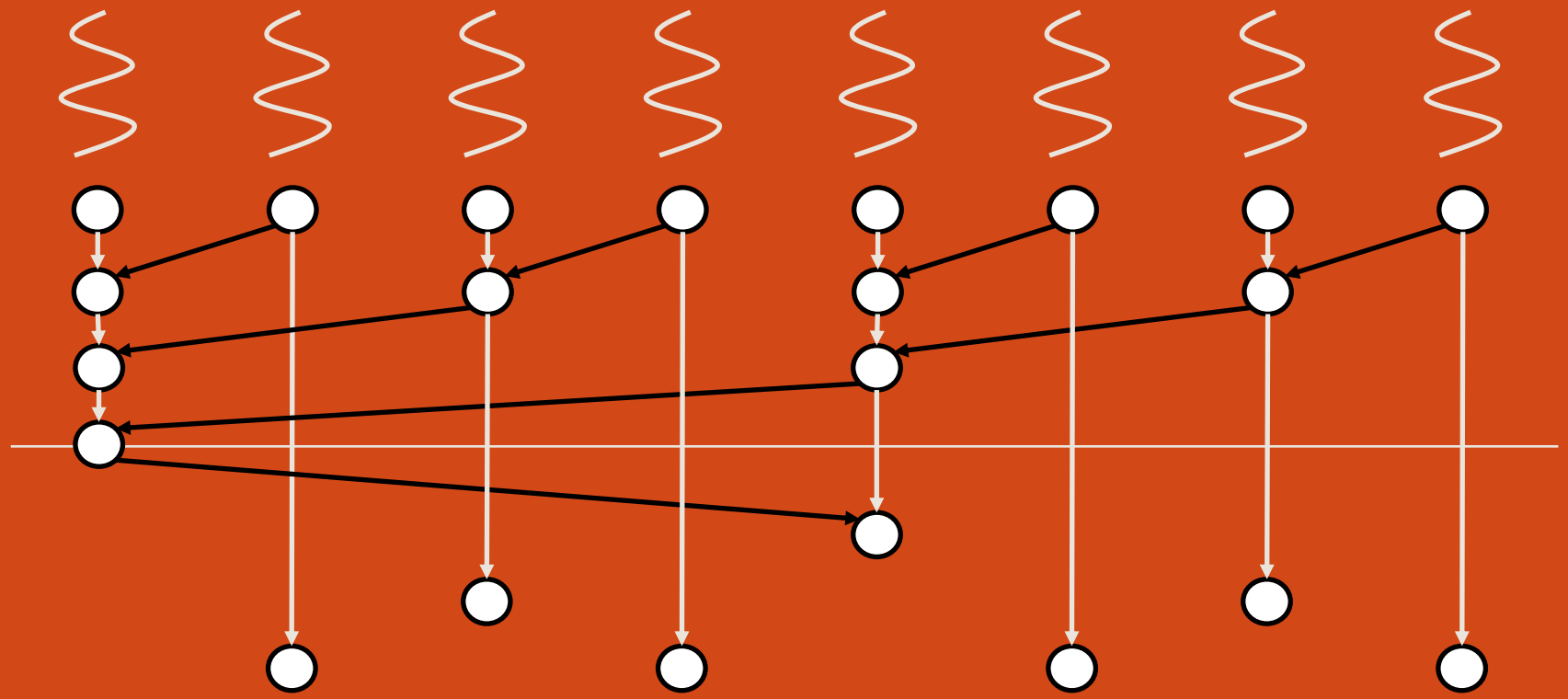
Tournament Barrier with P=8



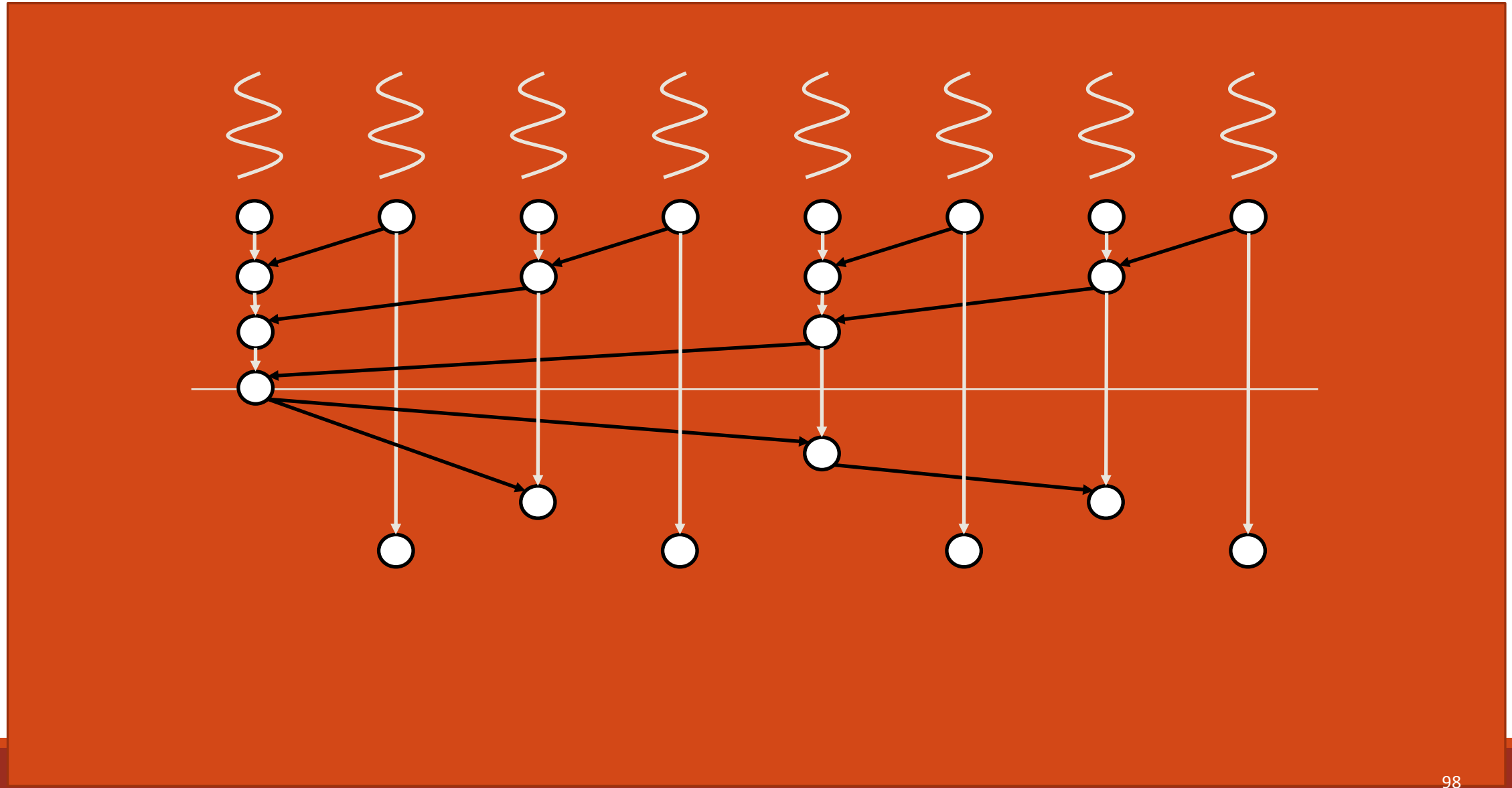
Tournament Barrier with P=8



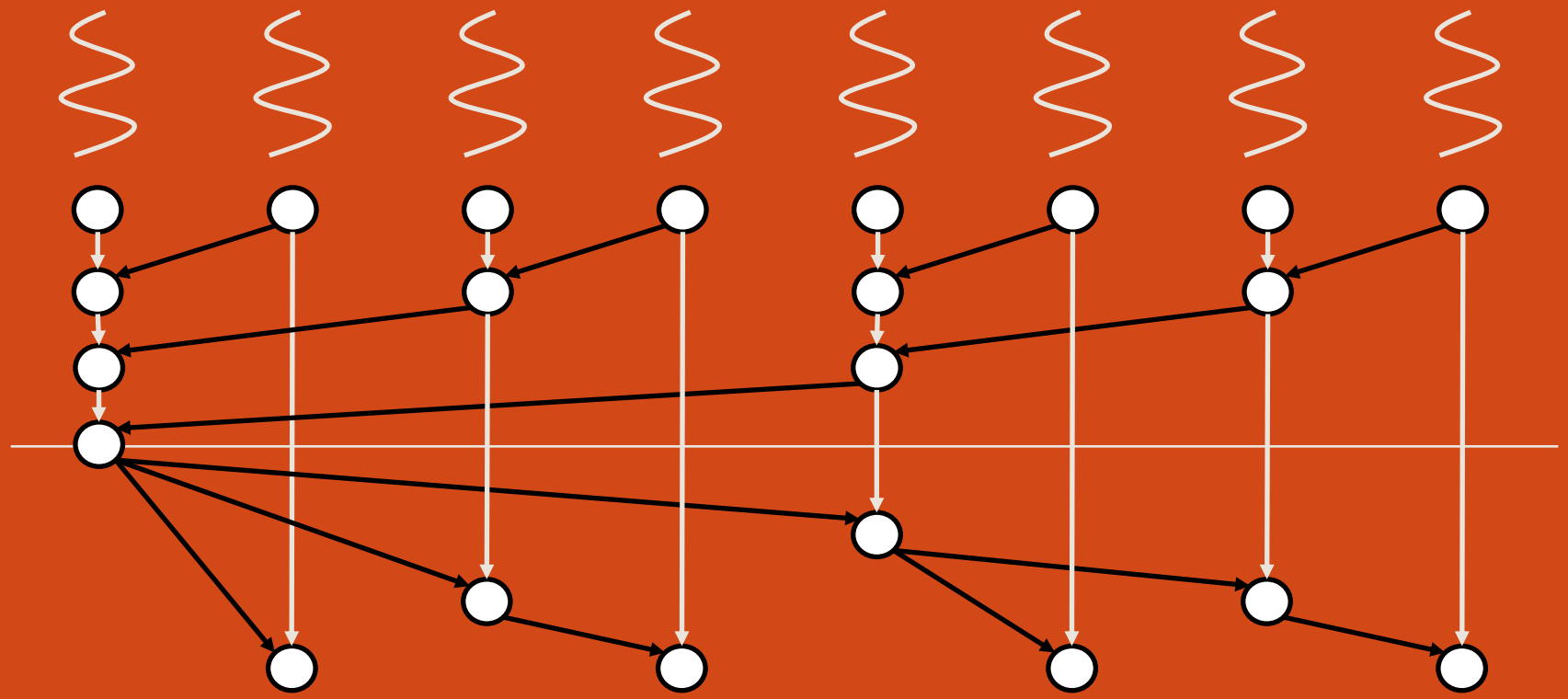
Tournament Barrier with P=8



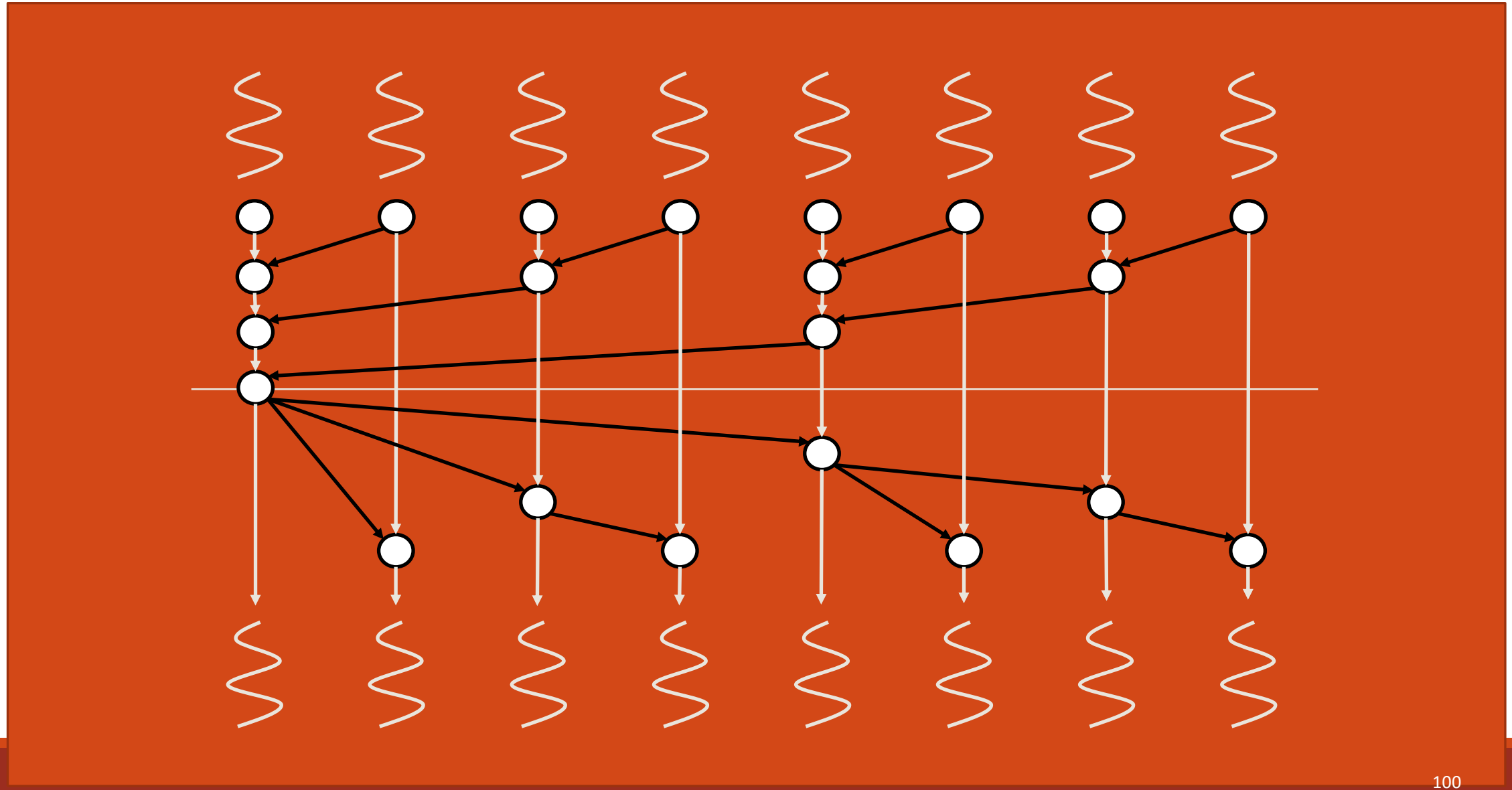
Tournament Barrier with P=8



Tournament Barrier with P=8



Tournament Barrier with P=8



Why Tournament Barrier Works

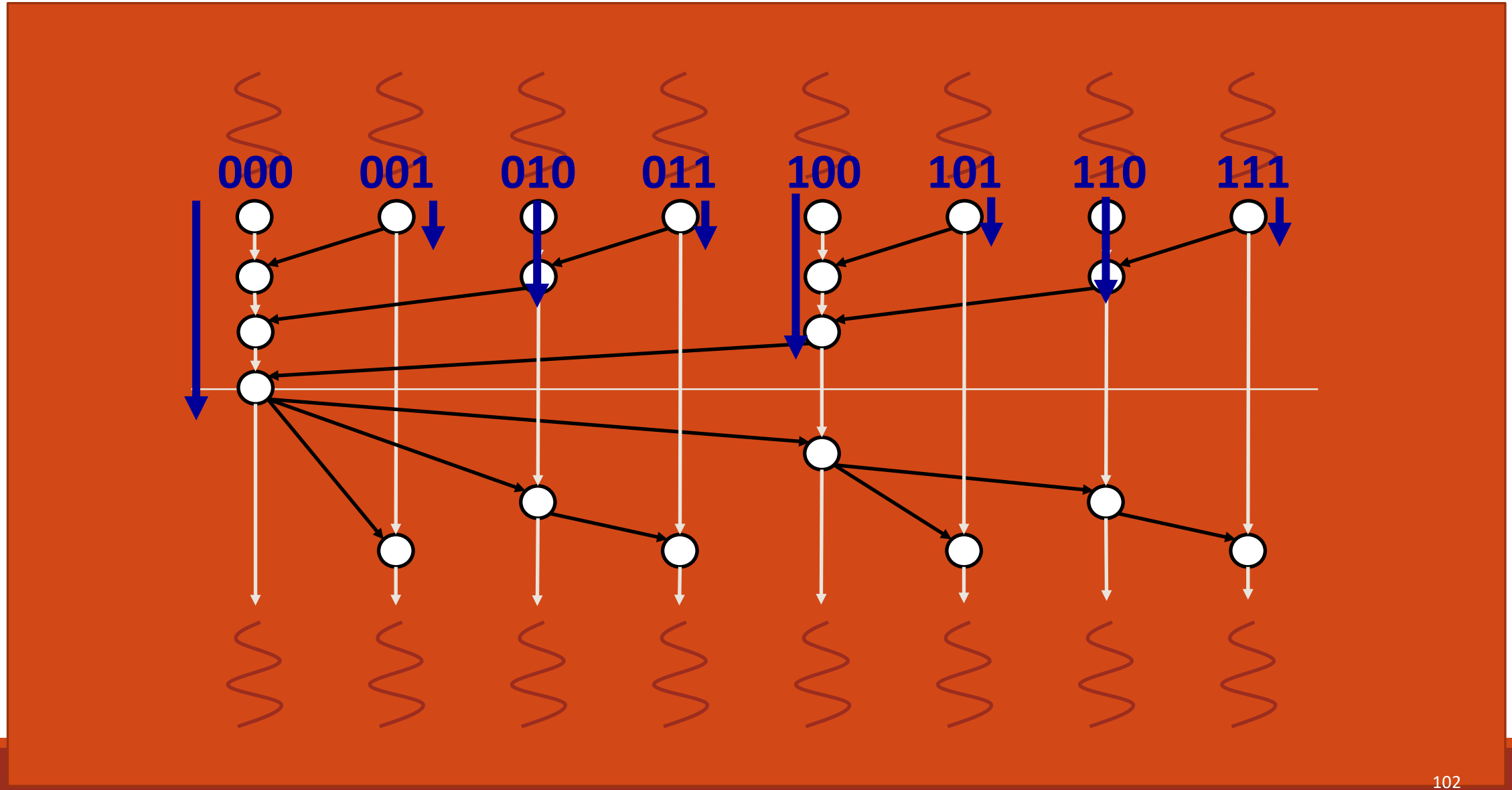
As before, threads can progress at different rates through tree

Easy to show correctness:

- Tournament root must unblock for any thread to exit barrier
- Root depends on all threads (leaves of tree)

Implemented by two loops, up & down tree
Depth encoded by first 1 in thread id bits

Depth == First 1 in Thread ID



Tournament Barrier Implementation

```
// for simplicity, assume P power of 2
void barrier(int tid) {
  int round;
  for (round = 0; // wait for children (depth == first 1)
       ((P | tid) & (1 << round)) == 0; round++) {
    while (flags[tid][round] != sense) { /* spin */ }
  }
  if (round < logP) { // signal + wait for parent (all but root)
    int parent = tid & ~((1 << (round+1)) - 1);
    flags[parent][round] = sense;
    while (flags[tid][round] != sense) { /* spin */ }
  }
  while (round-- > 0) { // wake children
    int child = tid | (1 << round);
    flags[child][round] = sense;
  }
  sense = !sense;
}
```

Tournament Barrier Analysis

Local spinning only

$O(\log P)$ messages on critical path (but > dissemination)

$O(P)$ space

$O(P)$ total messages on network

Only uses loads & stores

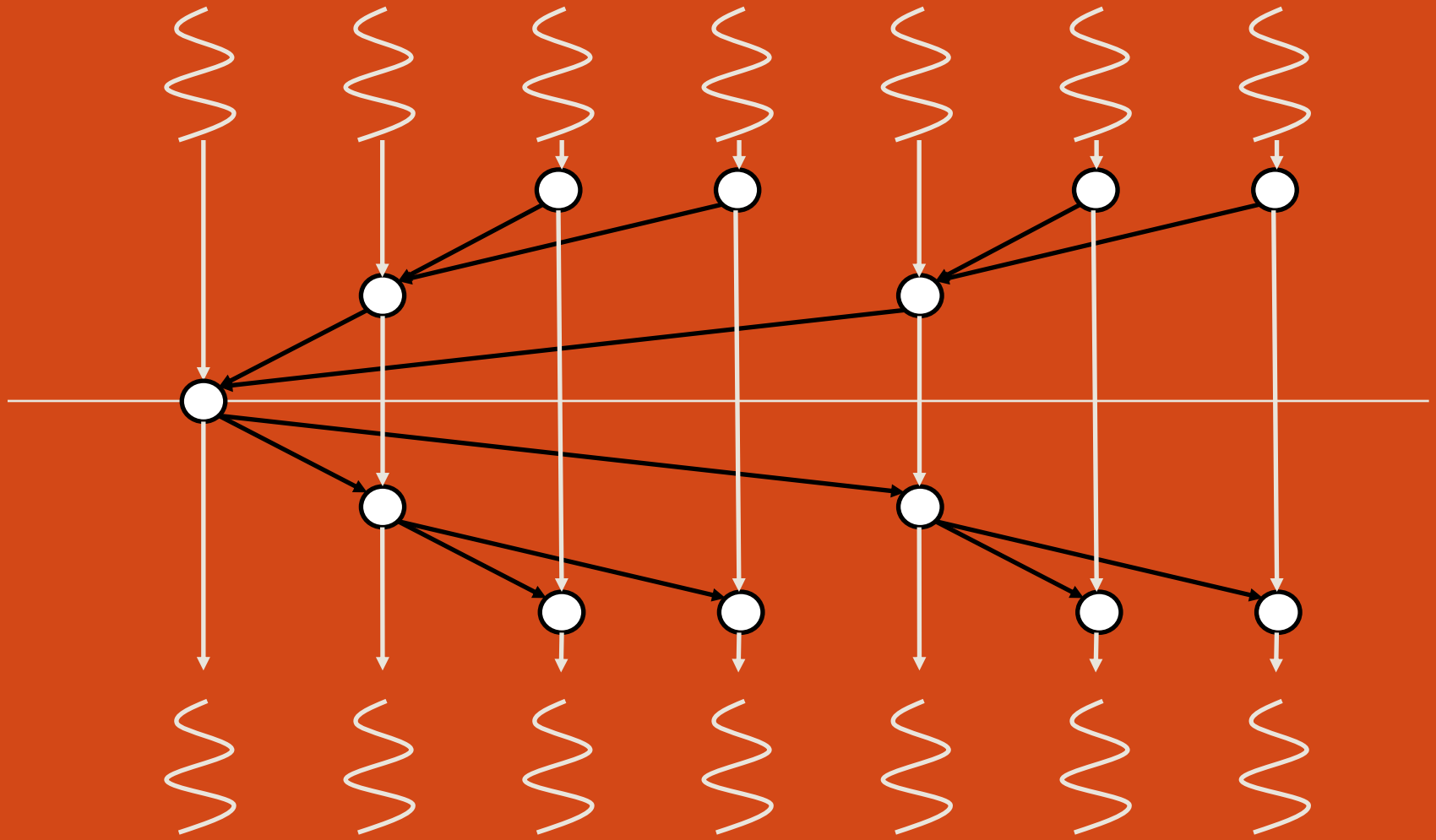
MCS Software Barrier

Modifies tournament barrier to allow static allocation in wakeup tree, and to use sense reversal

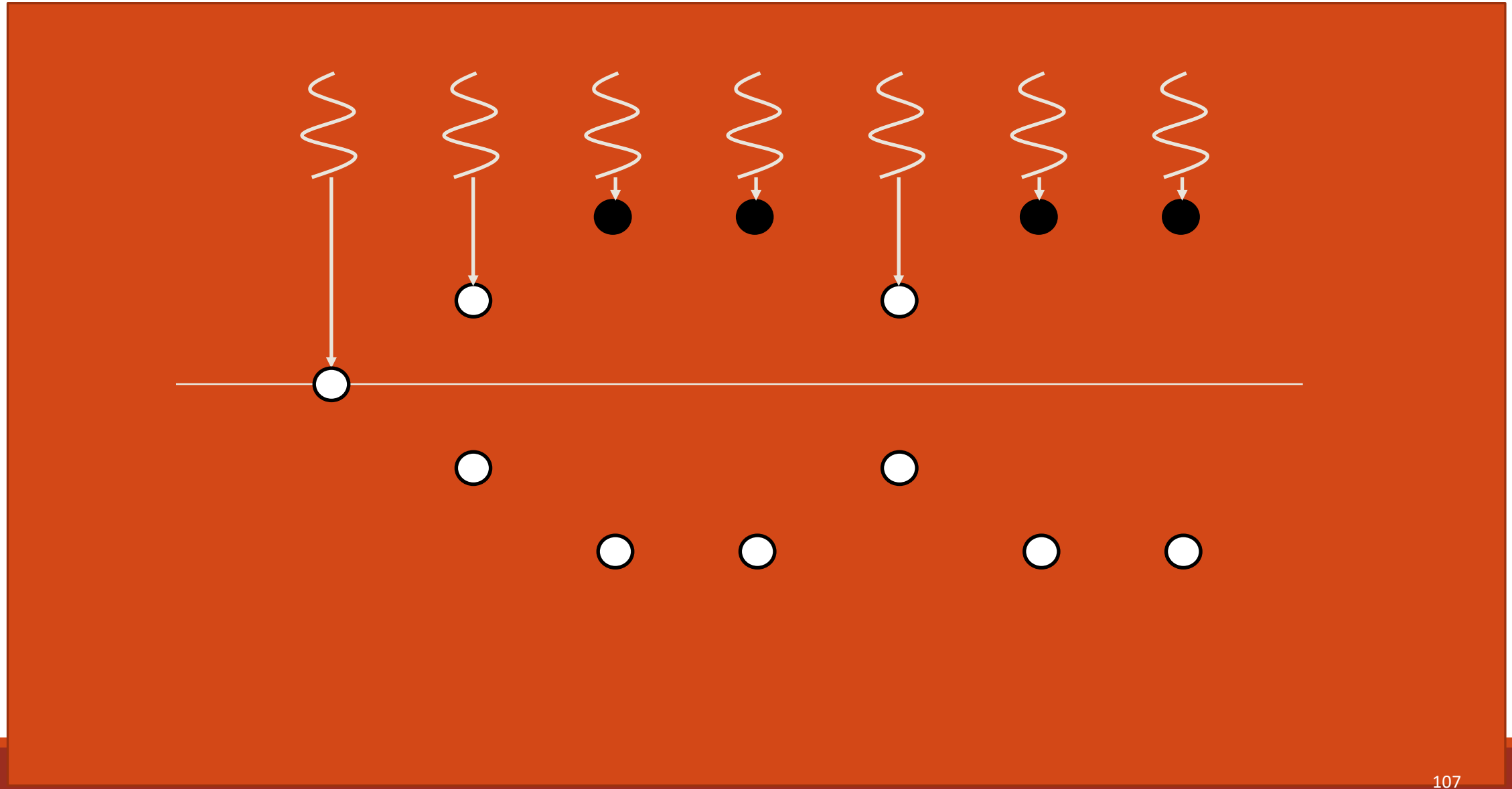
Every thread is a node in two P-node trees:

- has pointers to its parent building a fan-in-4 arrival tree
  - fan-in = flags / word for parallel checks
- has pointers to its children to build a fan-out-2 wakeup tree

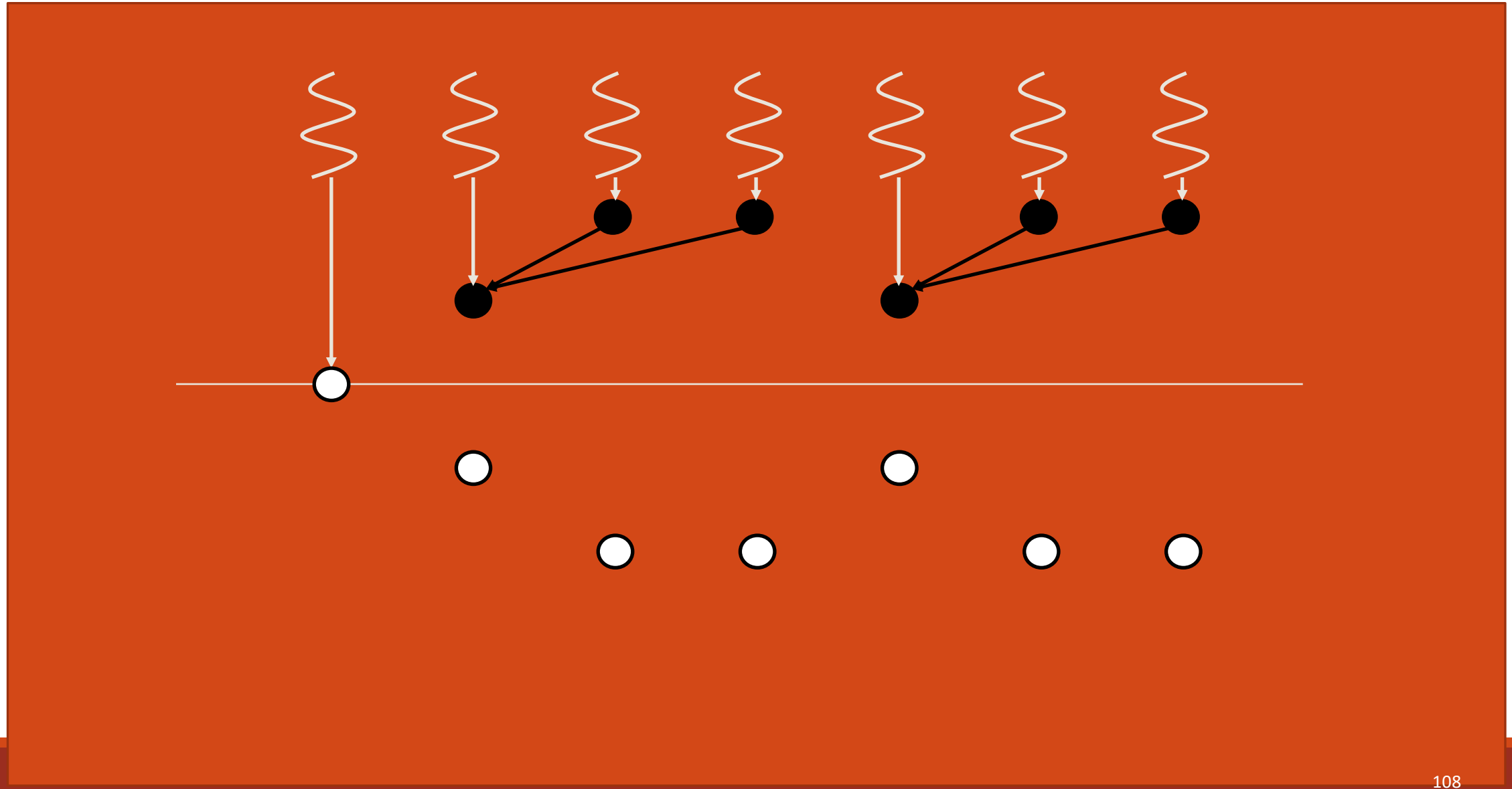
MCS Barrier with P=7



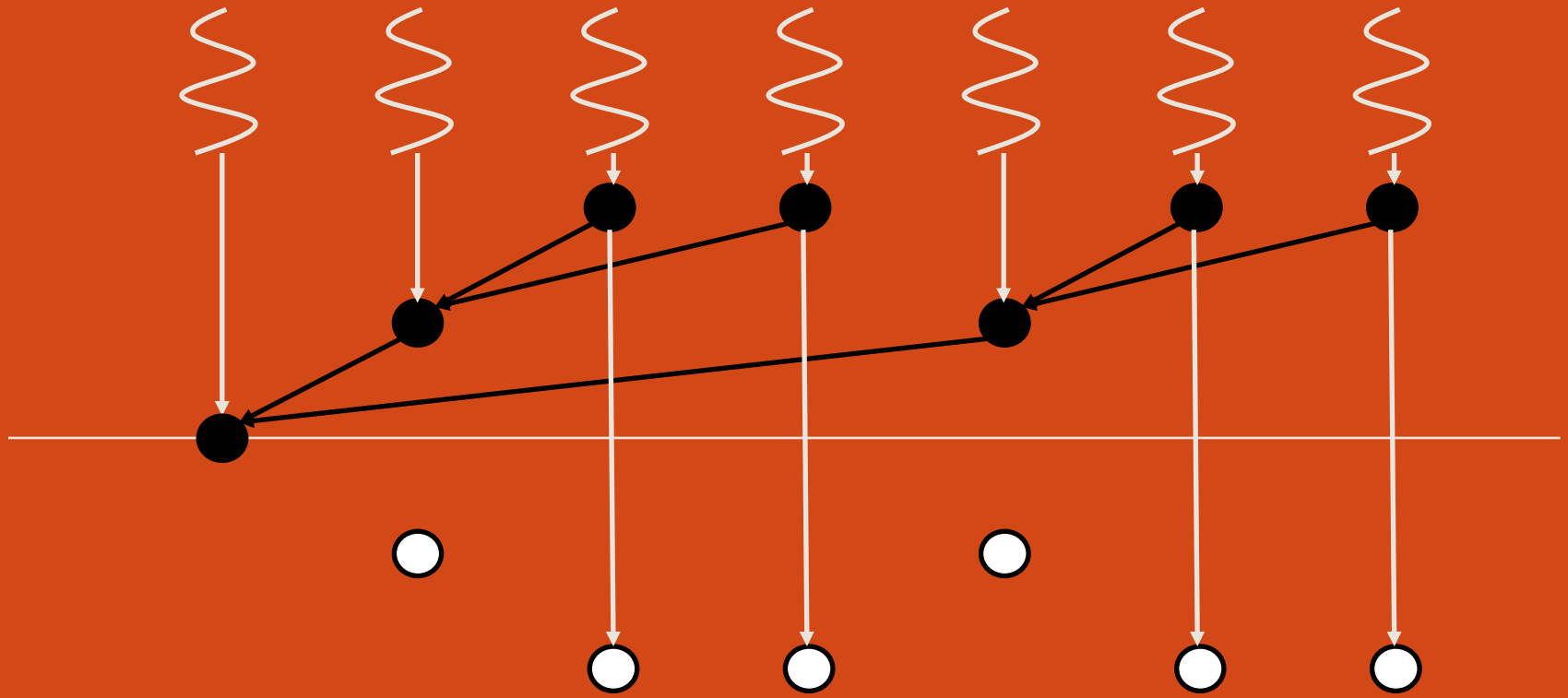
MCS Barrier with P=7



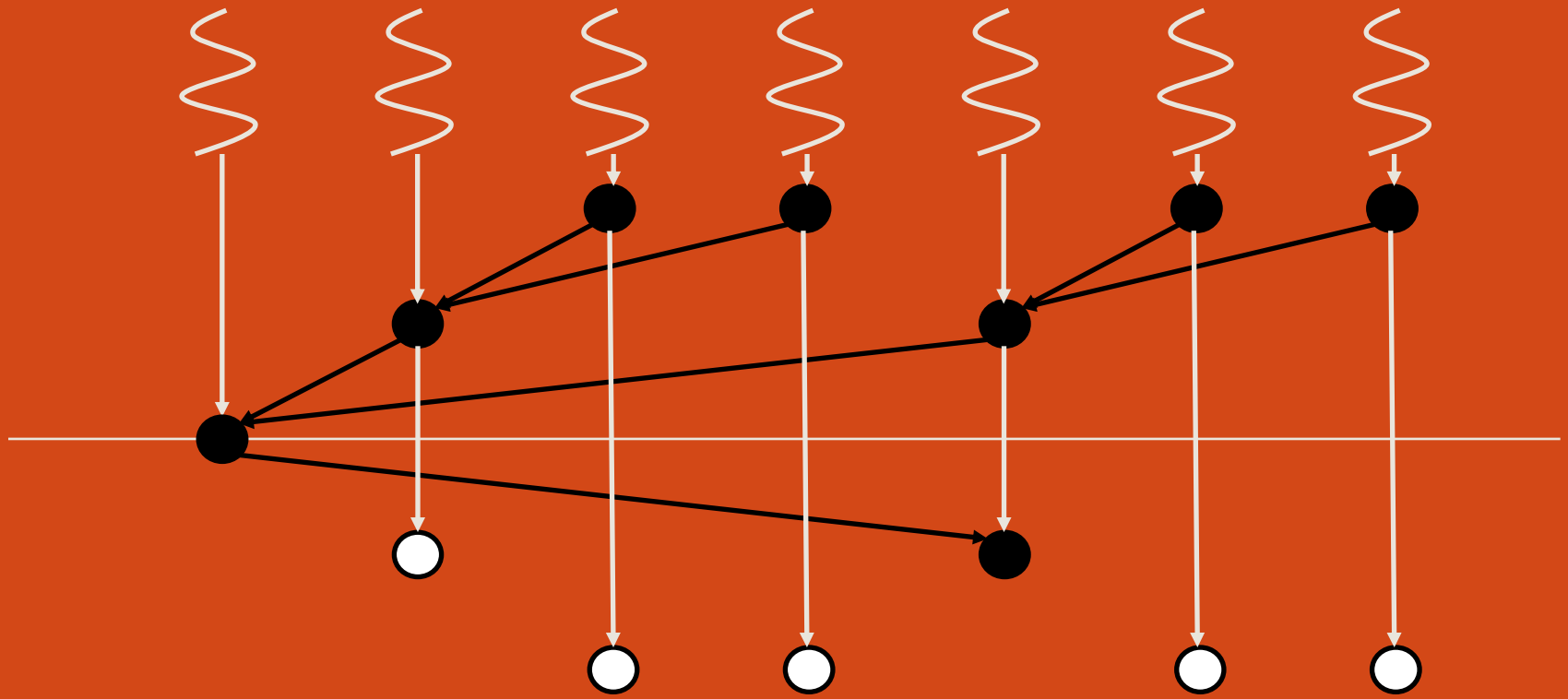
MCS Barrier with P=7



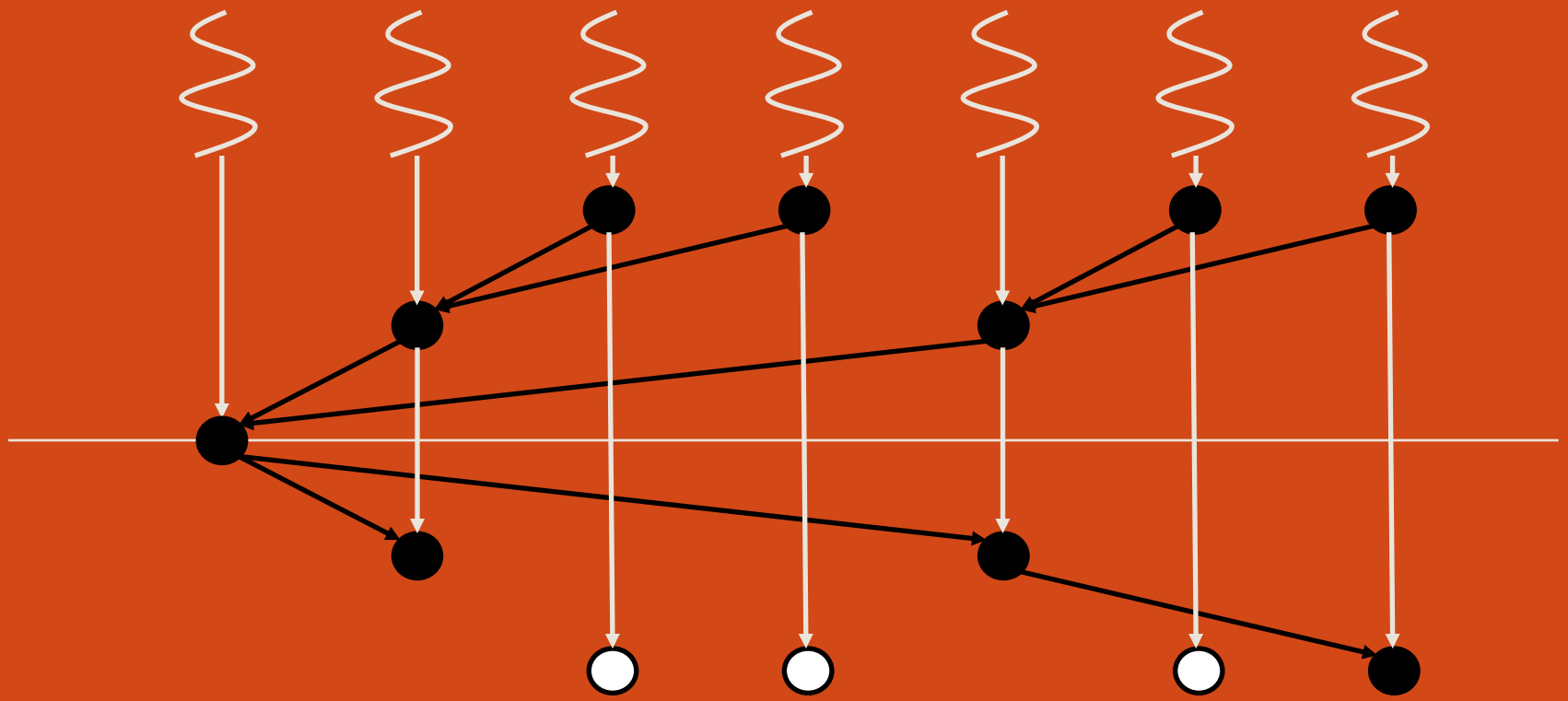
MCS Barrier with P=7



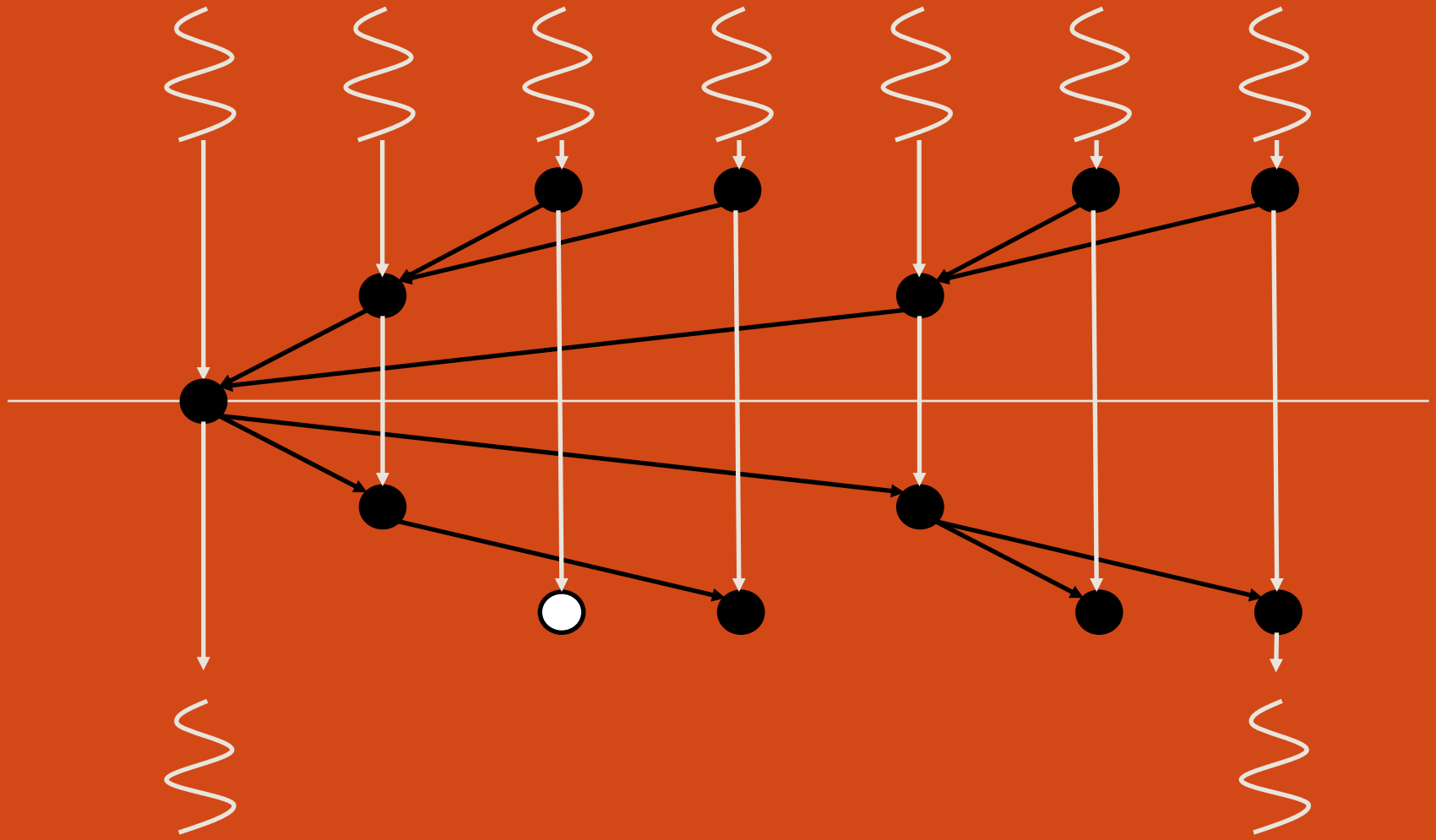
MCS Barrier with P=7



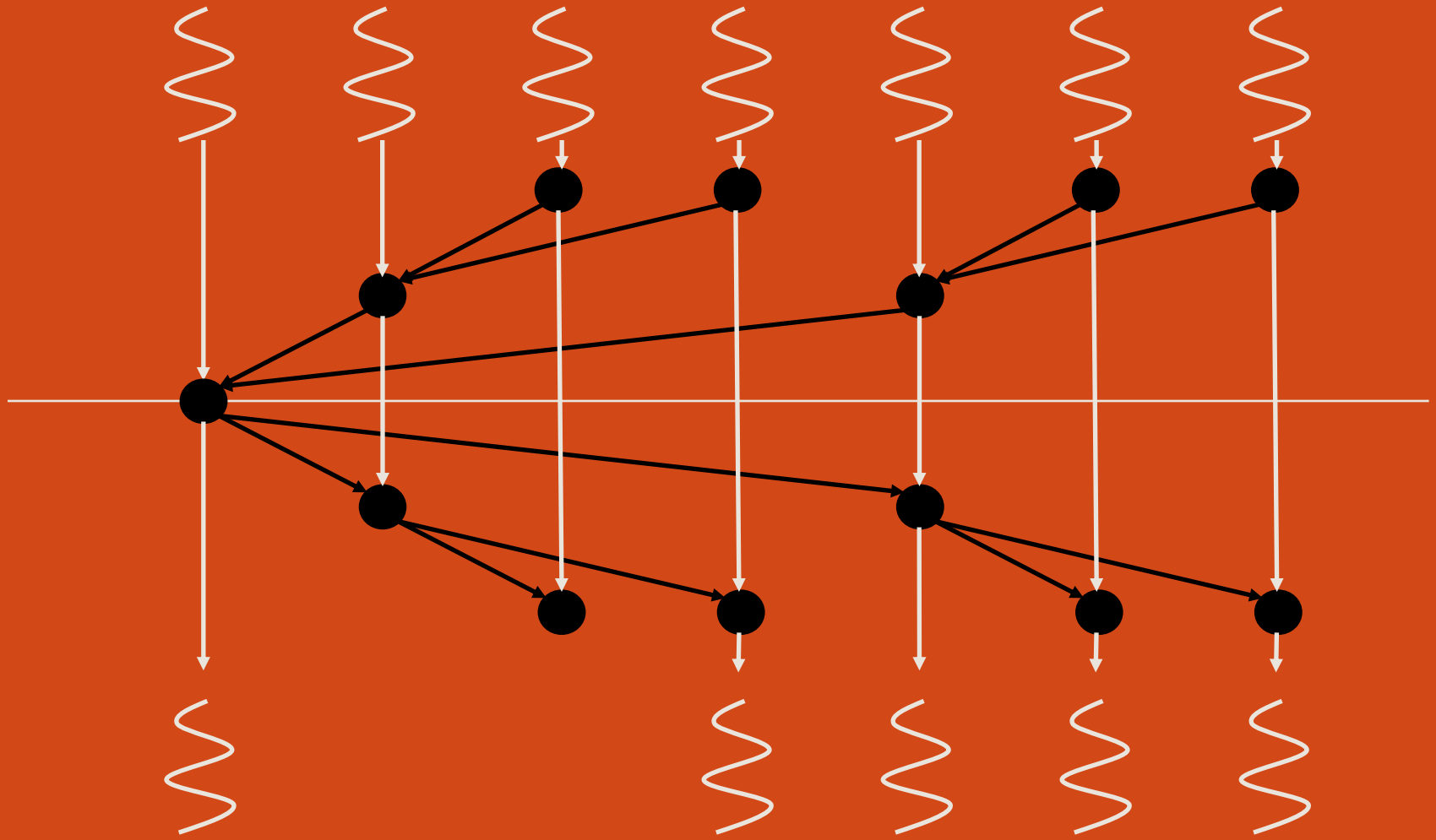
MCS Barrier with P=7



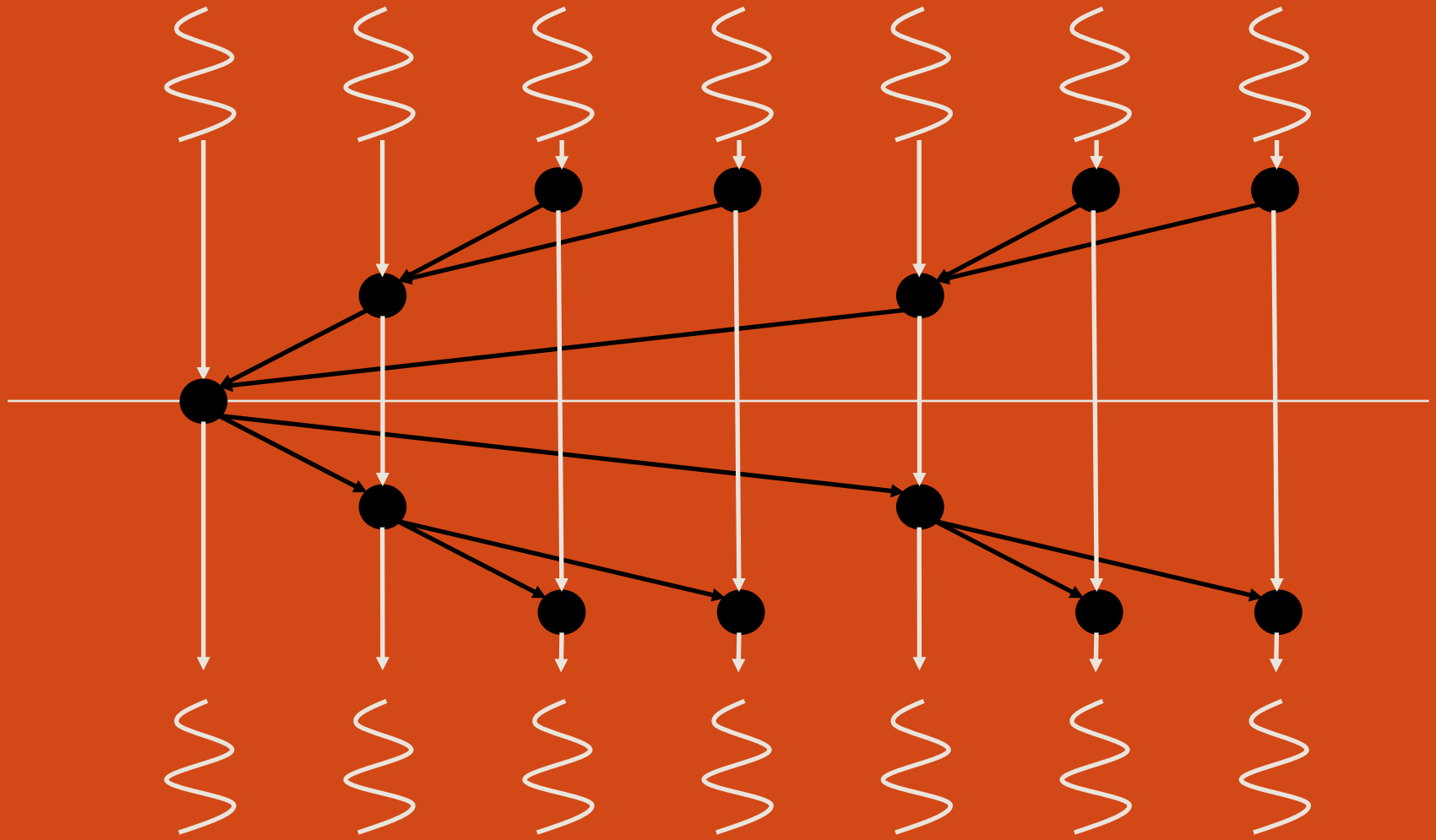
MCS Barrier with P=7



MCS Barrier with P=7



MCS Barrier with P=7



MCS Software Barrier Analysis

Local spinning only

$O(\log P)$ messages on critical path

$O(P)$ space for P processors

Achieves theoretical minimum communication of $(2P - 2)$ total messages

Only needs loads & stores

Review: Critical path

All critical paths $O(\log P)$, except centralized $O(P)$

But beware network contention!

➔ Linear factors dominate bus

Review: Network transactions

Centralized, combining tree:

- $O(P)$ if broadcast and coherent caches;
- unbounded otherwise

Dissemination:

- $O(P \log P)$

Tournament, MCS:

- $O(P)$

Review: Storage requirements

Centralized:

- $O(1)$

MCS, combining tree:

- $O(P)$

Dissemination, Tournament:

- $O(P \log P)$

Review: Primitives Needed

Centralized and software combining tree:

- atomic increment / atomic decrement

Others (dissemination, tournament, MCS):

- atomic read
- atomic write

Barrier recommendations

Without broadcast on distributed memory:

- *Dissemination*
- MCS is good, only critical path length is about 1.5X longer (for wakeup tree)
- MCS has somewhat better network load and space requirements

Cache coherence with broadcast (e.g., a bus):

- *MCS with flag wakeup*
- But centralized is best for modest numbers of processors

Big advantage of *centralized* barrier:

- Adapts to changing number of processors across barrier calls

Synchronization Summary

Required for concurrent programs

- mutual exclusion
- producer-consumer
- barrier

Hardware support

- ISA
- Cache
- Memory

Complex interactions

- Scalability, Efficiency, Indirect effects
- What about message passing?