Synchronization

15-740 SPRING'18

NATHAN BECKMANN

Types of Synchronization

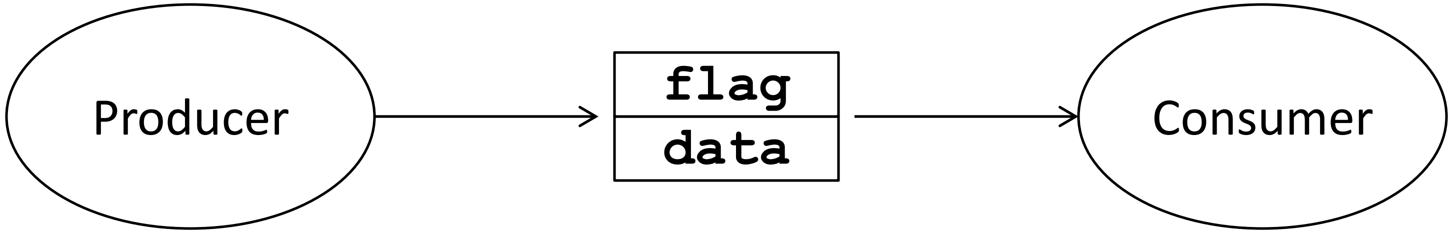
Mutual Exclusion

- Locks

Event Synchronization

- Global or group-based (barriers)
- Point-to-point (producer-consumer)

Simple Producer-Consumer Example



Producer

flag

data

Consumer

Initially **flag=0**

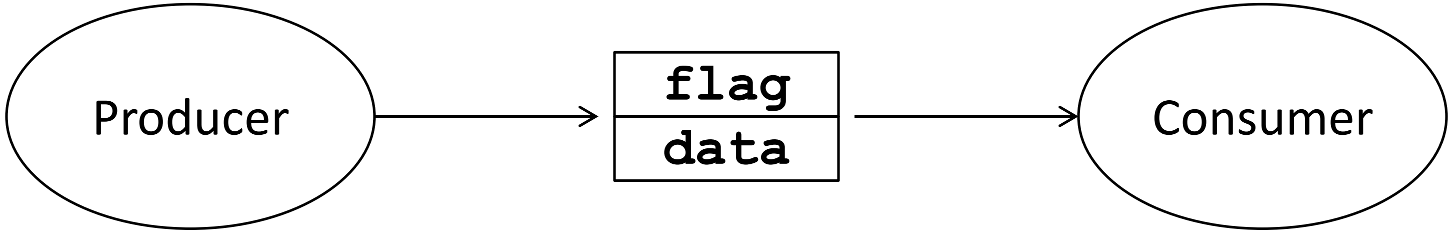
```
st xdata, (xdatap)

ld 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**

```
sd xdata, (xdatap)
li xflag, 1
sd xflag, (xflagp)
```

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

Dependencies from sequential ISA

Dependencies added by sequentially consistent memory model

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

Relaxed memory models

Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies were needed

Which dependencies are dropped depends on the particular memory model

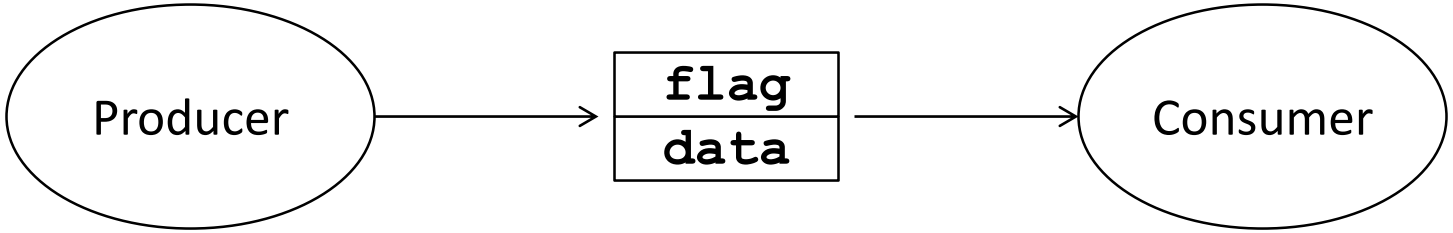
- IBM370, TSO, PSO, WO, PC, Alpha, RMO, …

How to introduce needed dependencies varies by system

- Explicit FENCE instructions (sometimes called sync or memory barrier instructions)
- Implicit effects of atomic memory instructions

Programmers supposed to work with this????

Fences in producer-consumer

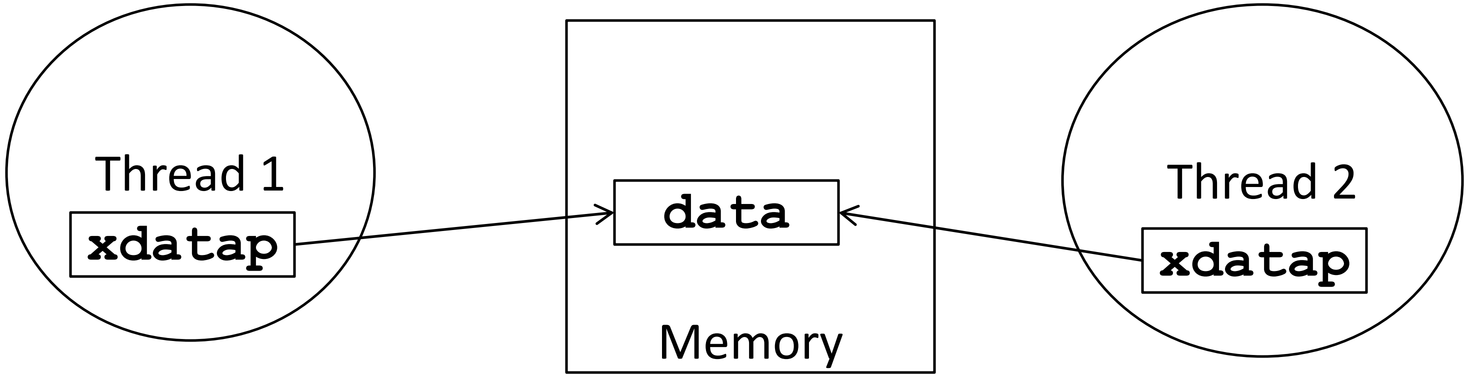


Initially **flag =0**

```
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)
```

Simple mutual-exclusion example



```
// Both threads execute:

ld xdata, (xdatap)

add xdata, 1

st xdata, (xdatap)
```

Is this correct?

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

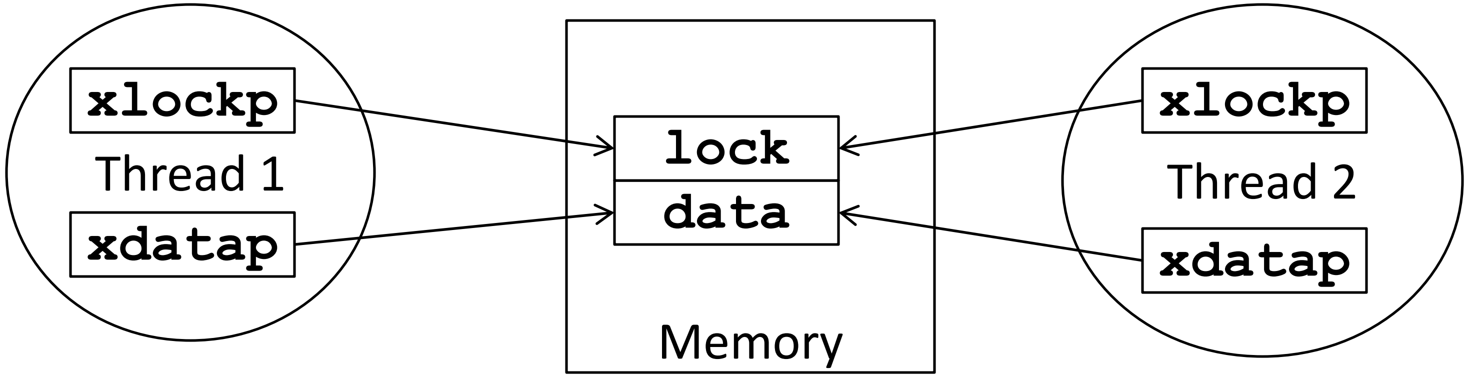
Fetch&Op

- 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



Assumes SC memory model

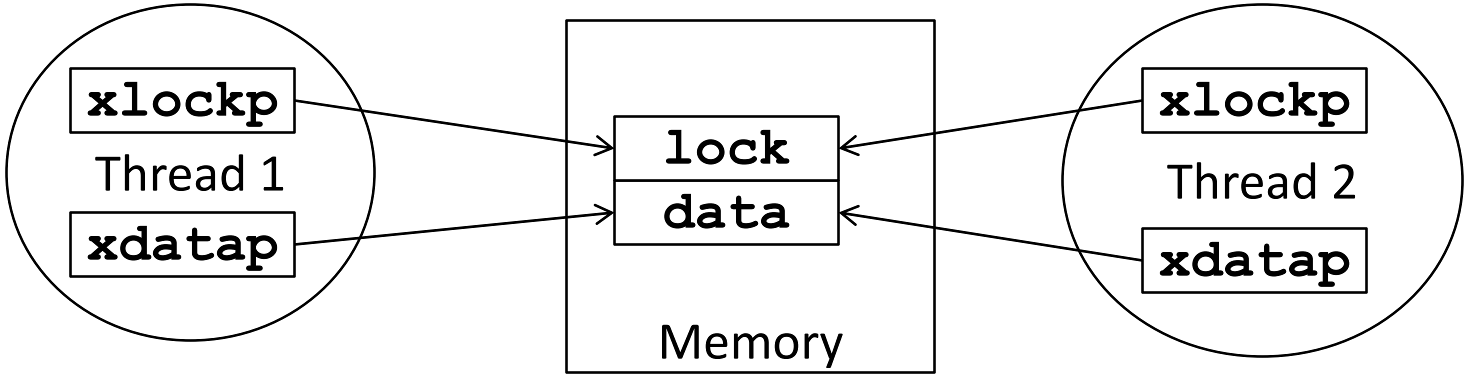
```
      li xone, 1
spin: amoswap xlock, xone, (xlockp)
      bnez xlock, spin
      ld xdata, (xdatap)
      add xdata, 1
      st xdata, (xdatap)
      st x0, (xlockp)
```

Acquire Lock

Critical Section

Release Lock

Mutual Exclusion with Relaxed Consistency



```
      li xone, 1
spin: amoswap xlock, xone, (xlockp)
      bnez xlock, spin
      fence.r.r
```

Acquire Lock

```
      ld xdata, (xdatap)
      add xdata, 1
      sd xdata, (xdatap)
```

Critical Section

```
      fence.w.w
      sd x0, (xlockp)
```

Release Lock

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 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:

Get exclusive permissions

Don’t respond to invalidates

Perform operation (e.g., add in fetch&add)

Resume normal operation

Implementing LL/SC

Invalidation-based directory protocol

- SC requests exclusive permissions
- If requestor is still sharer, success
- Otherwise, fail and don’t get permissions (invalidation in flight)

Add *link register* to store address of LL

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

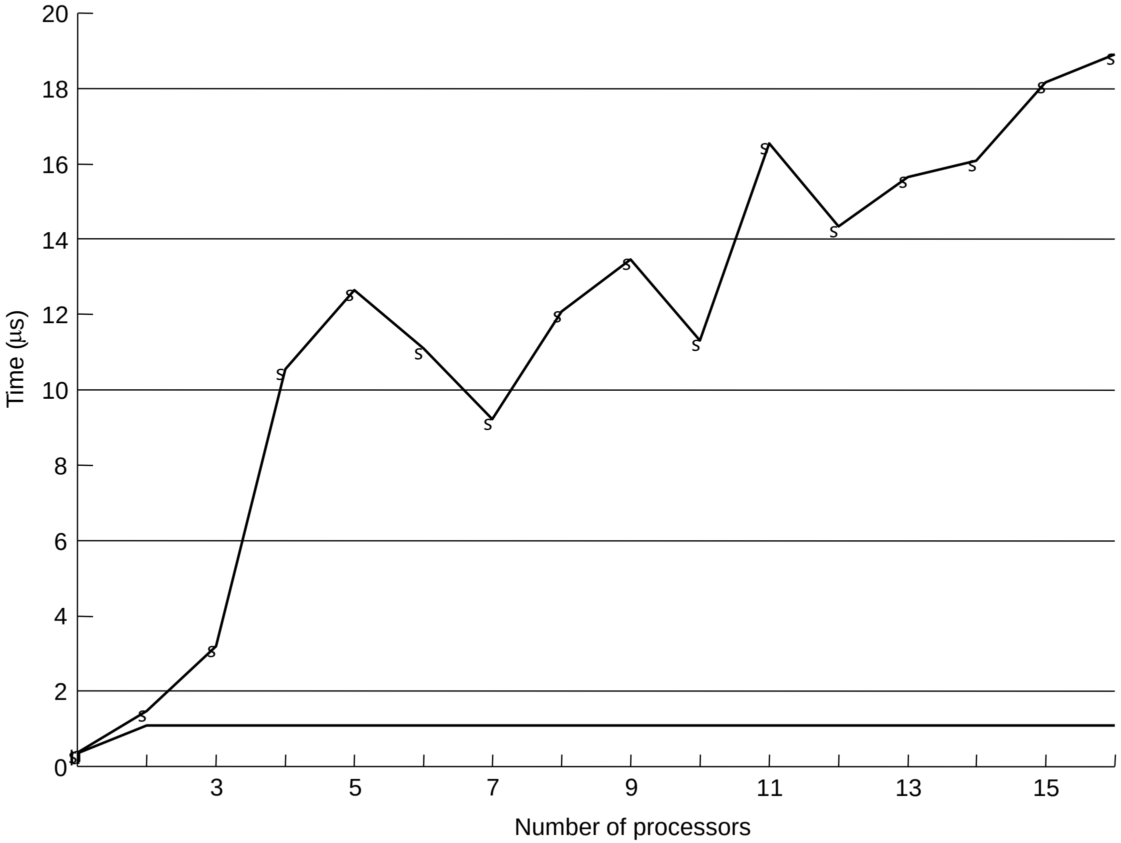
How to Evaluate?

- Scalability
- Network load
- Single-processor latency
- Space Requirements
- Fairness
- Required atomic operations
- Sensitivity to co-scheduling

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



Evaluation of Test&Set based lock

```
lock:       t&s   reg, (loc)
            bnz   lock
            ret


unlock:     st    location, #0
            ret
```

Scalability	poor
Network load	large
Single-processor latency	good
Space Requirements	good
Fairness	poor
Required atomic operations	T&S
Sensitivity to co-scheduling	good?

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

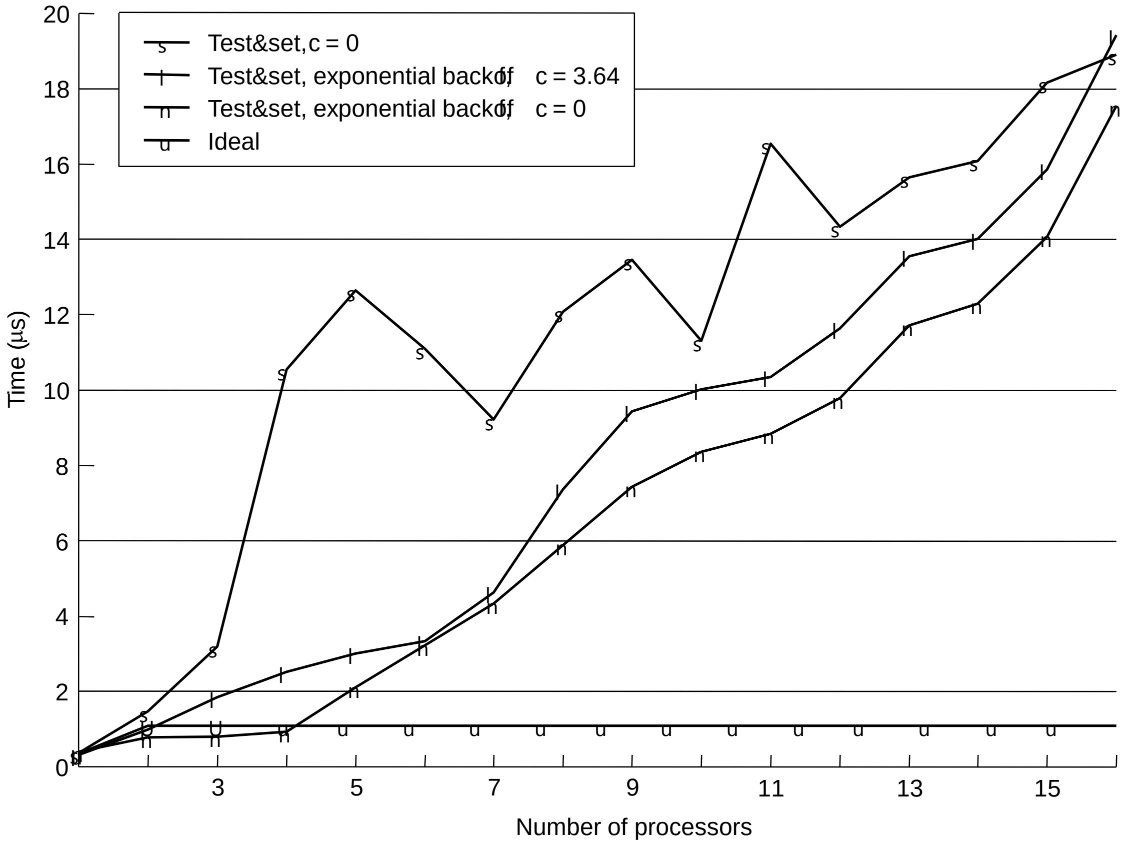
- new requestors back off for shorter times

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 with Update

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:

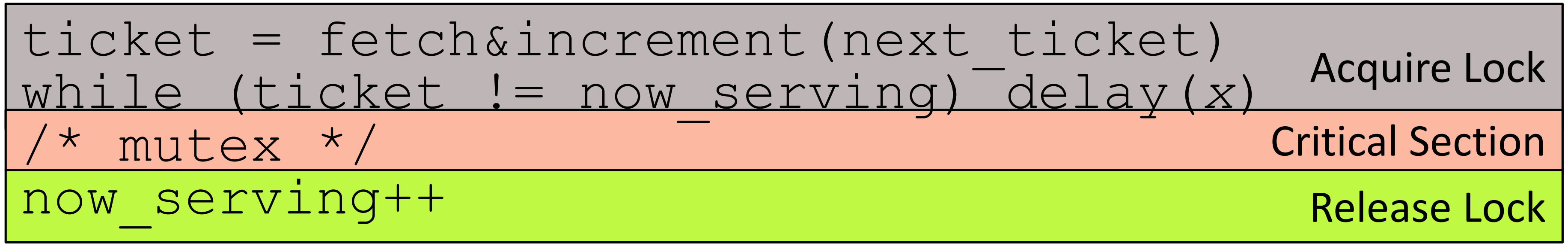
- a lock release causes all waiters to try to get the lock, using a test&set to try to get it.

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)
/* mutex */
now_serving++
```

What delay to use?

Not exponential! Why?

Instead: `ticket - now_serving`

+ Guaranteed FIFO order → no starvation

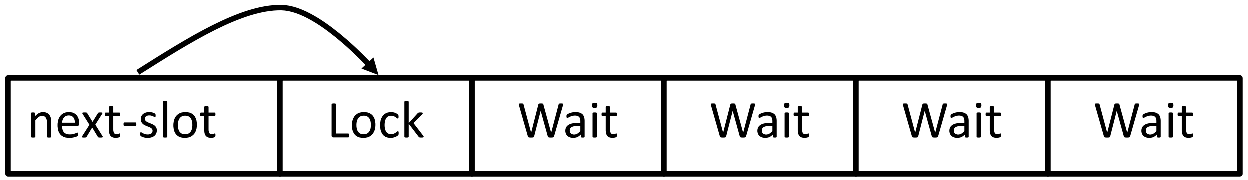
+ Latency can be low (f&i cacheable)

+ 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



next-slot	Lock	Wait	Wait	Wait	Wait

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

Acquire Lock

```
// mutex
```

Critical Section

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

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

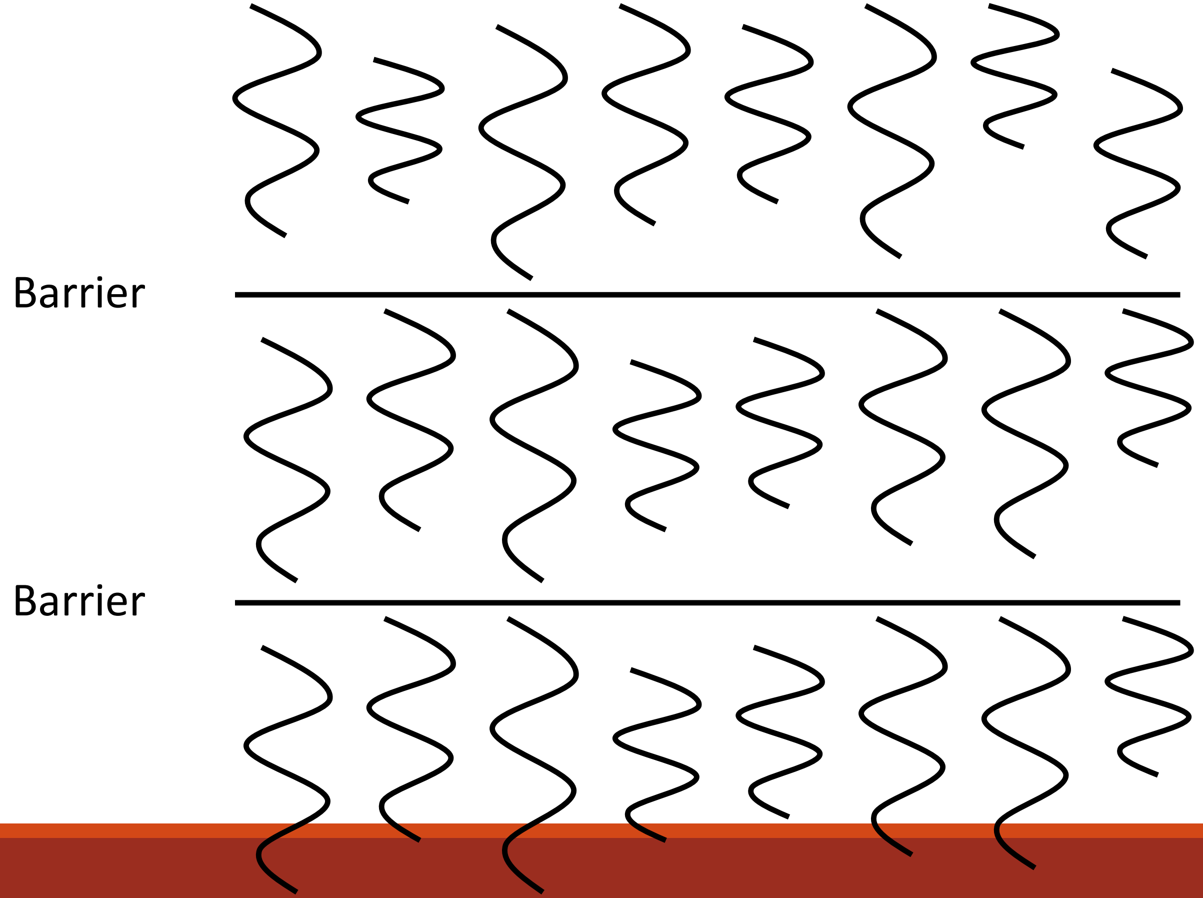
Tradeoffs:

(+) 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, <u>network contention</u> 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?

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

- Maybe OK on broadcast-based coherent systems, spinning traffic on non-coherent or directory-based systems can be unacceptable

$O(P)$ operations on critical path

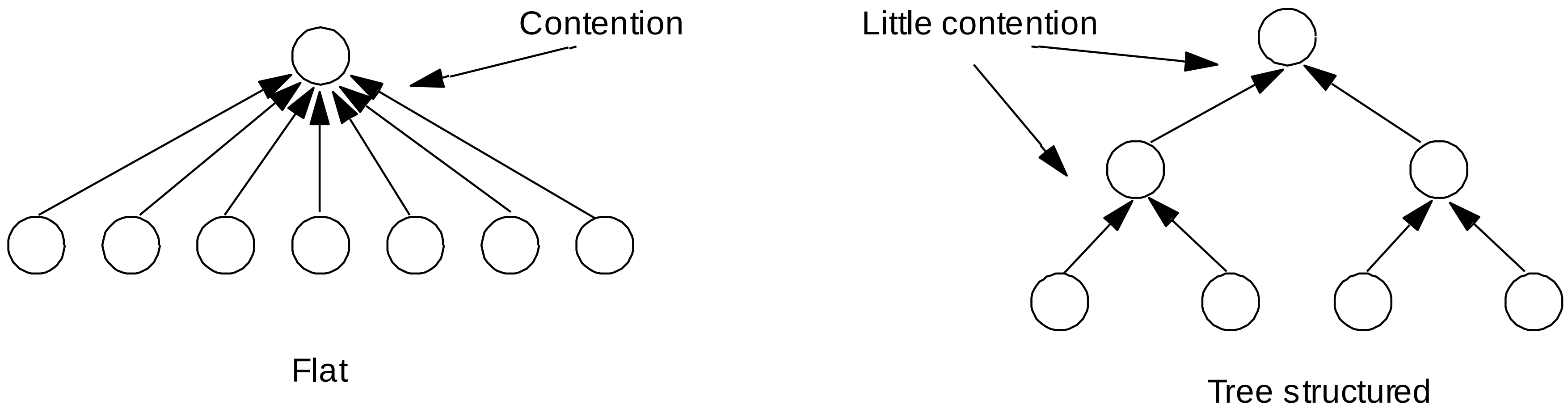
$O(1)$ space

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

Atomic fetch&increment

How about exponential backoff?

Software Combining-Tree Barrier



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 $\boldsymbol{k} \log_{\boldsymbol{k}} \boldsymbol{P}$

Critical path is minimized by choosing $\boldsymbol{k} = \mathbf{2}$

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 $(i + 2^k) \bmod P$

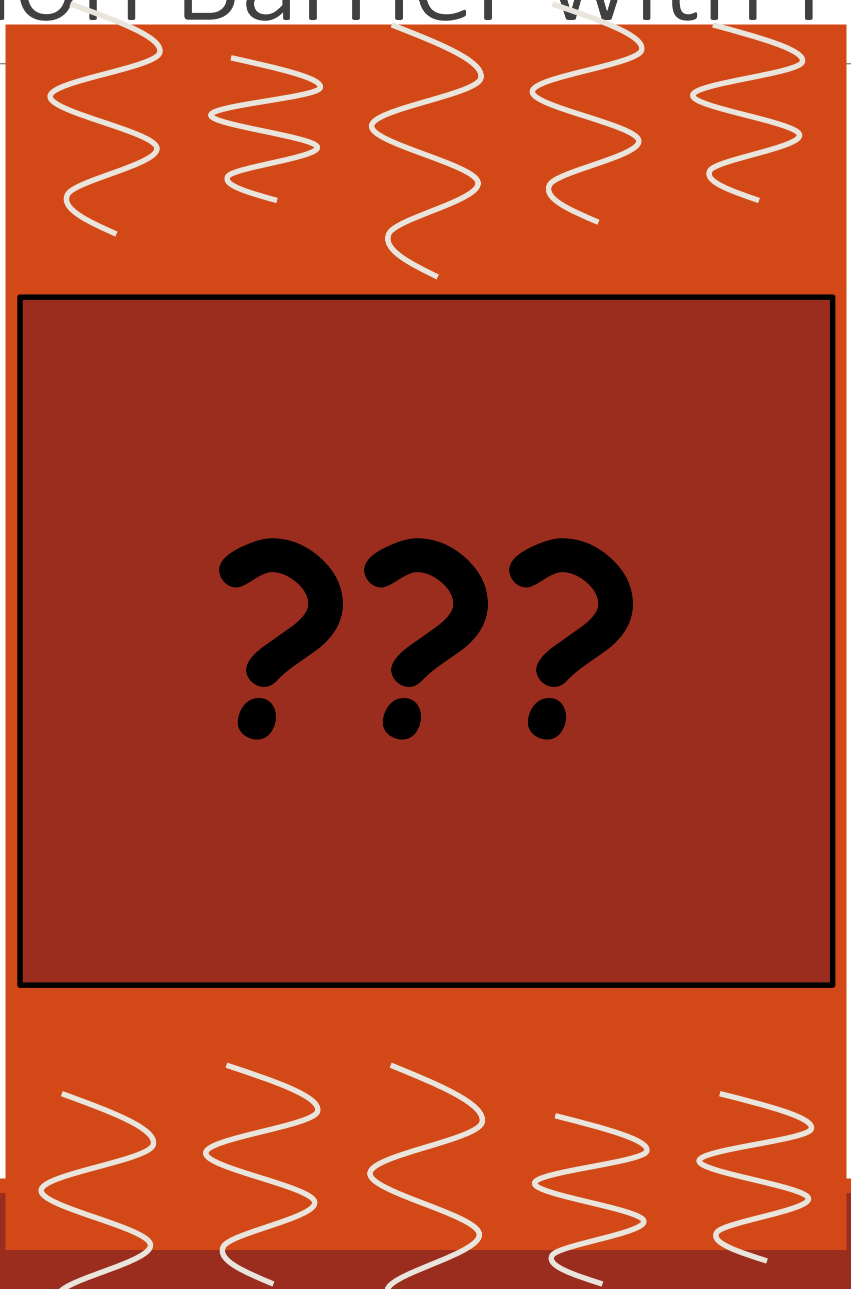
Threads signal each other by writing flags

- 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

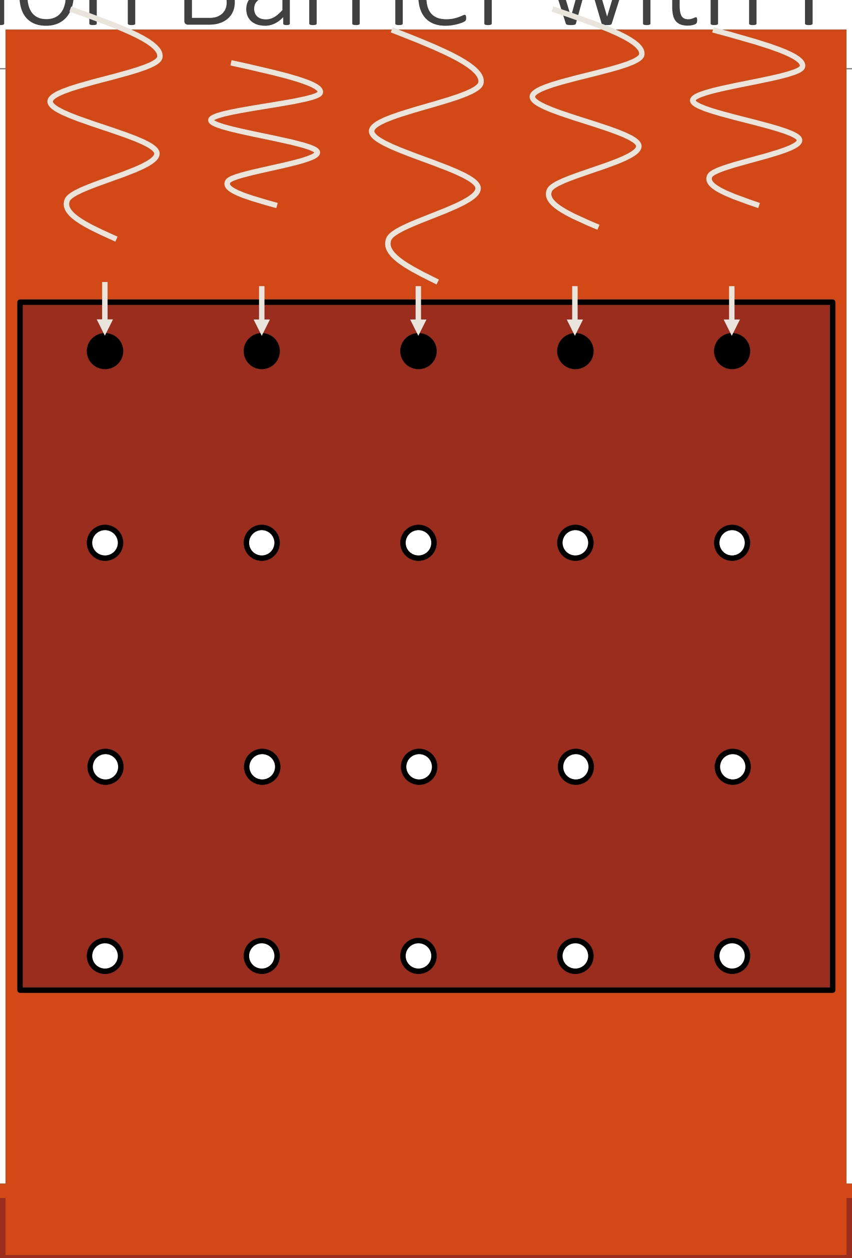


Barrier

Dissemination Barrier with P=5

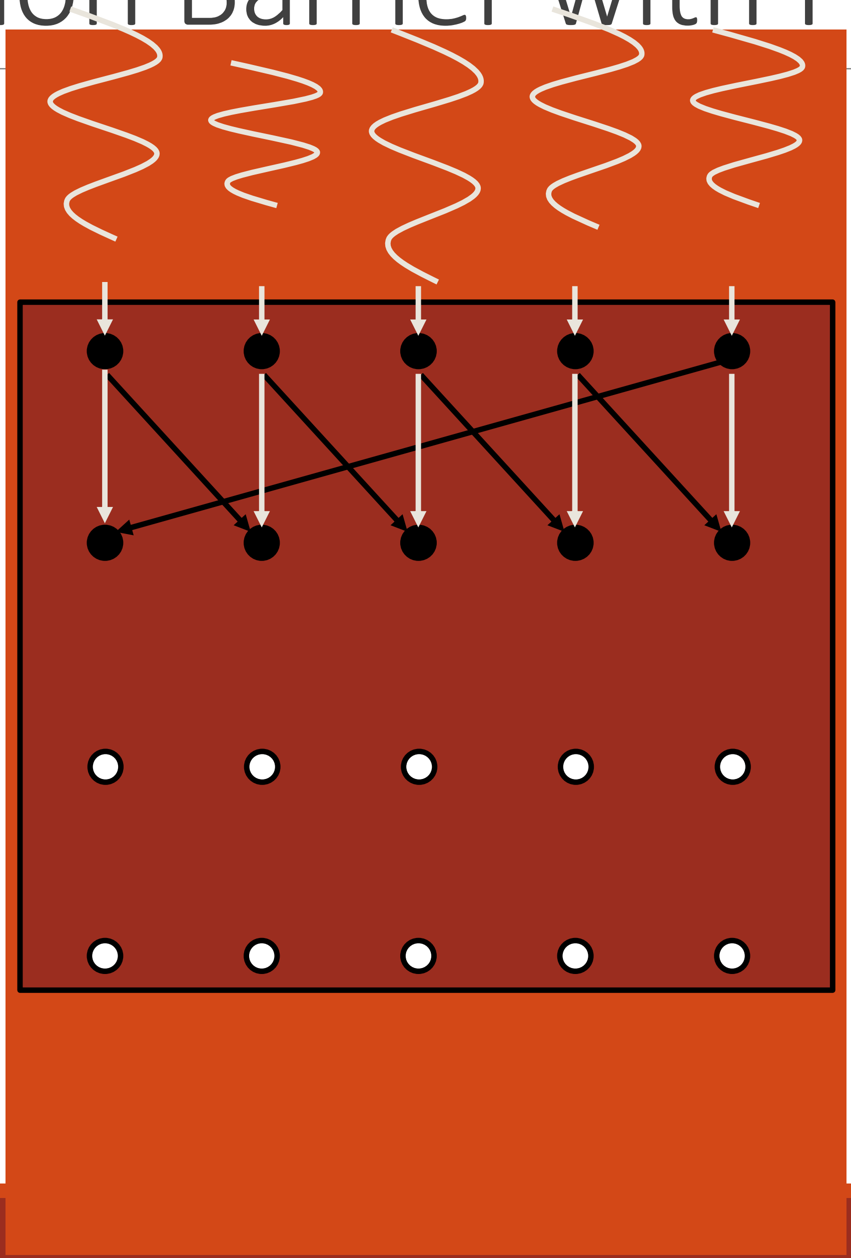
$3 = \log_2 P$ rounds

Barrier



Dissemination Barrier with P=5

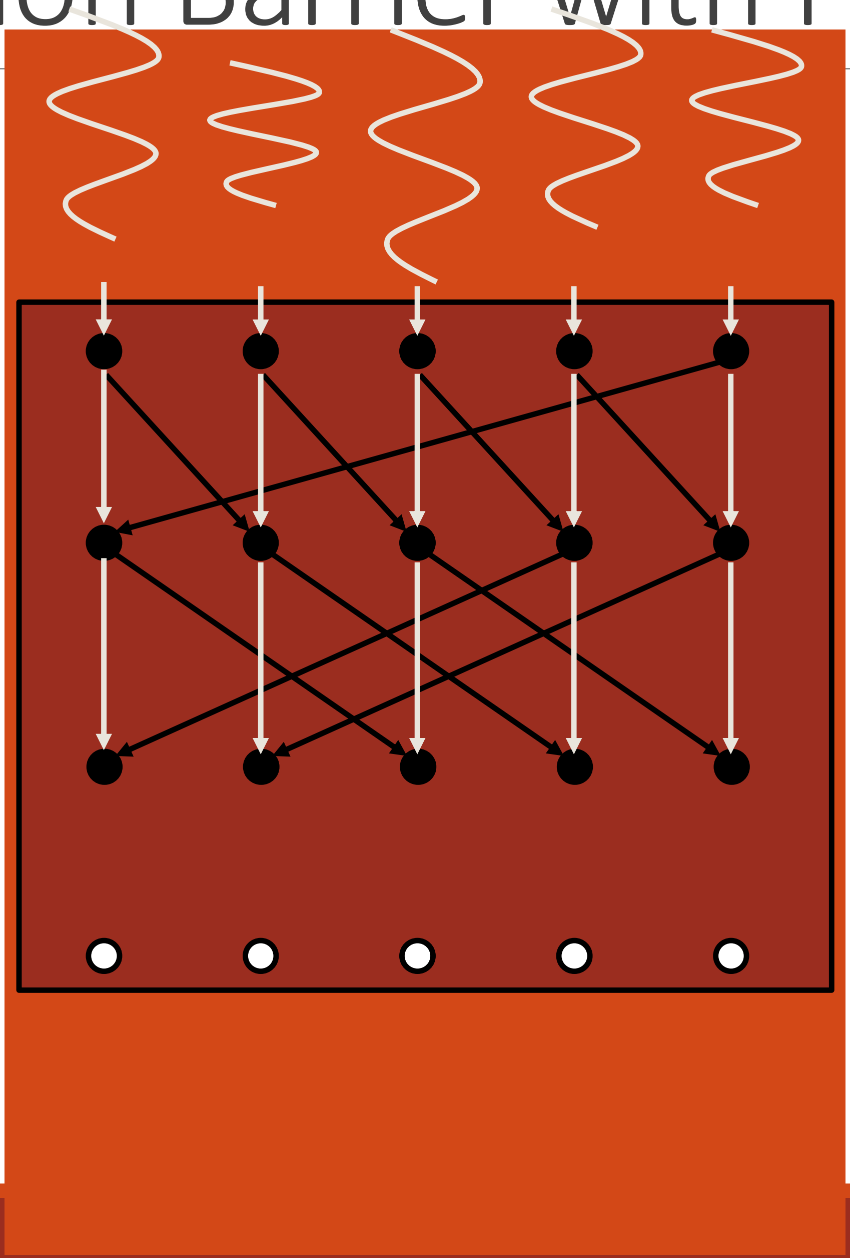
$3 = \log_2 P$ rounds



Round 1: offset $2^0 = 1$

Barrier

Dissemination Barrier with P=5



$3 = \log_2 P$ rounds

Barrier

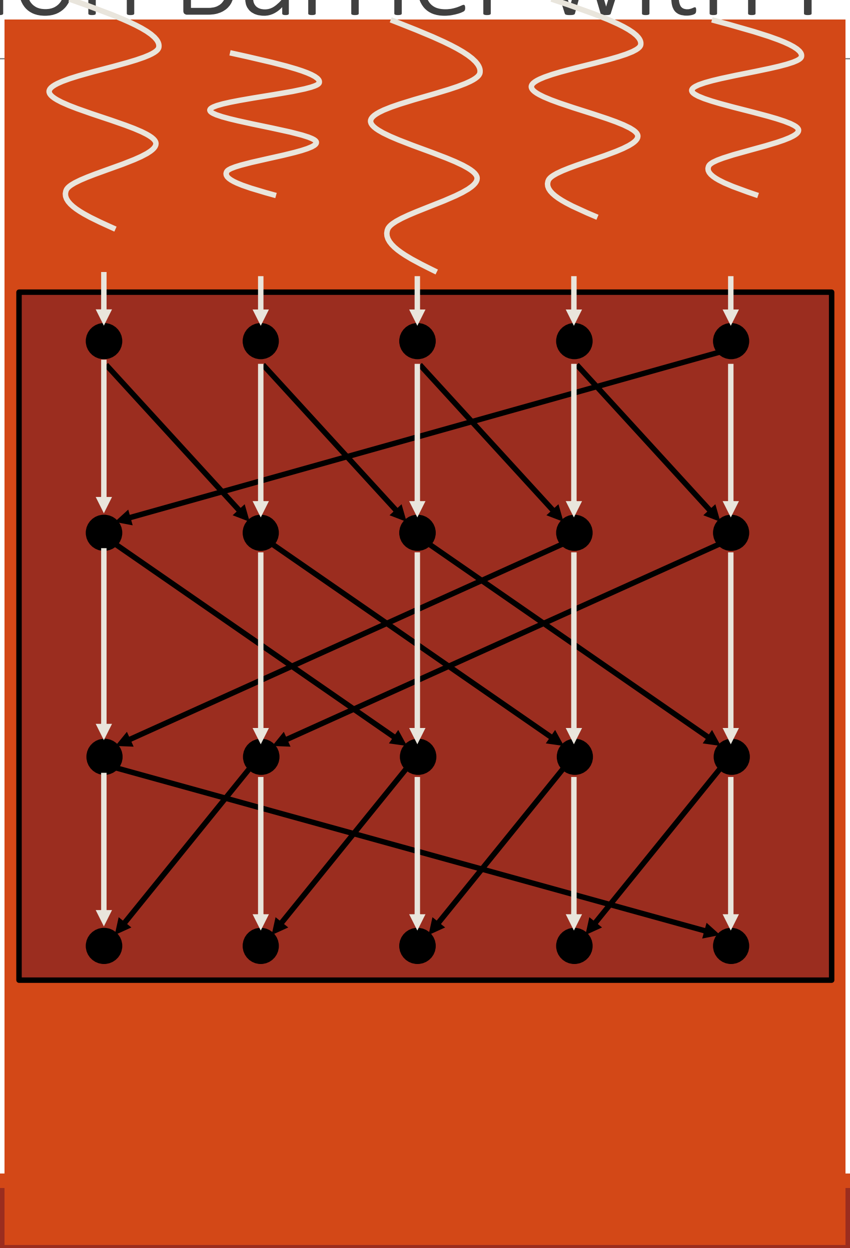
Round 1: offset $2^0 = 1$

Round 2: offset $2^1 = 2$

Dissemination Barrier with P=5

$3 = \log_2 P$ rounds

Barrier



Round 1: offset $2^0 = 1$

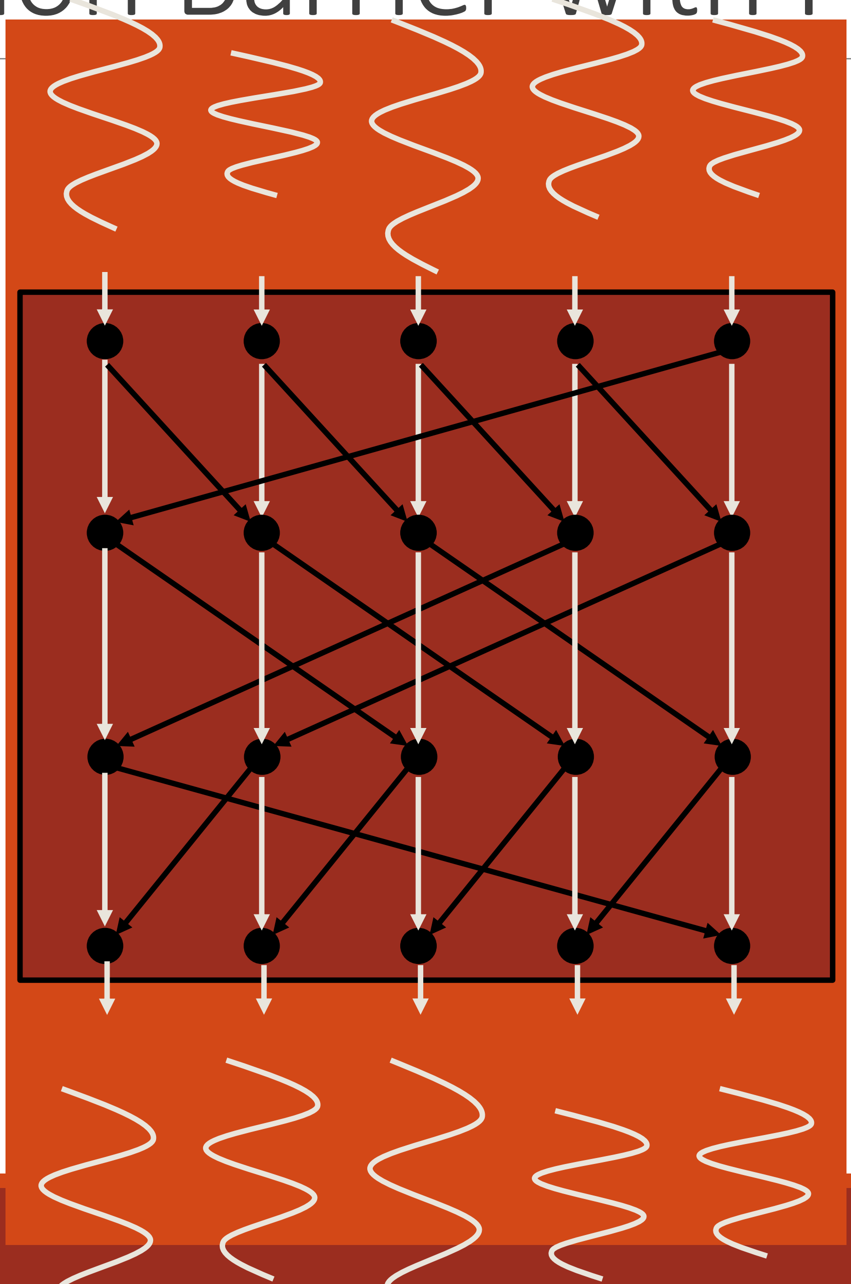
Round 2: offset $2^1 = 2$

Round 3: offset $2^2 = 4$

Dissemination Barrier with P=5

$3 = \log_2 P$ rounds

Barrier



Round 1: offset $2^0 = 1$

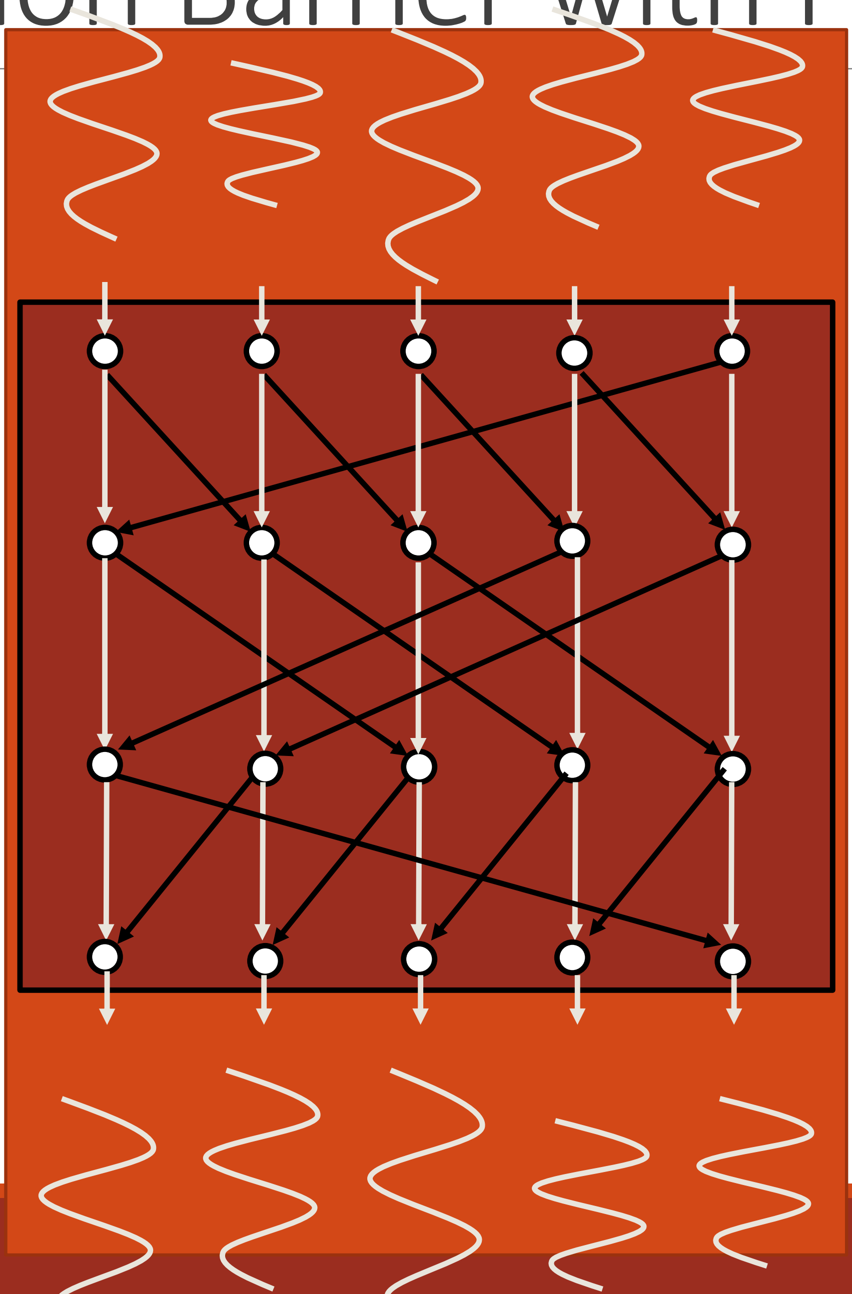
Round 2: offset $2^1 = 2$

Round 3: offset $2^2 = 4$

Dissemination Barrier with P=5

Threads can progress unevenly through barrier

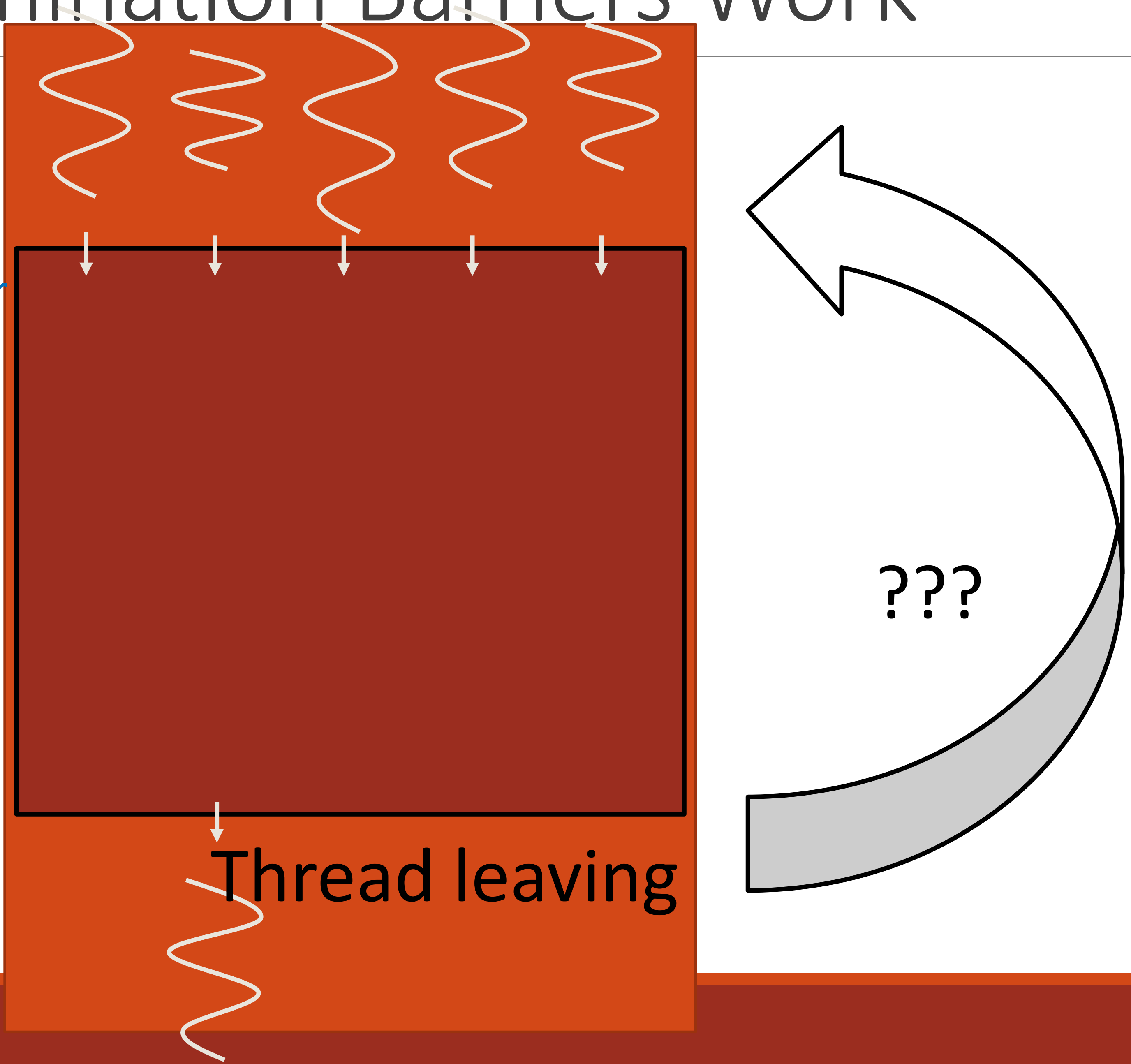
But none will exit until all arrive



Why Dissemination Barriers Work

Prove that:

Any thread leaves barrier
➔
All threads entered barrier



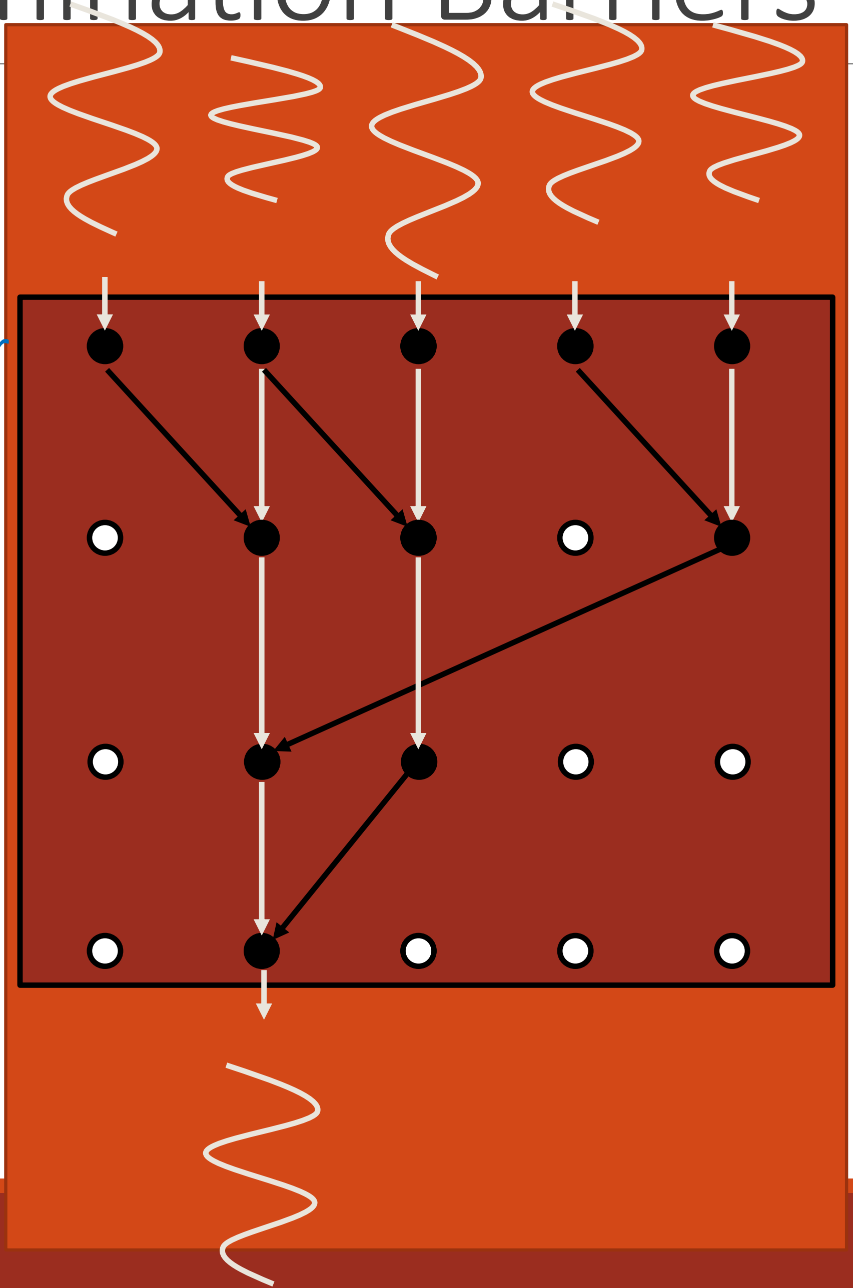
Why Dissemination Barriers Work

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

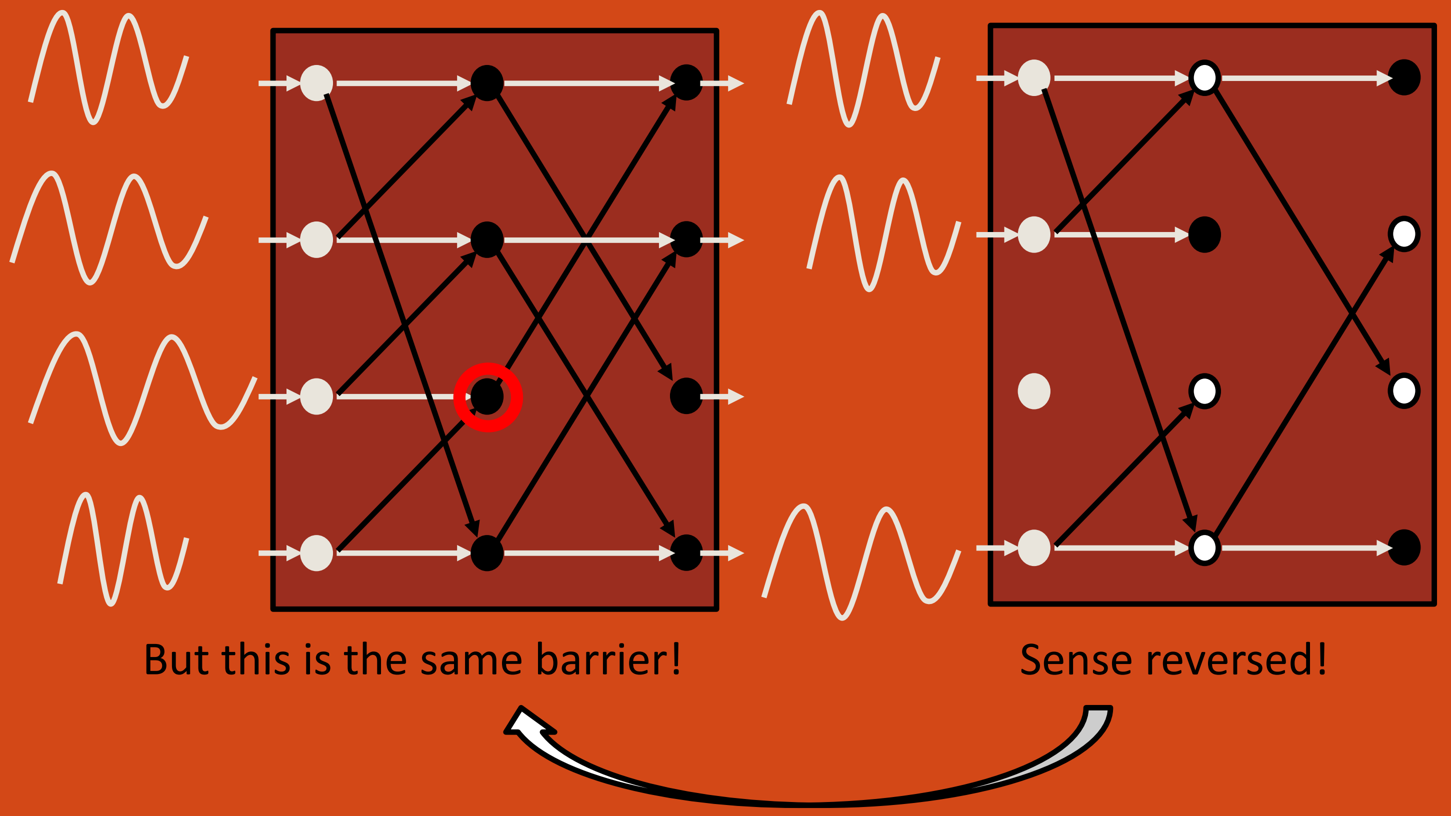
```
const int rounds = log(P);
bool flags[P][rounds]; // allocated in local storage per thread
local bool sense = false;

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

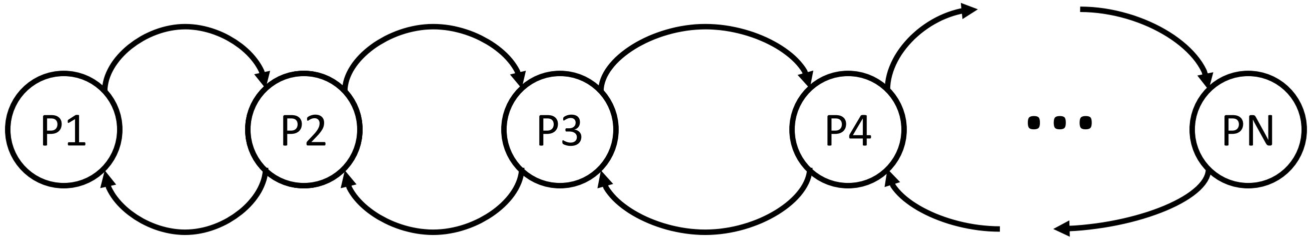
$O(P \log P)$ space – $\log P$ variables per processor

$O(P \log P)$ 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

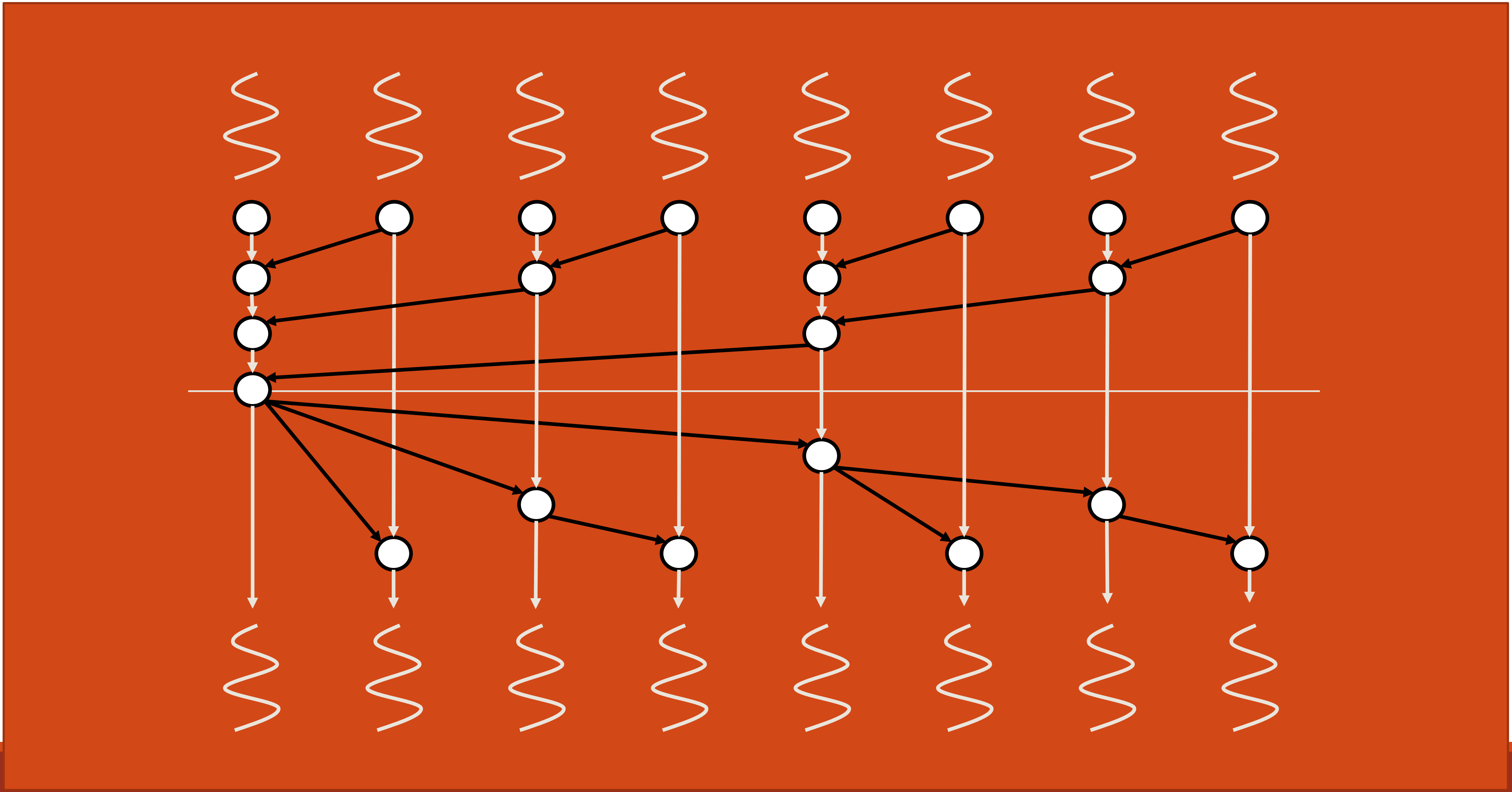
Representative processor at a node is statically chosen

- No fetch&op needed

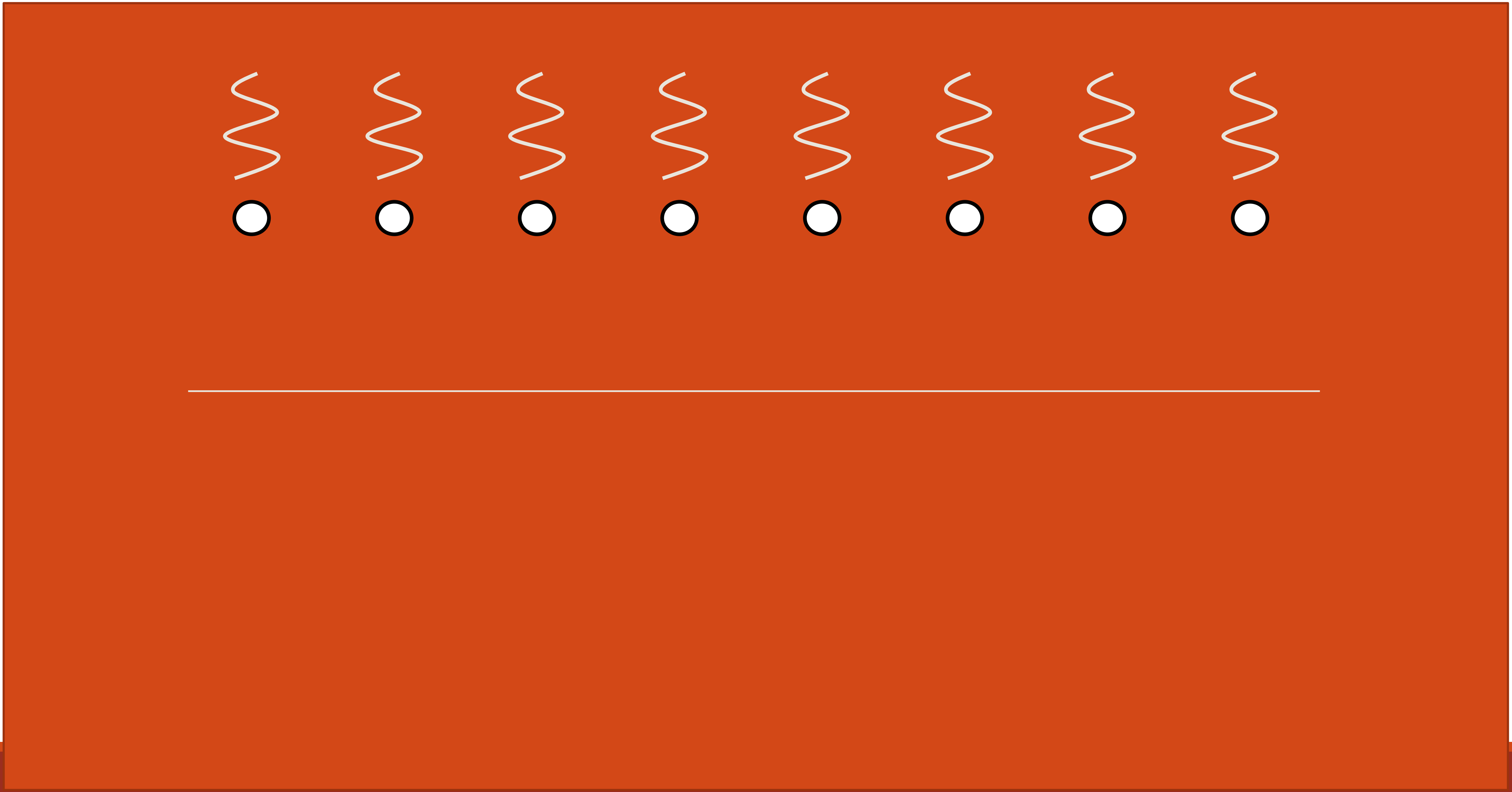
In round k, proc $i = 2^k$ sets a flag for proc $j = i - 2^k$

- 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

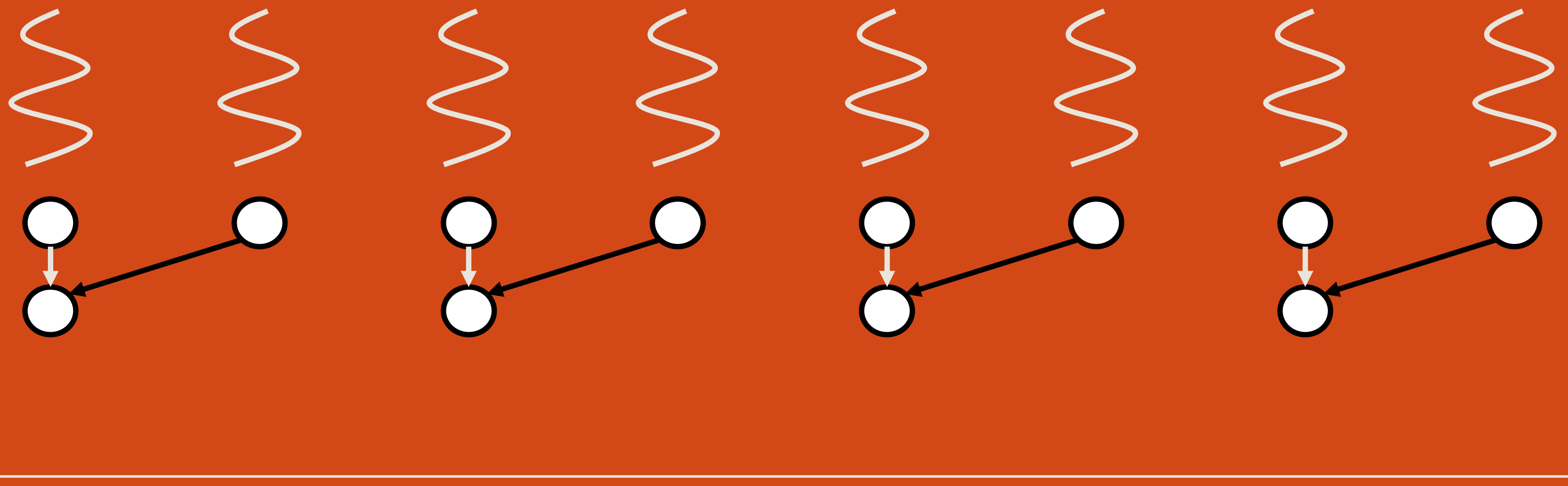
Tournament Barrier with P=8



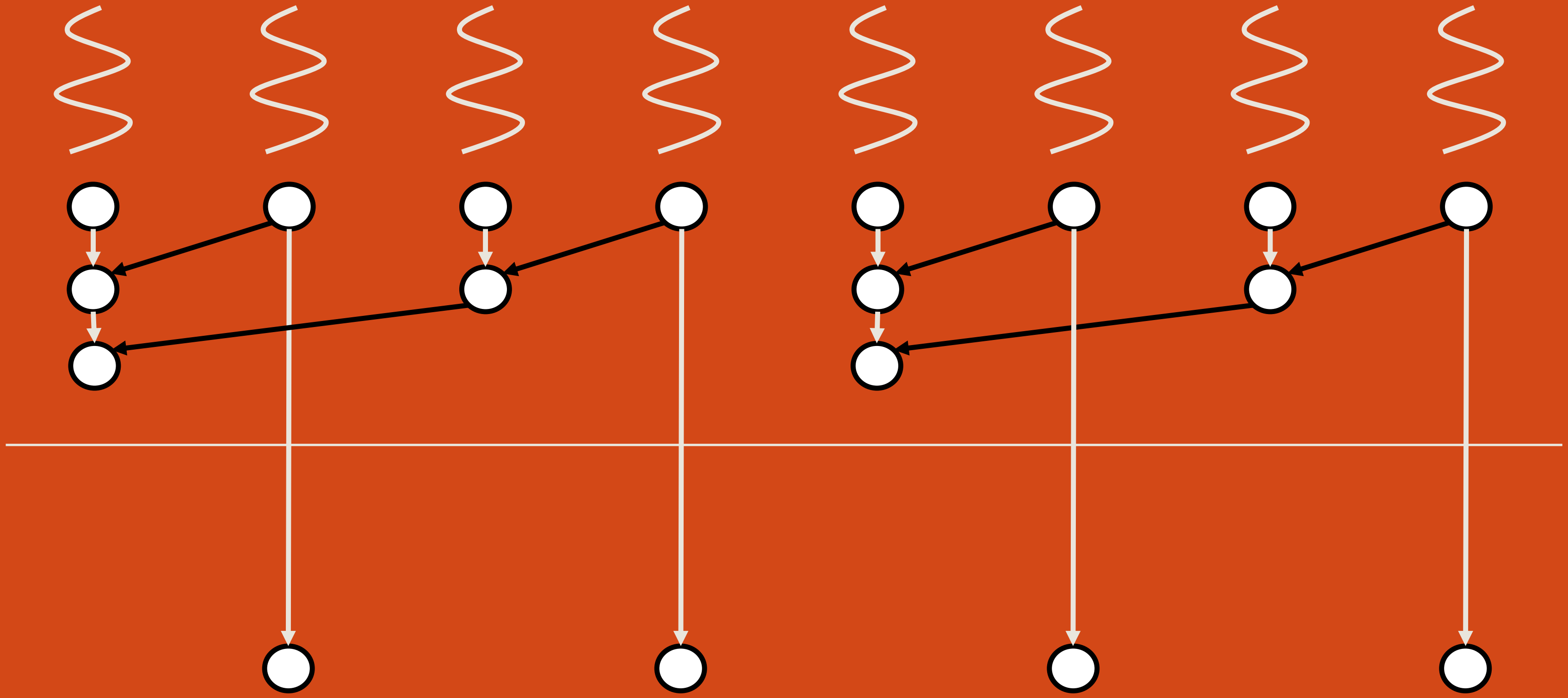
Tournament Barrier with P=8



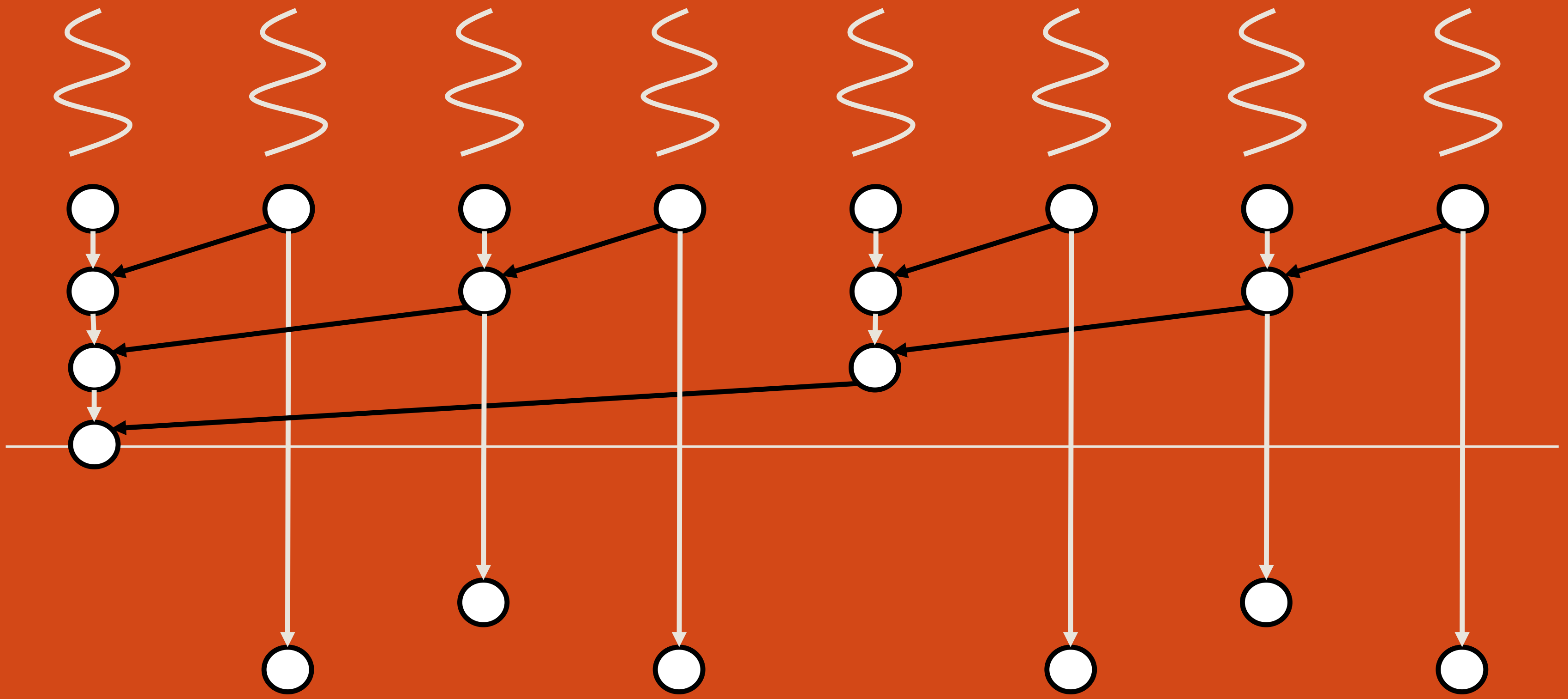
Tournament Barrier with P=8



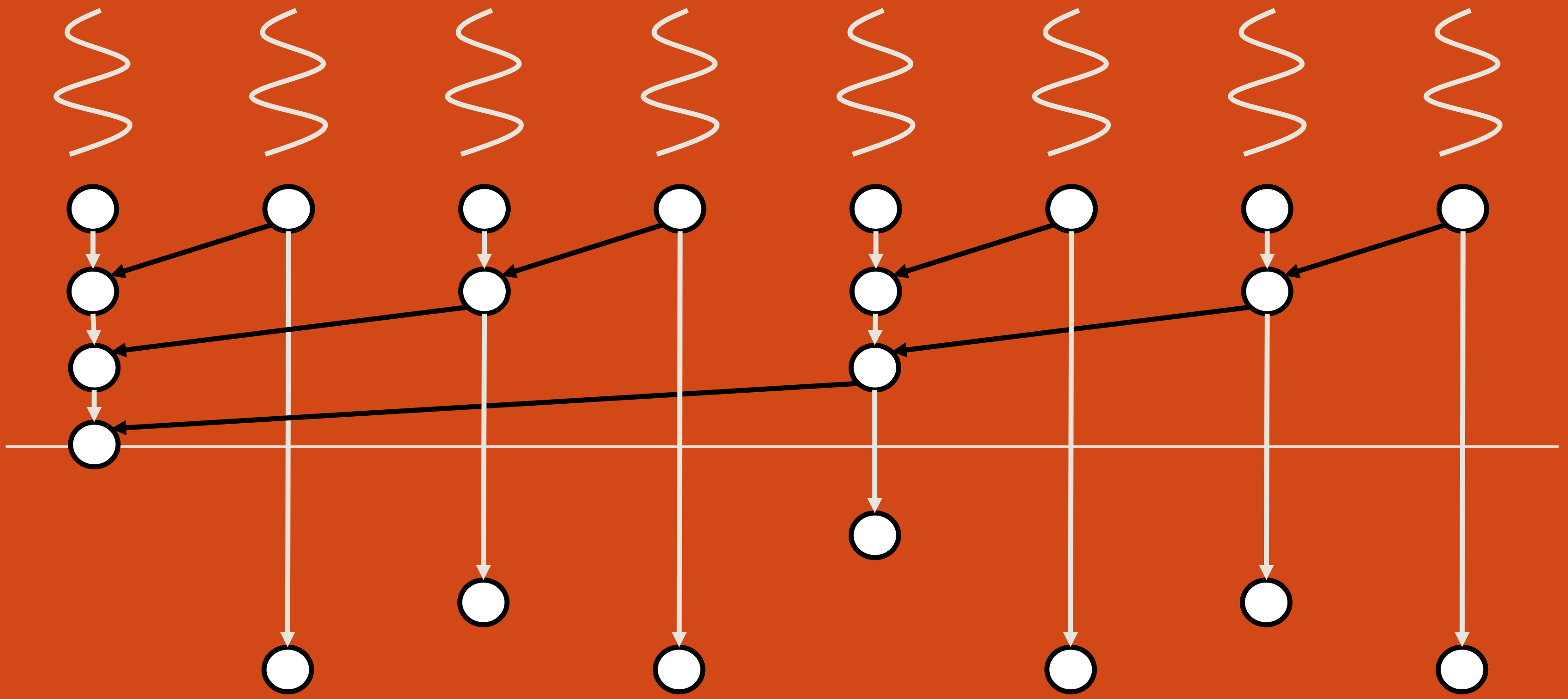
Tournament Barrier with P=8



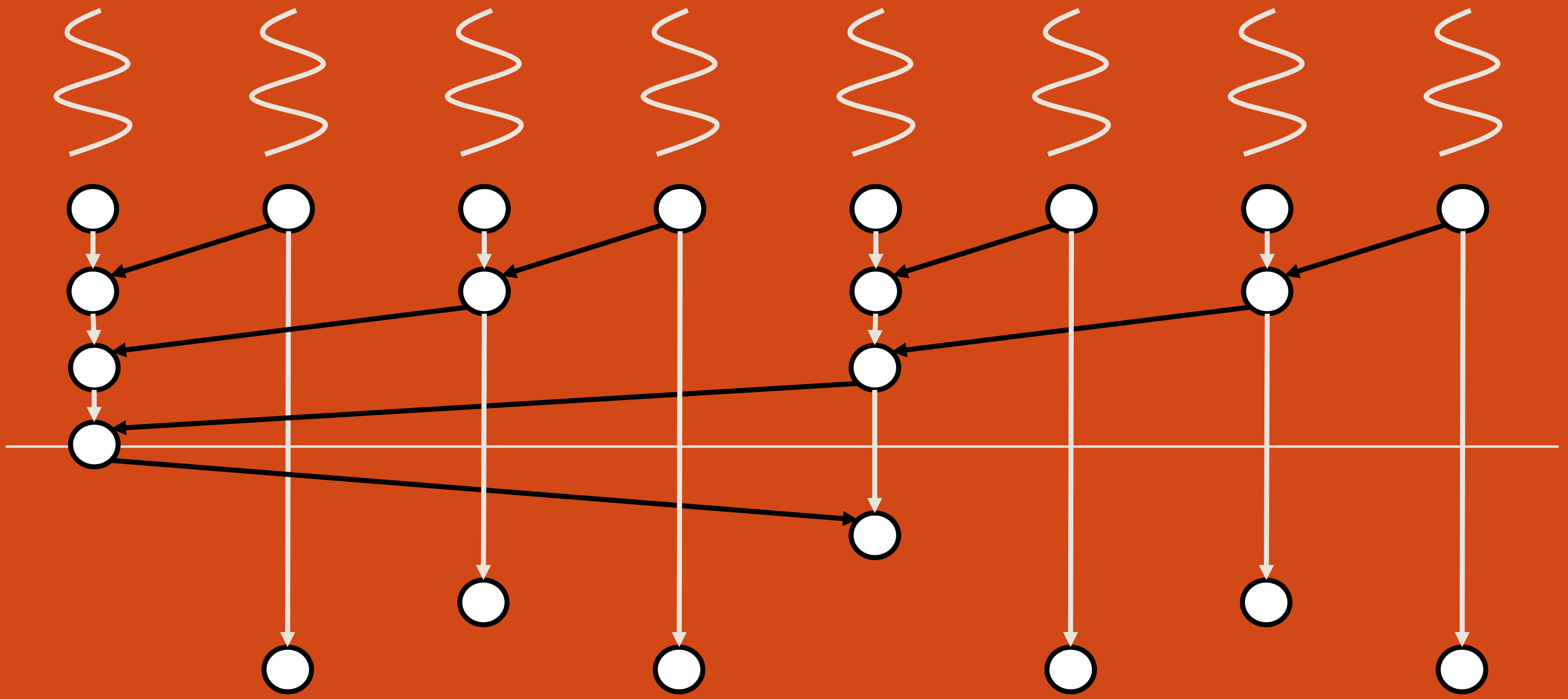
Tournament Barrier with P=8



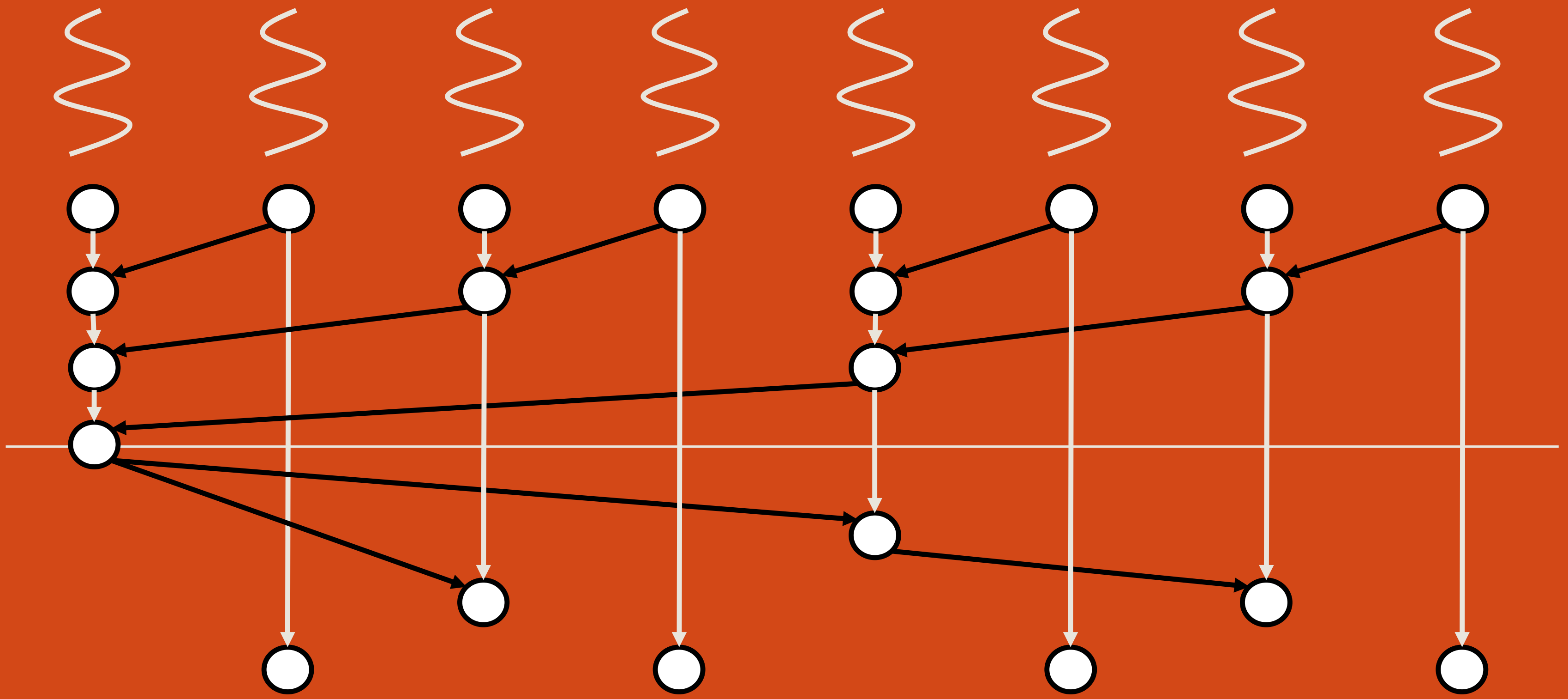
Tournament Barrier with P=8



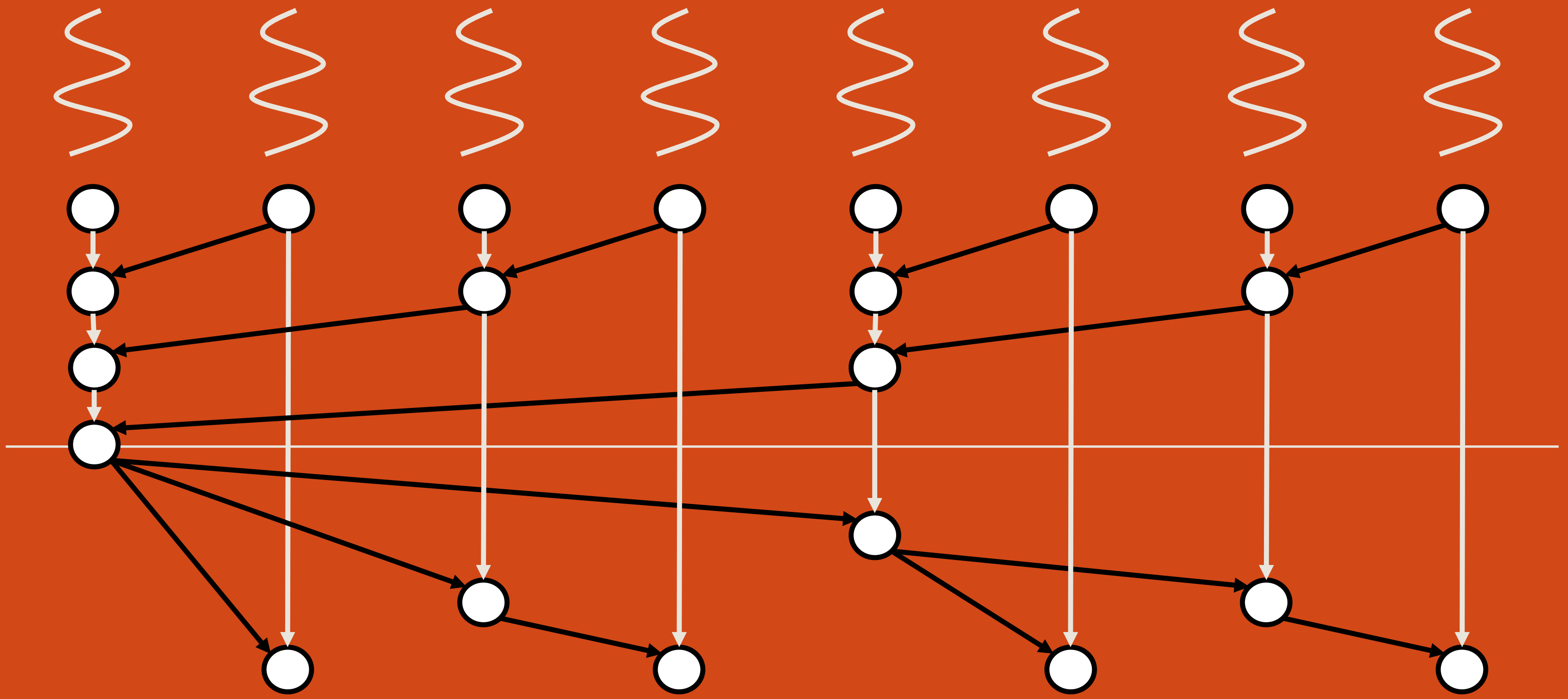
Tournament Barrier with P=8



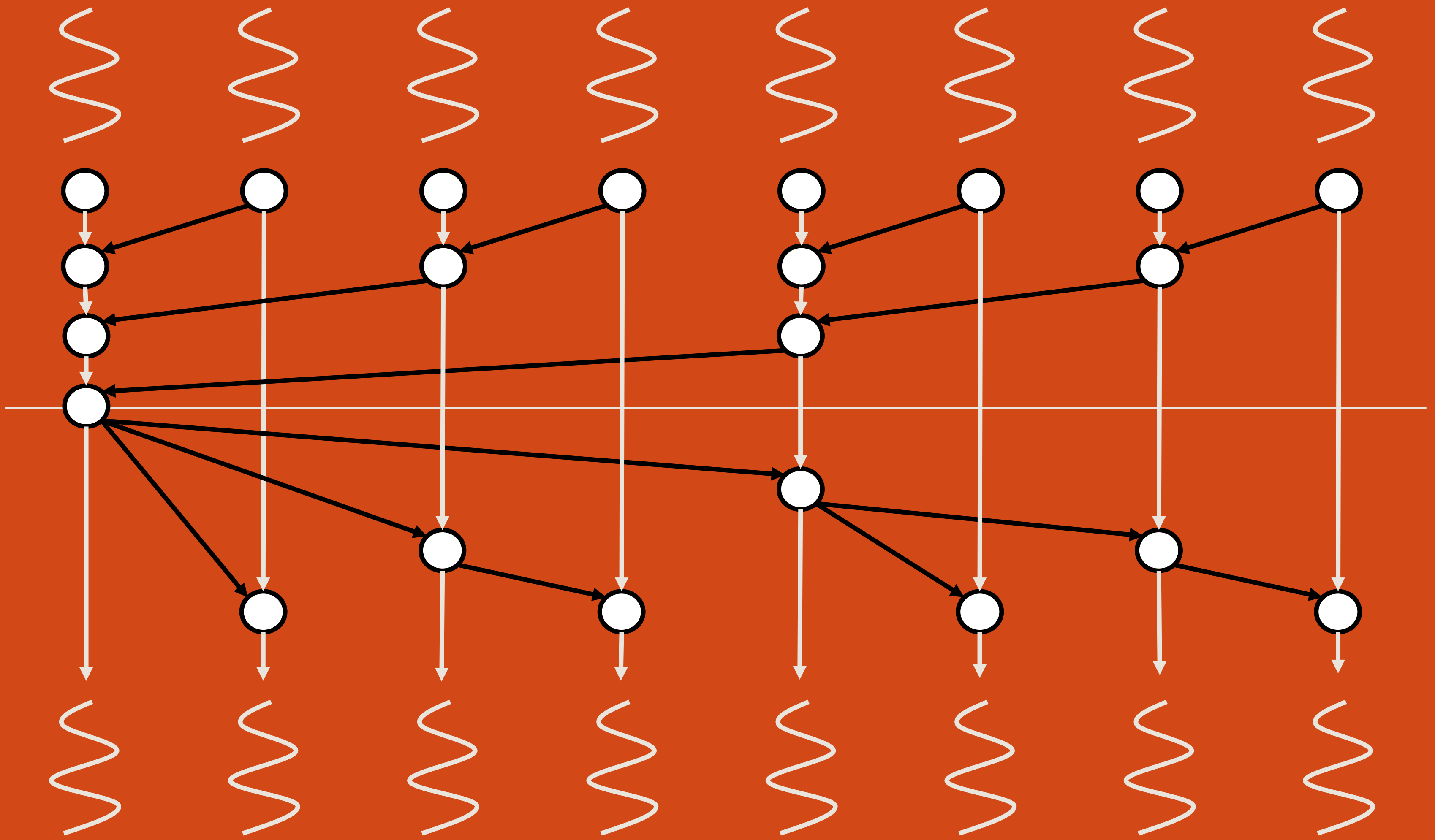
Tournament Barrier with P=8



Tournament Barrier with P=8



Tournament Barrier with P=8



Why Tournament Barrier Works

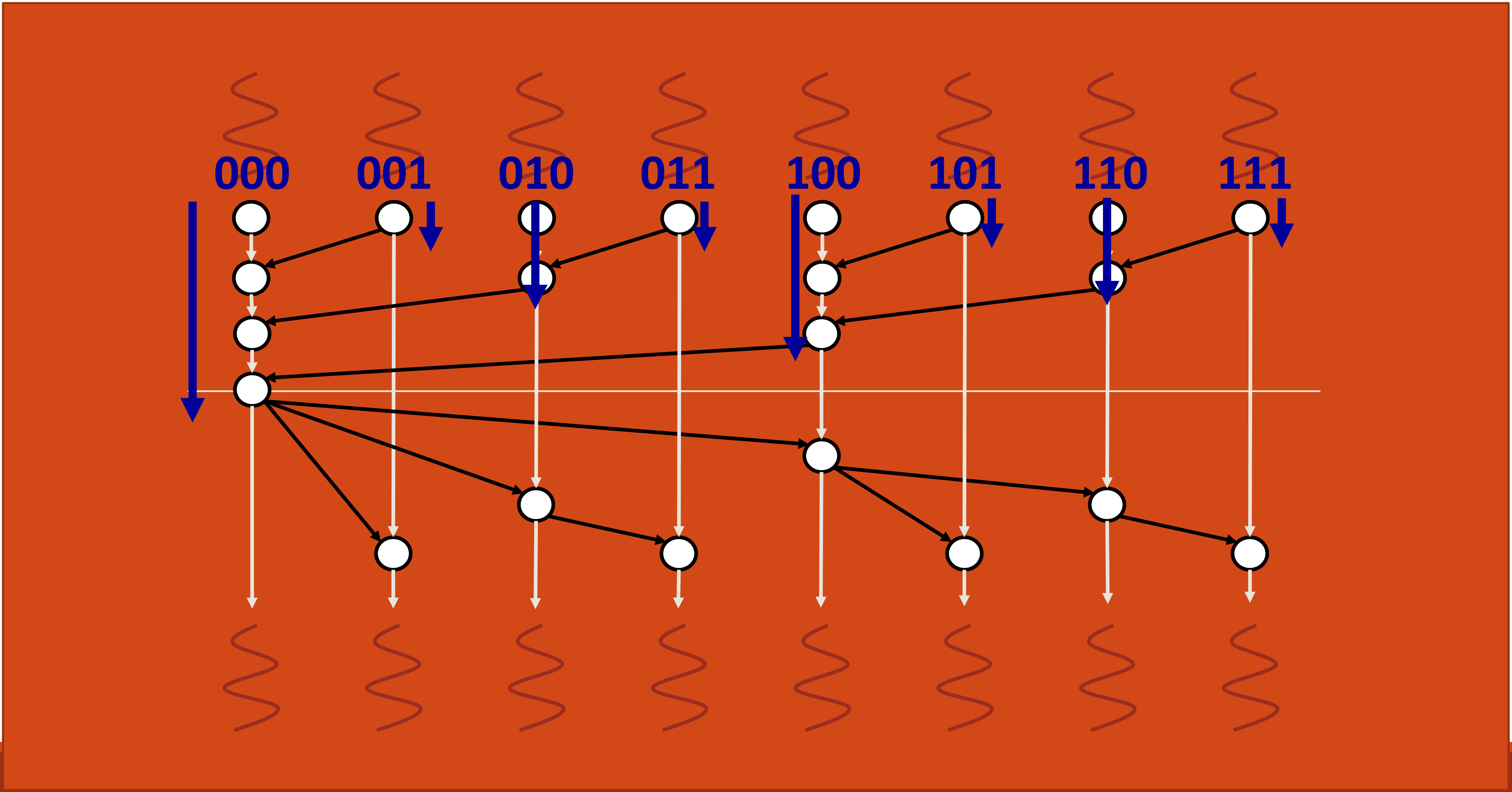
As before, threads can progress at different rates through tree

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

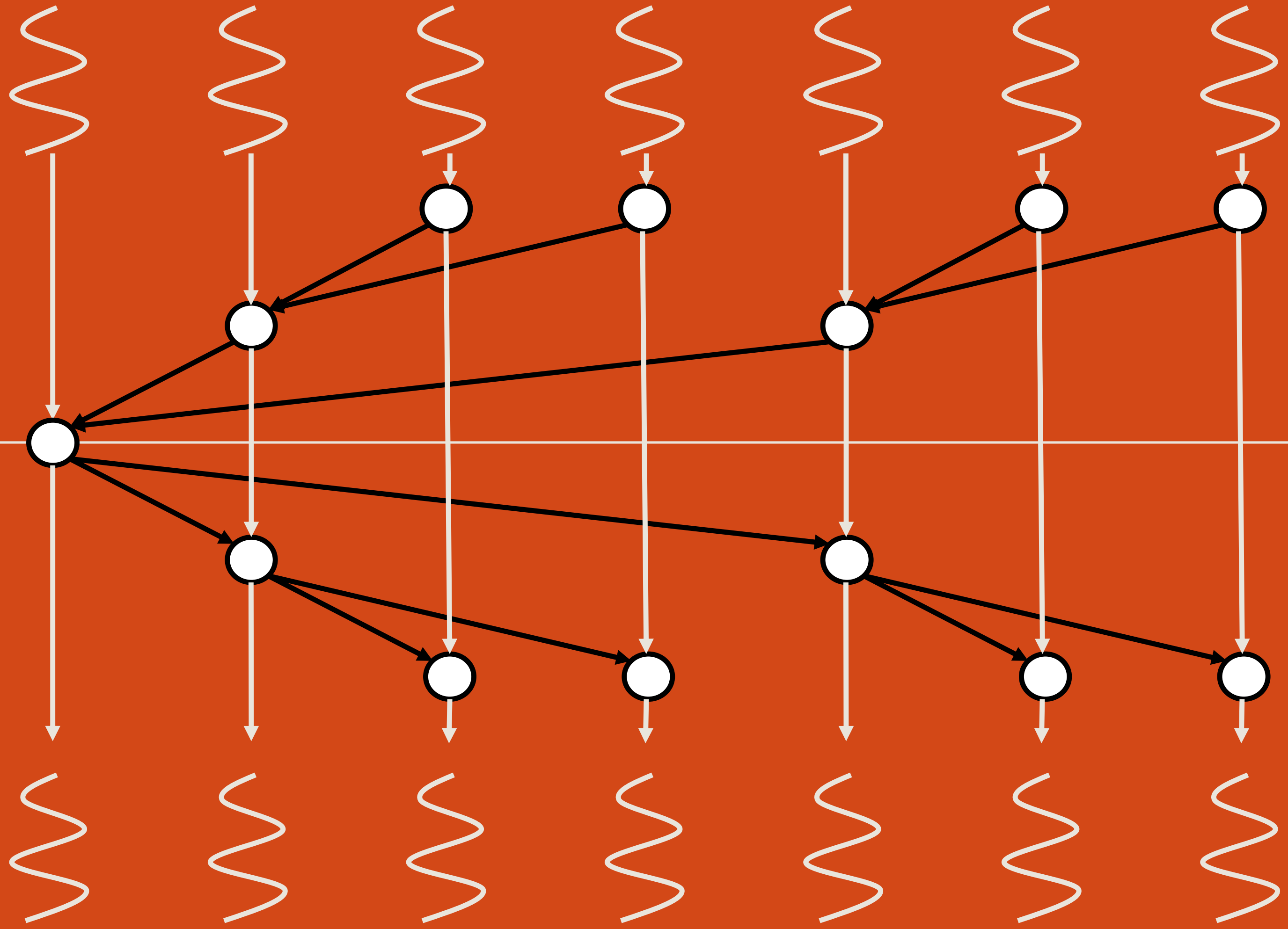
MCS Software Barrier

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

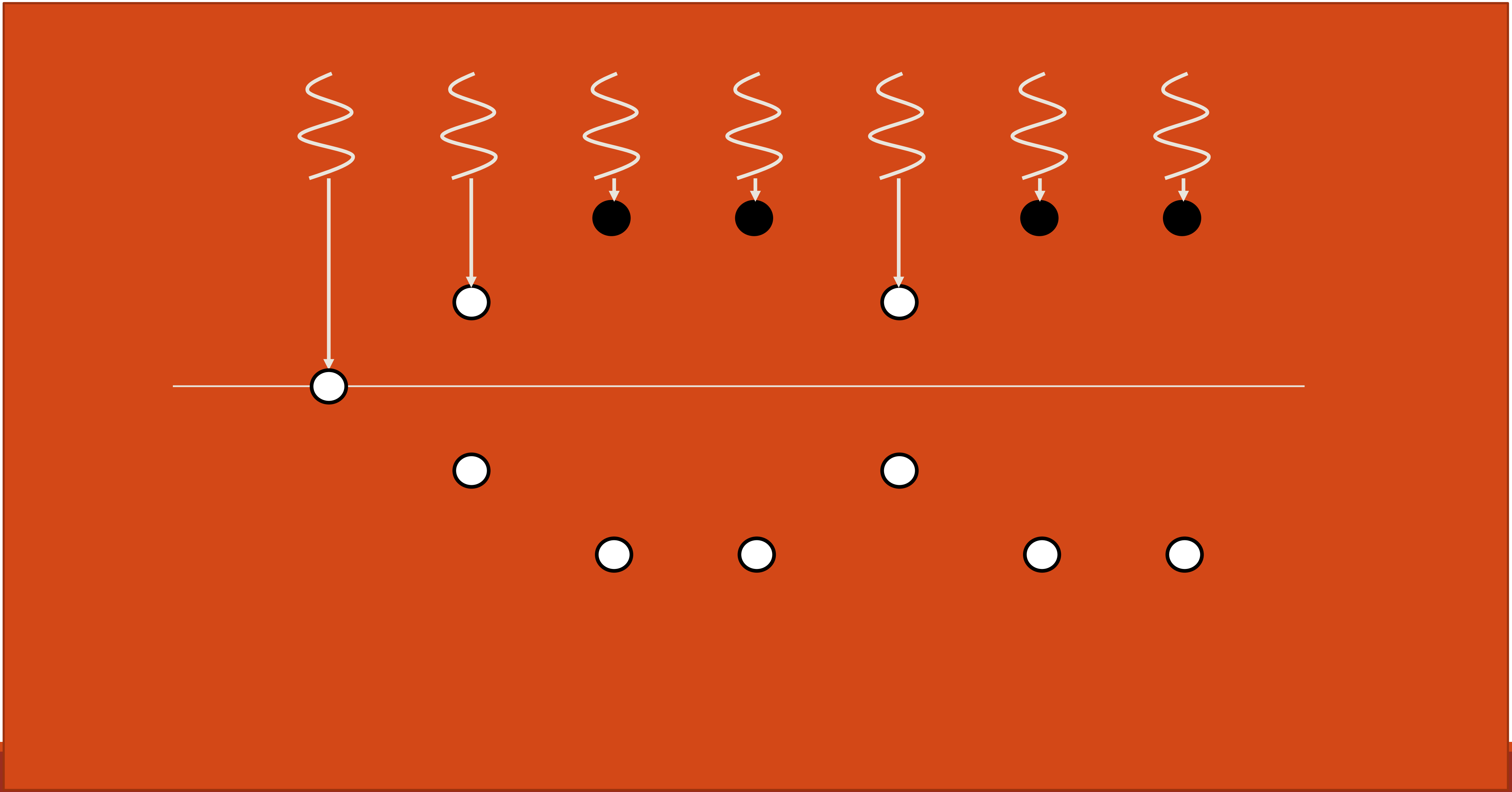
Every thread is a node in two P-node trees:

- 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

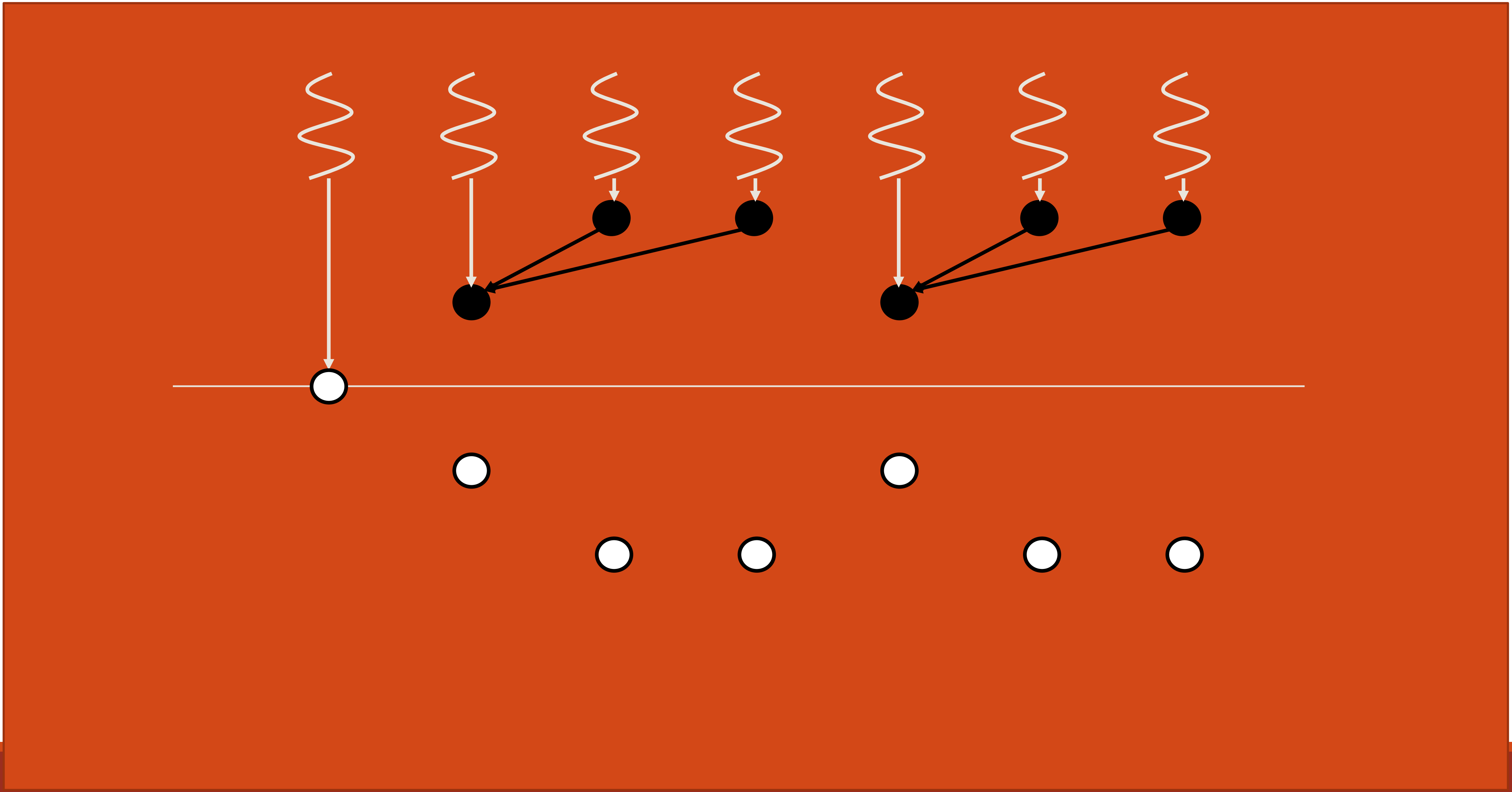
MCS Barrier with P=7



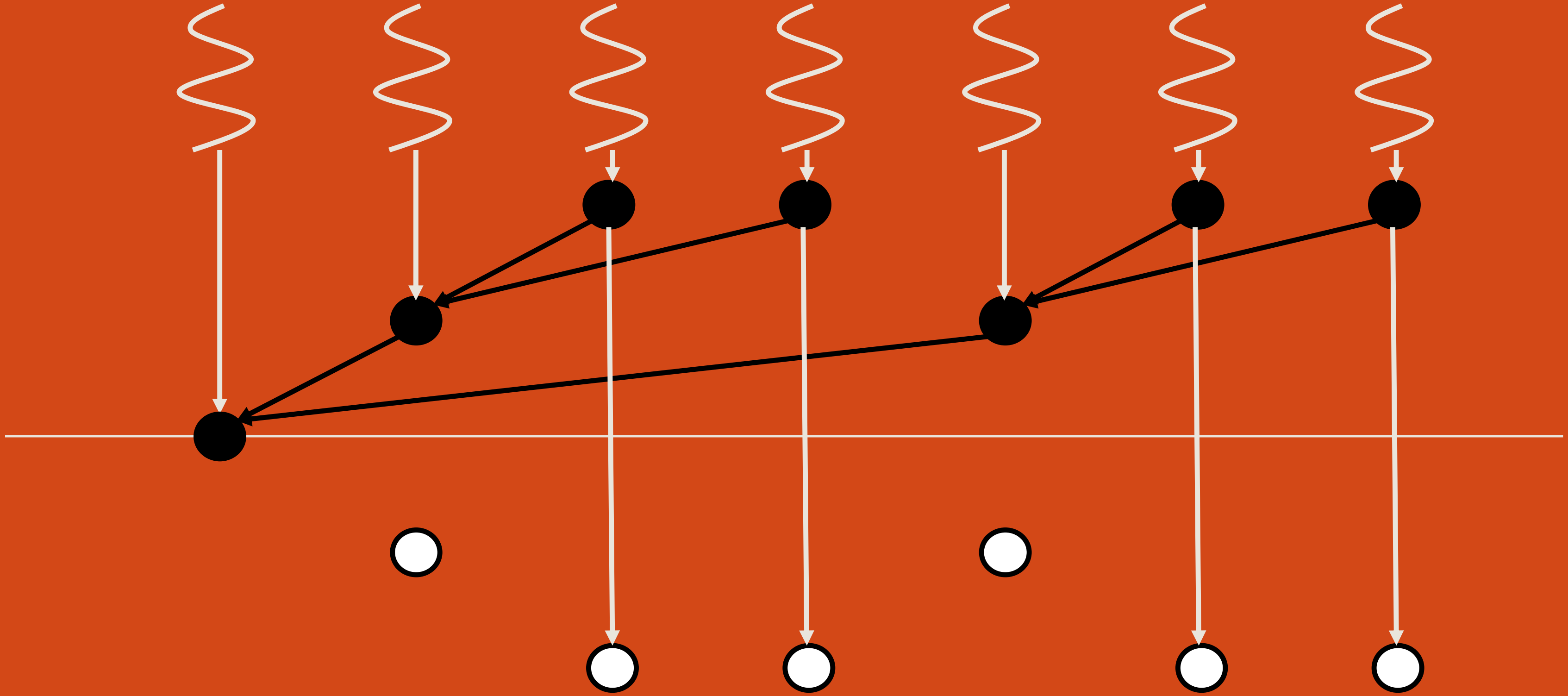
MCS Barrier with P=7



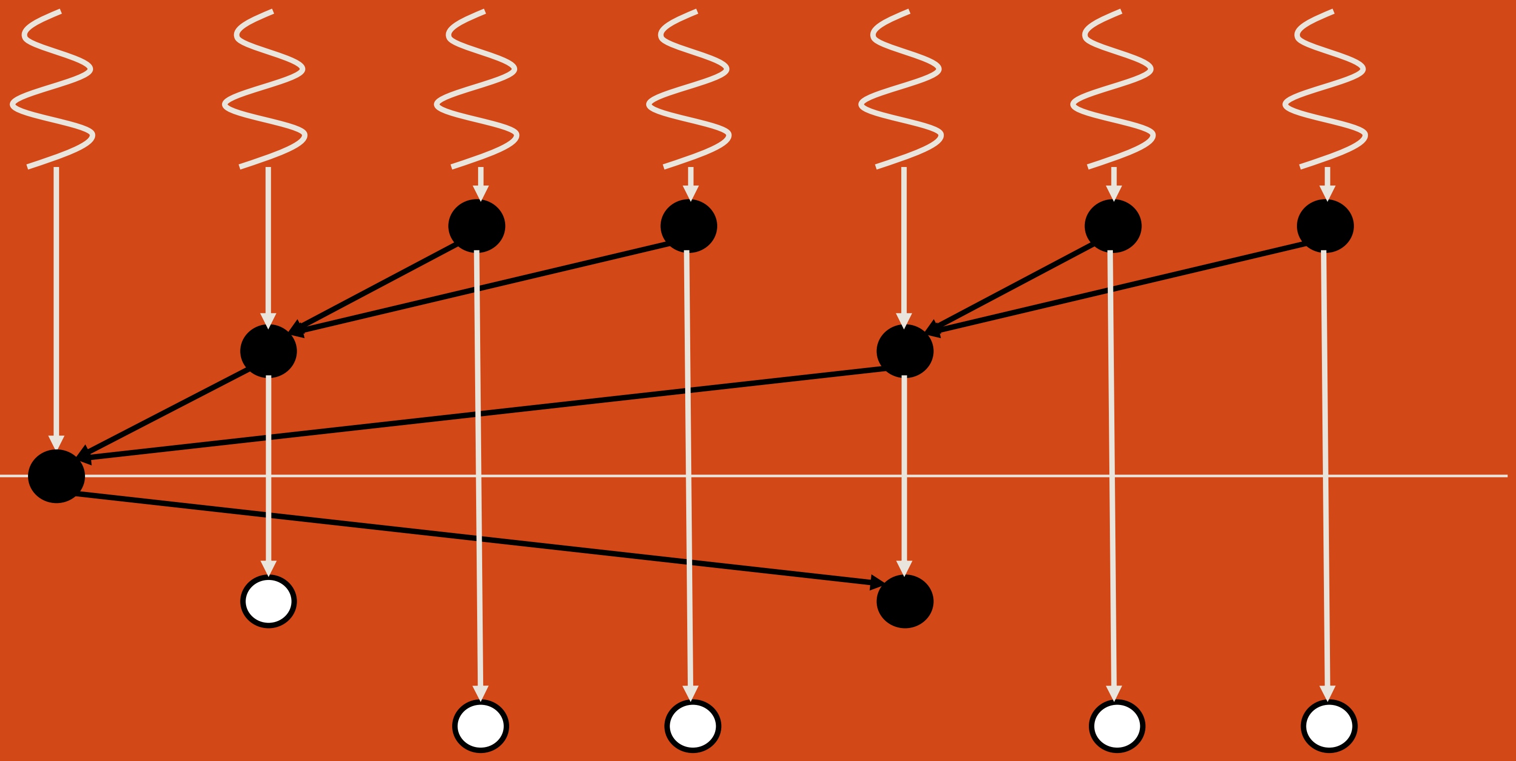
MCS Barrier with P=7



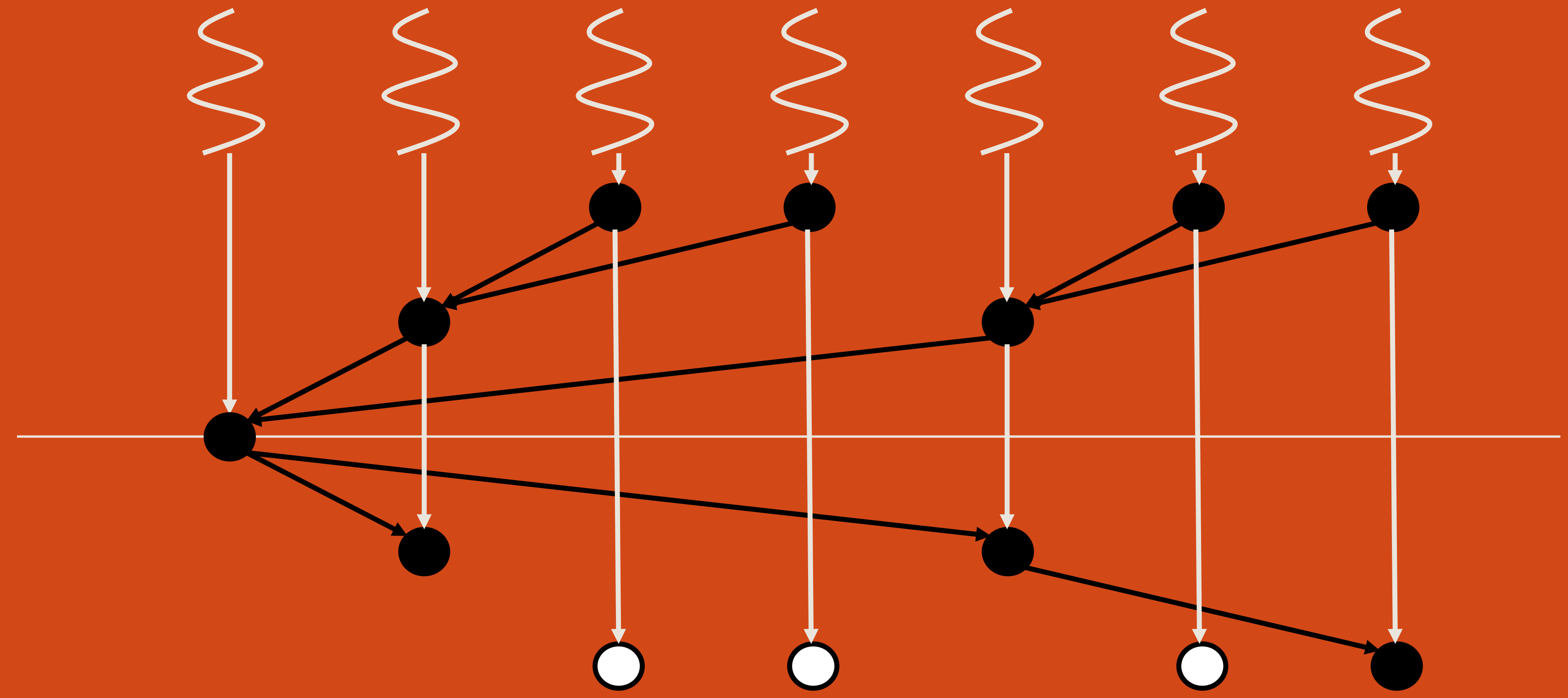
MCS Barrier with P=7



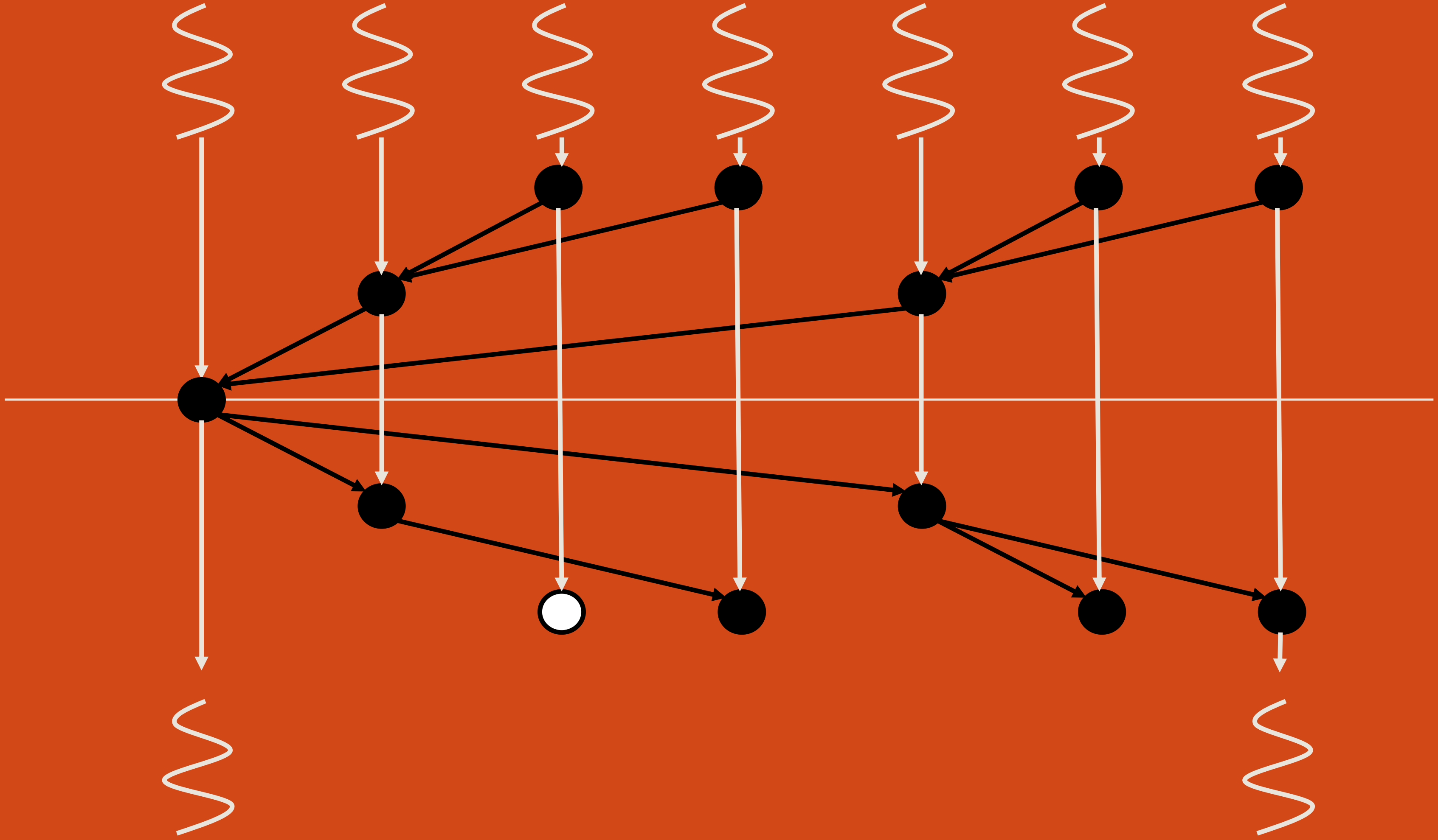
MCS Barrier with P=7



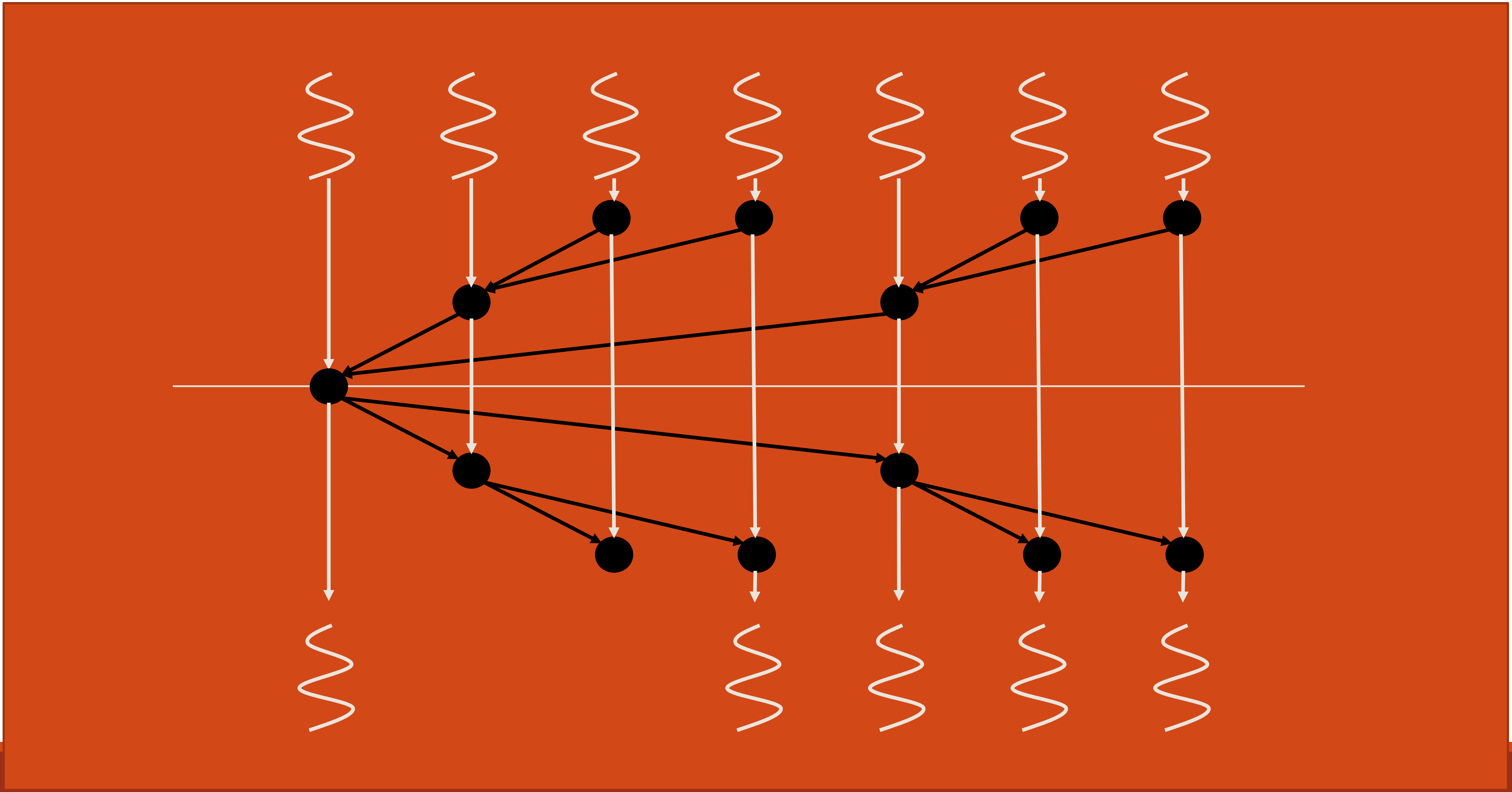
MCS Barrier with P=7



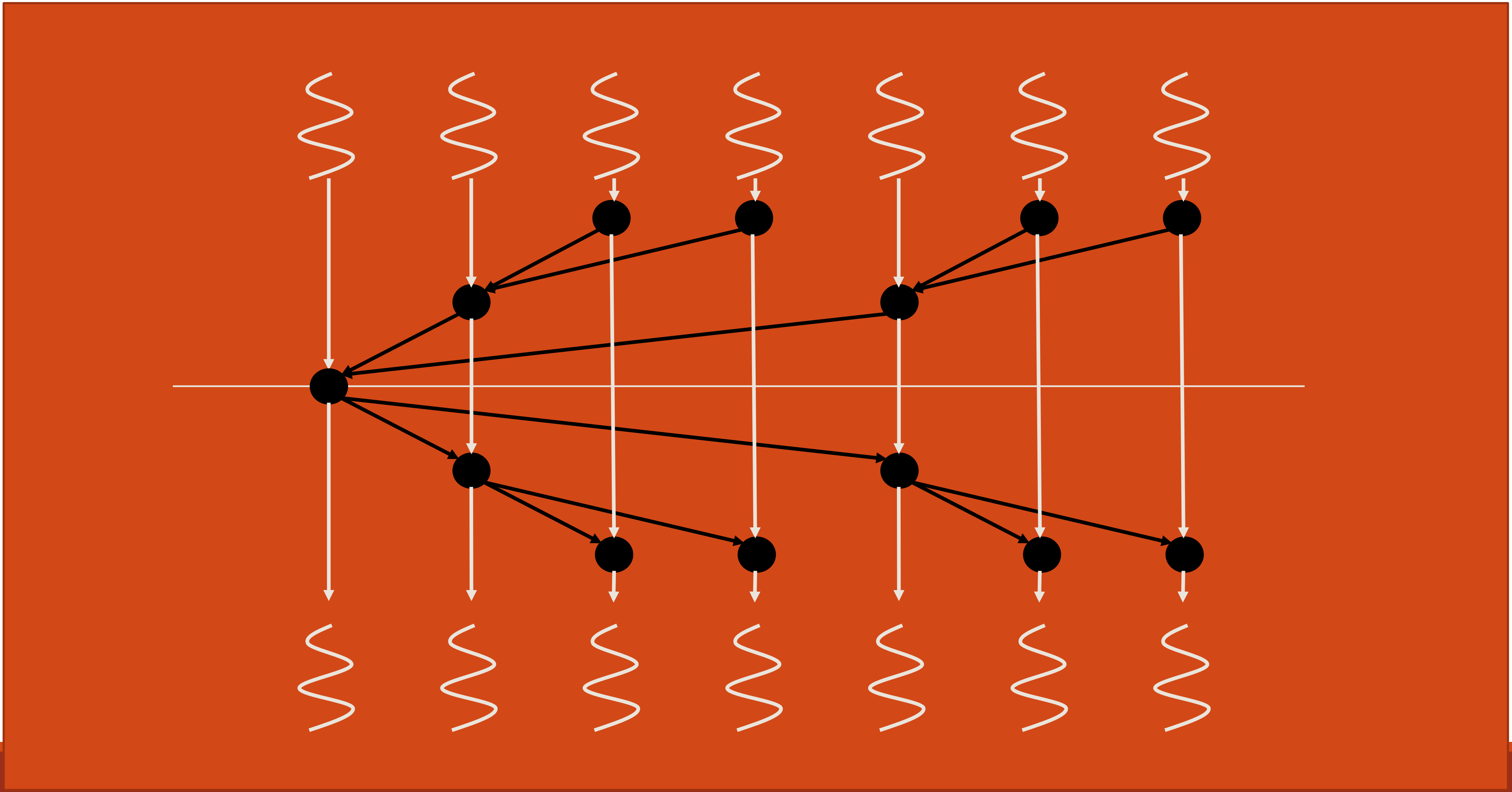
MCS Barrier with P=7



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?