Synchronization: Implementing Monitors + Barriers

Chris Rossbach & Calvin Lin

CS380P

Today

- Material for the day
  - Monitor implementation
  - Barrier implementation

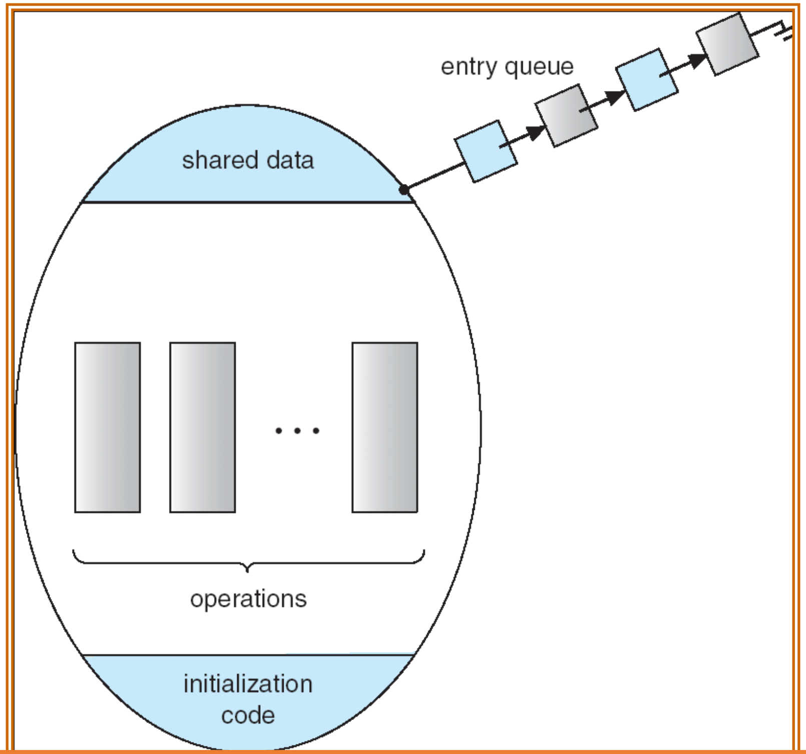
- Acknowledgements
  - Thanks to Gadi Taubenfield: we borrowed from some of his slides on barriers

What is a monitor?

- Same as a condition variable?

What is a monitor?

- Monitor: one big lock for set of operations/ methods
- Language-level implementation of mutex



- Entry procedure: called from outside
- Internal procedure: called within monitor
- Wait within monitor releases lock

Many variants…

Monitor != condition variable

- Encapsulates shared data behind API
- Compiler support usually involved
- May be built on conditions

Pthreads and conditions

- Why a mutex_t parameter for pthread_cond_wait?
- Why not in p_cond_init?

- Type `pthread_cond_t`

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_co
                      pthread_mut
int pthread_cond_signal(pthread_
int pthread_cond_broadcast(pthrea
```

Java:
`synchronized keyword`
`wait()/notify()/notifyAll()`

C#: Monitor class
`Enter()/Exit()/`
`Pulse()/PulseAll()`

Does this code work?

```
public class SynchronizedQueue<T> {

    public void enqueue(T item) {
        lock.lock();
        try {
            if(head == tail - 1)
                notFull.wait();
            Q[head] = item;
            if(++head == MAX_Q)
                head = 0;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public T dequeue() {
        T retval = null;
        lock.lock();
        try {
            if(head == tail)
                notEmpty.wait();
            retval = Q[tail];
            if(++tail == MAX_Q)
                tail = 0;
            notFull.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

```
private Lock lock = new ReentrantLock();
private Condition notEmpty = lock.newCondition();
private Condition notFull = lock.newCondition();
private int head = 0;
private int tail = 0;
private int size = MAX_Q;
private T[] Q = new T[size];
```

- Uses “if” to check invariants.
- Why doesn’t **if** work?
- How could we MAKE it work?

Hoare-style Monitors

(aka blocking condition variables)

Given entrance queue ‘e’, signal queue ‘s’, condition var ‘c’

```
enter:
    if(locked):
       e.push_back(thread)
    else
      lock
```

```
schedule:
    if s.any()
        t ← s.pop_first()
        t.run
    else if e.any()
        t ← e.pop_first()
        t. run
    else
        unlock // monitor unoccupied
```

```
wait C:

  C.q.push_back(thread)
  schedule   // block this thread
```

```
leave:
    schedule
```

```
signal C :

   if (C.q.any())
        t = C.q.pop_front() // t → "the signaled thread"
        s.push_back(thread)
        t.run
```

- Signaler must wait, but gets priority over threads on entrance queue
- Lock only released by
  - Schedule (if no waiters)
  - Application
- Pros/Cons?

Must run signaled thread immediately
Options for signaler:
- Switch out (go on s queue)
- Exit (Hansen monitors)
- Continue executing?

Mesa-style monitors
(aka non-blocking condition variables)

```
enter:
    if locked:
        e.push_back(thread)
        block
    else
        lock
```

```
schedule:
    if e.any()
        t ← e.pop_front
        t. run
    else
        unlock
```

```
notify C:

    if C.q.any()

        t ← C.q.pop_front() // t is "notified "
        e.push_back(t)
```

```
wait C:

  C.q.push_back(thread)
  schedule
  block
```

- Leave still calls schedule
- No signal queue
- Extendable with more queues for priority
- What are the differences/pros/cons?

Example: anyone see a bug?

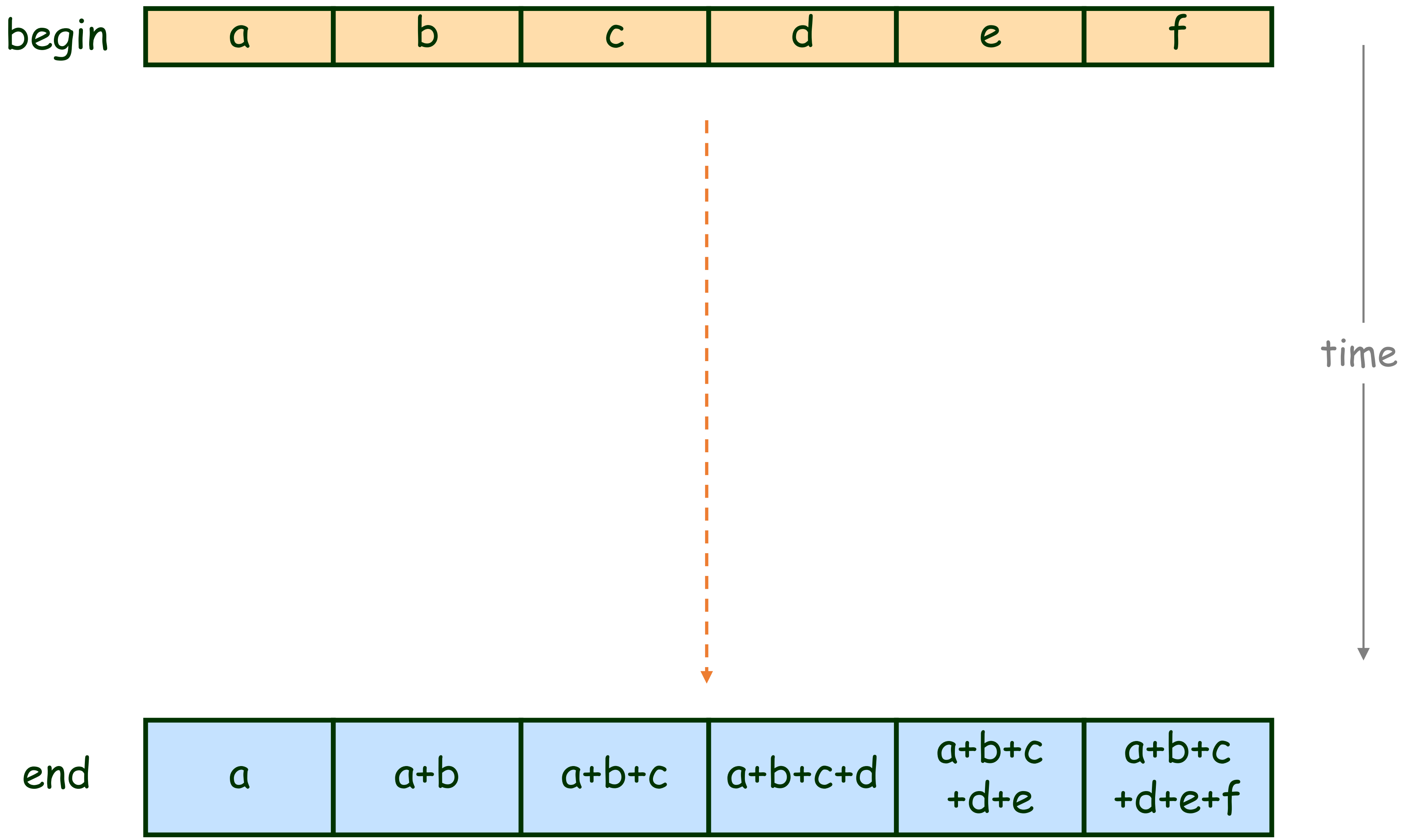
```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
        UNTIL availableStorage ≥ size
                DO WAIT moreAvailable ENDLOOP;
        p ← <remove chunk of size words & update availableStorage>
        END;

Free:   ENTRY PROCEDURE [p: POINTER, Size: INTEGER] = BEGIN
         <put back chunk of size words & update availableStorage>;
        NOTIFY moreAvailable END;

Expand:PUBLIC PROCEDURE [pOld: POINTER, size: INTEGER] RETURNS [pNew: POINTER] = BEGIN
        pNew ← Allocate[size];
        <copy contents from old block to new block>;
        Free[pOld] END;

END.
```

Solutions?

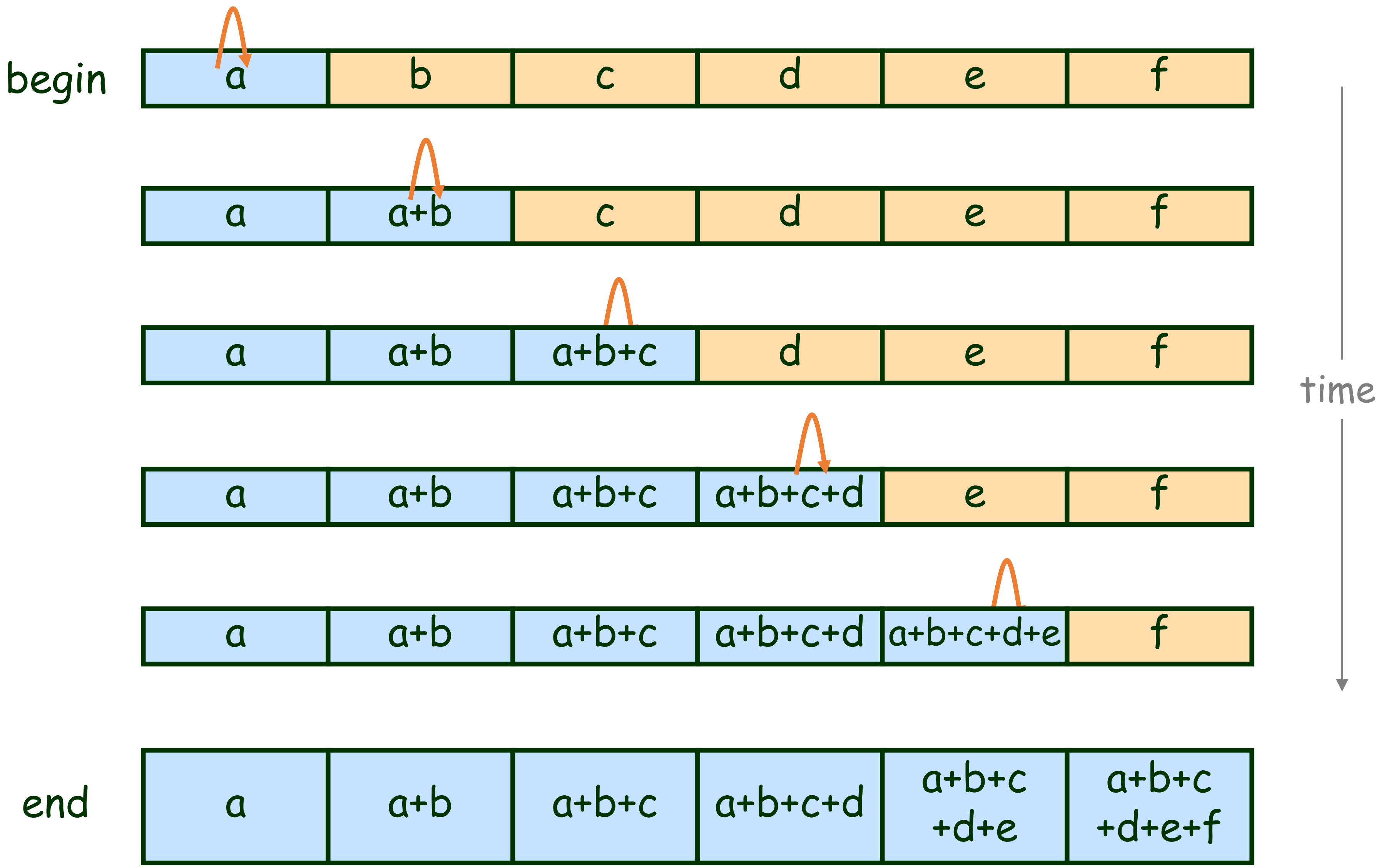
- Timeouts
- notifyAll
- Can Hoare monitors support notifyAll?

Barriers

Prefix Sum

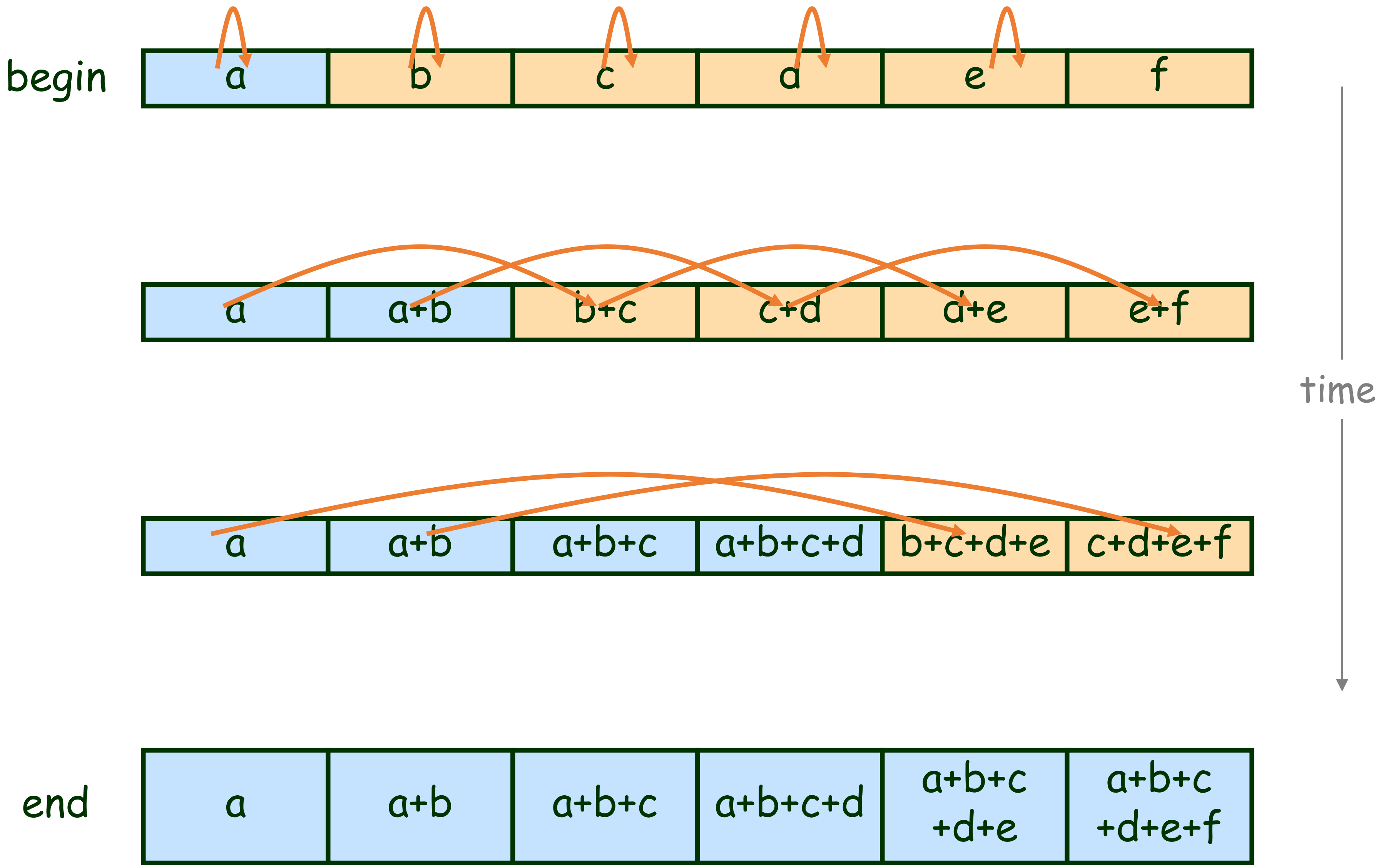


Prefix Sum

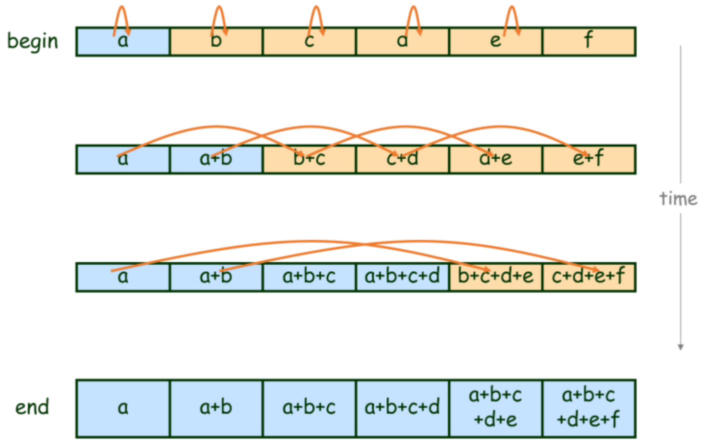


begin
a
b
c
d
e
f
a
a+b
c
d
e
f
a
a+b
a+b+c
d
e
f
a
a+b
a+b+c
a+b+c+d
e
f
a
a+b
a+b+c
a+b+c+d
a+b+c+d+e
f
end
a
a+b
a+b+c
a+b+c+d
a+b+c +d+e
a+b+c +d+e+f
time

Parallel Prefix Sum



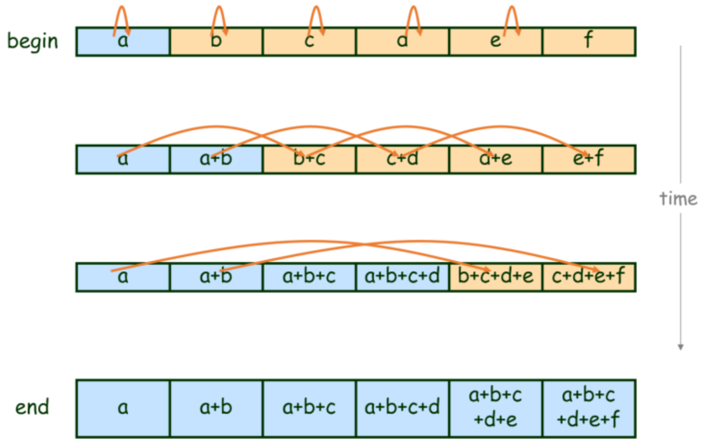
Pthreads Parallel Prefix Sum




```
int g_values[N] = { a, b, c, d, e, f };

void prefix_sum_thread(void * param) {

  int i;
  int id = *((int*)param);
  int stride = 0;

  for(stride=1; stride<=N/2; stride<<1) {
    g_values[id+stride] += g_values[id];
  }

}
```

Will this work?

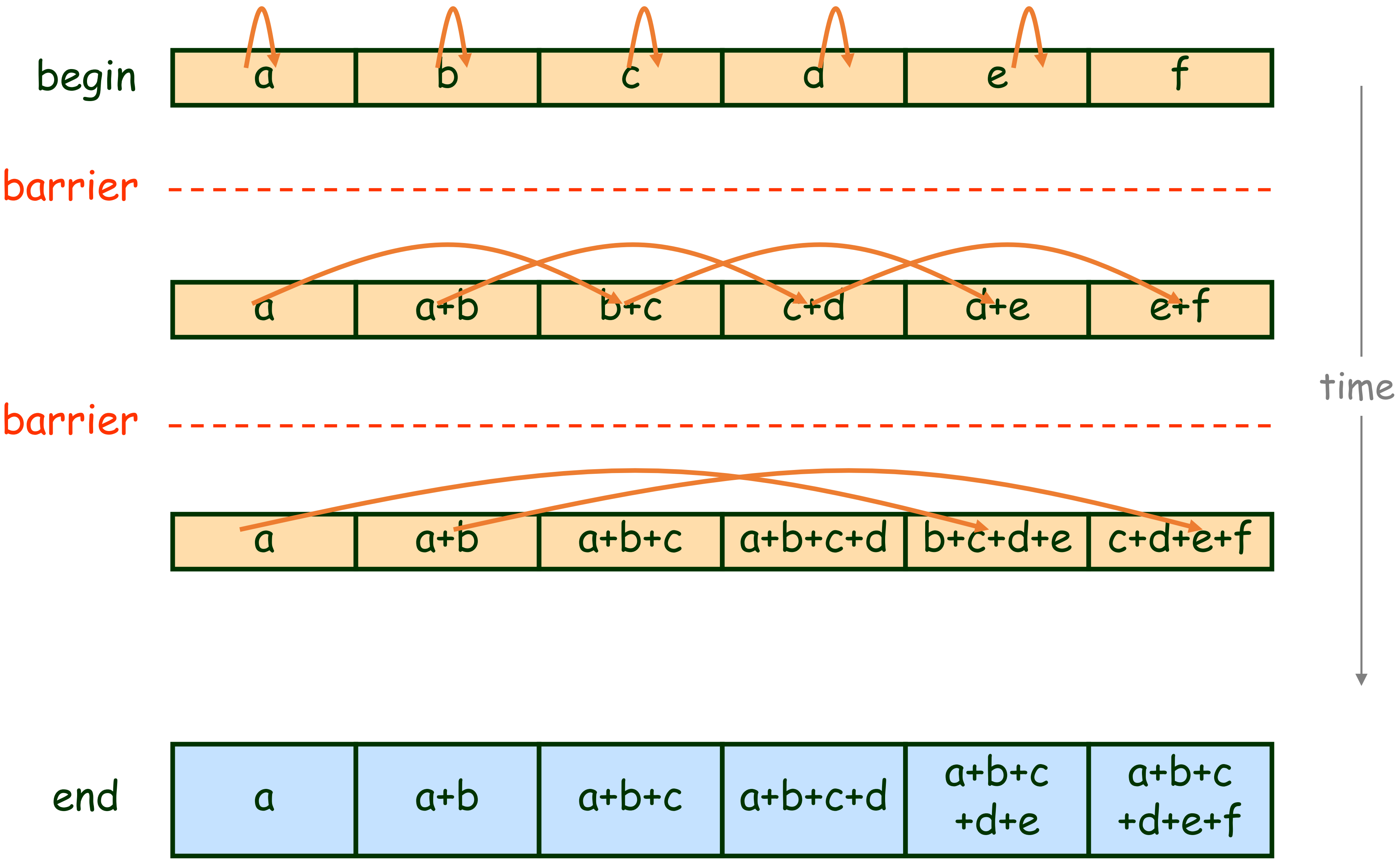
Pthreads Parallel Prefix Sum




```c
pthread_mutex_t g_locks[N] = { MUTEX_INITIALIZER, ...};
int g_values[N] = { a, b, c, d, e, f };

void prefix_sum_thread(void * param) {

  int i;
  int id = *((int*)param);
  int stride = 0;

  for(stride=1; stride<=N/2; stride<<1) {
    pthread_mutex_lock(&g_locks[id]);
    pthread_mutex_lock(&g_locks[id+stride]);
    g_values[id+stride] += g_values[id];
    pthread_mutex_unlock(&g_locks[id]);
    pthread_mutex_unlock(&g_locks[id+stride]);
  }

}
```

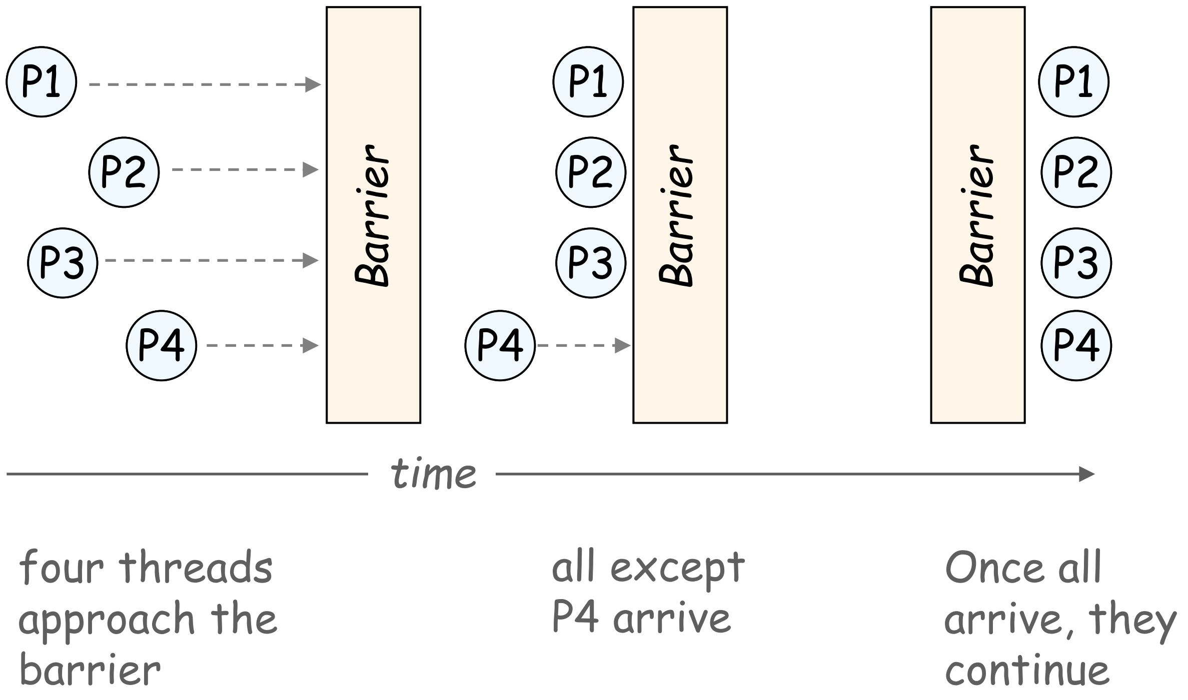
fixed?

Parallel Prefix Sum



What is a Barrier ?

- *Coordination mechanism (algorithm)*
- *threads wait until all reach specified point.*
- *Once all reach barrier, all can pass.*

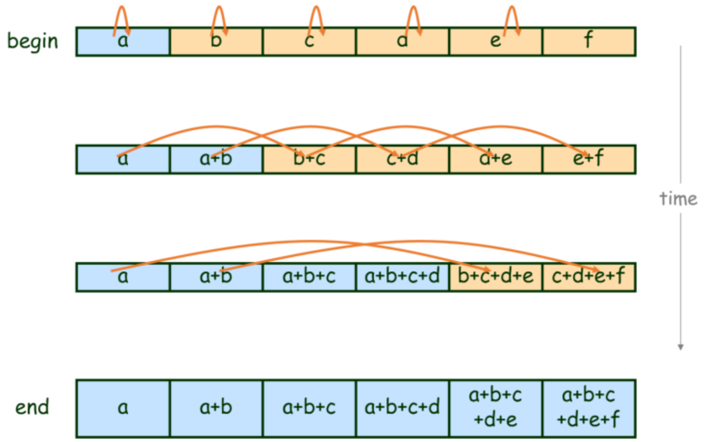


Pthreads and barriers

Type `pthread_barrier_t`

```
int pthread_barrier_init(pthread_barrier_t *barrier,
                         const pthread_barrierattr_t *attr,
                         unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Pthreads Parallel Prefix Sum




```
pthread_barrier_t g_barrier;
pthread_mutex_t g_locks[N];
int g_values[N] = { a, b, c, d, e, f };

void init_stuff() {
    ...
    pthread_barrier_init(&g_barrier, NULL, N-1);
}

void prefix_sum_thread(void * param) {

  int i;
  int id = *((int*)param);
  int stride = 0;

  for(stride=1; stride<=N/2; stride<<1) {

    pthread_mutex_lock(&g_locks[id]);
    pthread_mutex_lock(&g_locks[id+stride]);
    g_values[id+stride] += g_values[id];
    pthread_mutex_unlock(&g_locks[id]);
    pthread_mutex_unlock(&g_locks[id+stride]);

    pthread_barrier_wait(&g_barrier);
  }
}
```

fixed?

Barrier Goals

Desirable barrier properties:

- Low shared memory space complexity
- Low contention on shared objects
- Few shared memory references per thread/process
- No need for shared memory initialization
- Symmetric: same amount of work for all processes
- Algorithm simplicity
- Minimal propagation time
- Reusability (a must!)

Barrier Building Blocks

- Conditions
- Semaphores
- Atomic Bit
- Atomic Register
- Fetch-and-increment register
- Test and set bits
- Read-Modify-Write register

Barrier with Semaphores

Barrier using Semaphores

Algorithm for N threads

```
shared   sem_t arrival = 1;          // sem_init(&arrival, NULL, 1)
         sem_t departure = 0;        // sem_init(&departure, NULL, 0)
         atomic int counter = 0;     // (gcc intrinsics are verbose)
```

```
type __sync_fetch_and_add (type *ptr, type value, ...)
type __sync_fetch_and_sub (type *ptr, type value, ...)
type __sync_fetch_and_or (type *ptr, type value, ...)
type __sync_fetch_and_and (type *ptr, type value, ...)
type __sync_fetch_and_xor (type *ptr, type value, ...)
type __sync_fetch_and_nand (type *ptr, type value, ...)
```

Phase I

```
sem_wait(arrival);
if(++counter < N)
  sem_post(arrival);
else
  sem_post(departure);
```

Phase II

First N-1 threa[…] arrival, wait on

Nth thread post on departure, releasing threads into phase II (what is value of arrival?)

First N-1 threads post on departure, last posts arrival

Semaphore Barrier Action Zone

N == 3

shared sem_t arrival = 0

sem_t departure = 0

atomic int counter = 0

CPU 0

```
sem_wait(arrival);
if(++counter < N)
  sem_post(arrival);
else
  sem_post(departure);
sem_wait(departure);
if(--counter > 0)
  sem_post(departure)
else
  sem_post(arrival)
```

CPU 1

```
sem_wait(arrival);
if(++counter < N)
  sem_post(arrival);
else
  sem_post(departure);
sem_wait(departure);
if(--counter > 0)
  sem_post(departure)
else
  sem_post(arrival)
```

CPU 2

```
sem_wait(arrival);
if(++counter < N)
  sem_post(arrival);
else
  sem_post(departure);
sem_wait(departure);
if(--counter > 0)
  sem_post(departure)
else
  sem_post(arrival)
```

1

Do we need two phases?

Still correct if counter is not atomic?

Barrier using Semaphores

Properties

- Pros:
  - Very Simple
  - Space complexity O(1)
  - Symmetric
- Cons:
  - Required a strong object
    - Requires some central manager
    - High contention on the semaphores
  - Propagation delay O(n)

Barriers based on counters

Counter Barrier Ingredients

Fetch-and-Increment register

- A shared register that supports a F&I operation:
- Input: register *r*
- Atomic operation:
  - *r* is incremented by 1
  - the old value of r is returned

```
function fetch-and-increment (r : register)
        orig_r := r;
        r:= r + 1;
        return (orig_r);
end-function
```

Await

- For brevity, we use the **await** macro
- Not an operation of an object
- This is also called: “spinning”

```
macro await (condition : boolean condition)
        repeat
                cond = eval(condition);
        until (cond)
end-macro
```

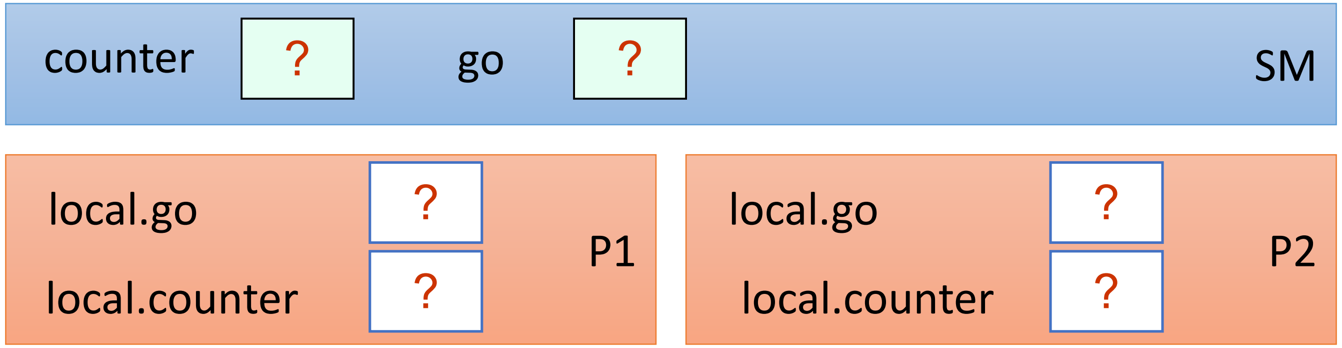
Simple Barrier Using an Atomic Counter

shared	counter: fetch and increment reg. – {0,..n}, initially = 0
	go: atomic bit, *initial value doesn't matter*
local	local.go: a bit, *initial value doesn't matter*
	local.counter: register

```
local.go := go
local.counter := fetch-and-increment (counter)
if local.counter + 1 = n then
        counter := 0
        go := 1 - go
else await(local.go ≠ go)
```

Simple Barrier Using an Atomic Counter

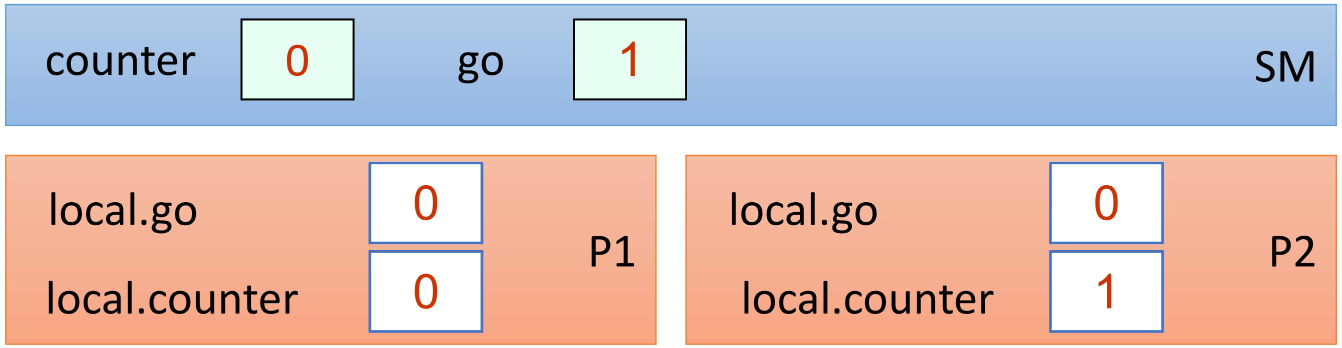
Run for n=2 Threads




```
local.go := go
local.counter := fetch-and-increment (counter)
if local.counter + 1 = n then
        counter := 0
        go := 1 - go
else await(local.go ≠ go)
```

Simple Barrier Using an Atomic Counter

Run for n=2 Threads



P2 → P1 →

```
local.go := go
local.counter := fetch-and-increment
if local.counter + 1 = n then
        counter := 0
        go := 1 - go
else await(local.go ≠ go)
```

1+1=2

P1 Busy wait

Pros/Cons?

- There is high memory contention on *go* bit
- Reducing the contention:
  - Replace the *go* bit with *n* bits: *go*[1],…,*go*[*n*]
  - Process p_i may spin only on the bit *go*[*i*]

A Local Spinning Counter Barrier

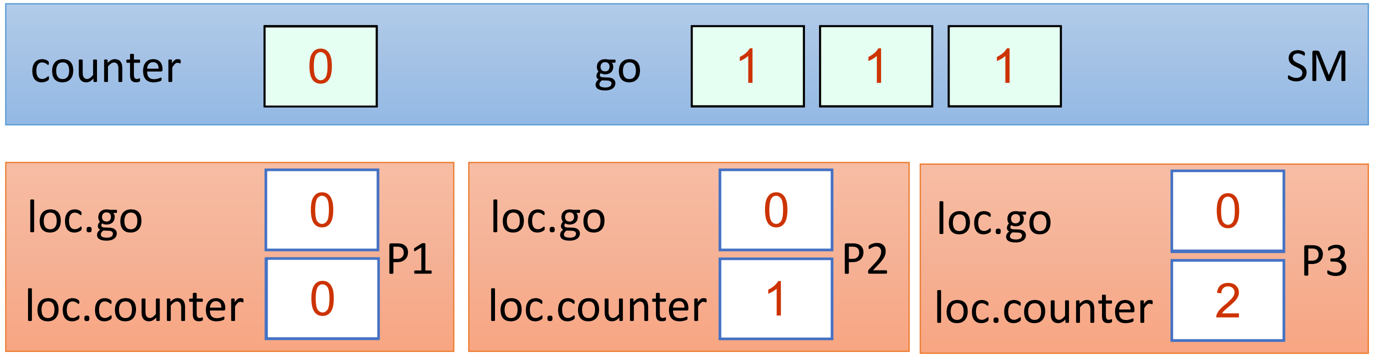
Program of a Thread i

shared	counter: fetch and increment reg. – {0,..n}, initially = 0
	go[1..n]: array of atomic bits, initial values are immaterial
local	local.go: a bit, initial value is immaterial
	local.counter: register

```
local.go := go[i]
local.counter := fetch-and-increment (counter)
if local.counter + 1 = n then
        counter := 0
        for j=1 to n { go[j] := 1 – go[j] }
else await(local.go ≠ go[i])
```

A Local Spinning Counter Barrier

Example Run for n=3 Threads



P3 → P2 → P1 →

```
local.go := go[i]
local.counter := fetch-and-increment(counter)
if local.counter + 1 = n then
        counter := 0
        for j=1 to n { go[j] := 1 – go[j] }
else await(local.go ≠ go[i])
```

2+1=3

P1,P2 Busy wait

Pros/Cons?
Does this actually reduce contention?

Comparison of counter-based Barriers

Simple Barrier

- Pros:
- Cons:

Simple Barrier with go array

- Pros:
- Cons:

Comparison of counter-based Barriers

Simple Barrier

- Pros:
  - Very Simple
  - Shared memory: O(log n) ***bits***
  - Takes O(1) until last waiting p is awaken
- Cons:
  - High contention on the go bit
  - Contention on the counter register (*)

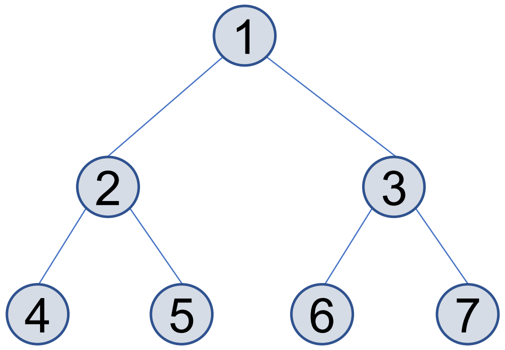
Simple Barrier with go array

- Pros:
  - Low contention on the go array
  - In some models:
    - spinning is done on local memory
    - remote mem. ref.: O(1)
- Cons:
  - Shared memory: O(n)
  - Still contention on the counter register (*)
  - Takes O(n) until last waiting p is awaken

Tree Barriers

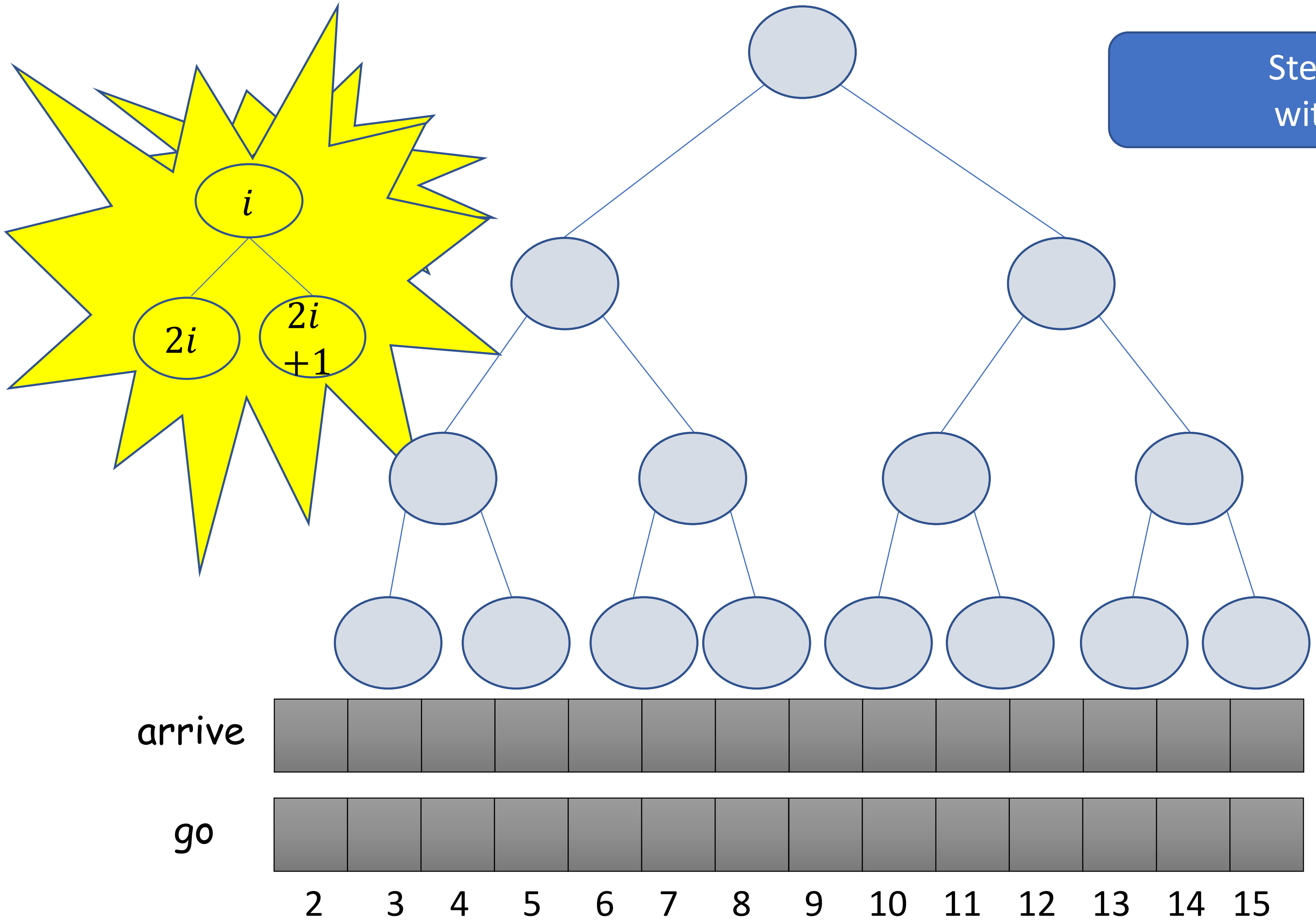
A Tree-based Barrier

- Threads are organized in a binary tree
- Each node is owned by a predetermined thread
- Each thread waits until its 2 children arrive
  - combines results
  - passes them on to its parent
- Root learns that its 2 children have arrived→tells children they can go
- The signal propagates down the tree until all the threads get the message



A Tree-based Barrier: indexing

Step 1: label numerically with depth-first traveral



Indexing starts from 2
Root → 1, doesn't need wait objects

A Tree-based Barrier

program of thread i

```
shared    arrive[2..n]: array of atomic bits, initial values = 0
          go[2..n]: array of atomic bits, initial values = 0
```

```
if i=1 then                                        // root
      await(arrive[2] = 1); arrive[2] := 0
      await(arrive[3] = 1); arrive[3] := 0
      go[2] = 1; go[3] = 1
else if i ≤ (n-1)/2 then                           // internal node
      await(arrive[2i] = 1); arrive[2i] := 0
      await(arrive[2i+1] = 1); arrive[2i+1] := 0
      arrive[i] := 1
      await(go[i] = 1); go[i] := 0
      go[2i] = 1; go[2i+1] := 1
else                                               // leaf
      arrive[i] := 1
      await(go[i] = 1); go[i] := 0 fi
fi
```

Root (lines 1–4)

Internal (lines 5–10)

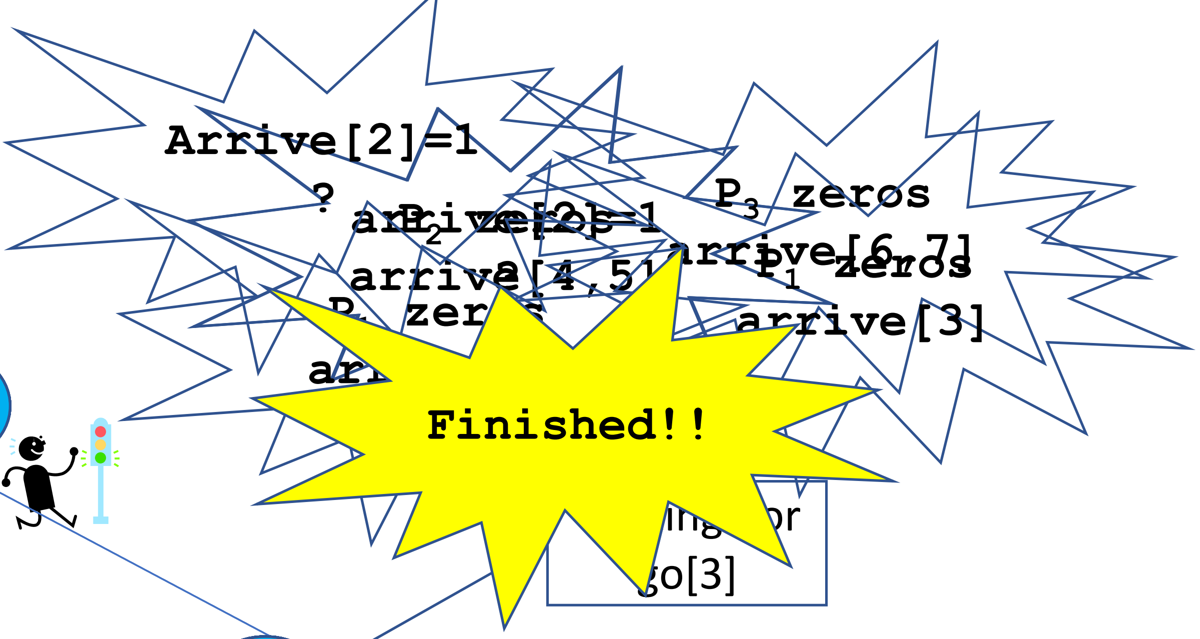
Leaf (lines 11–14)

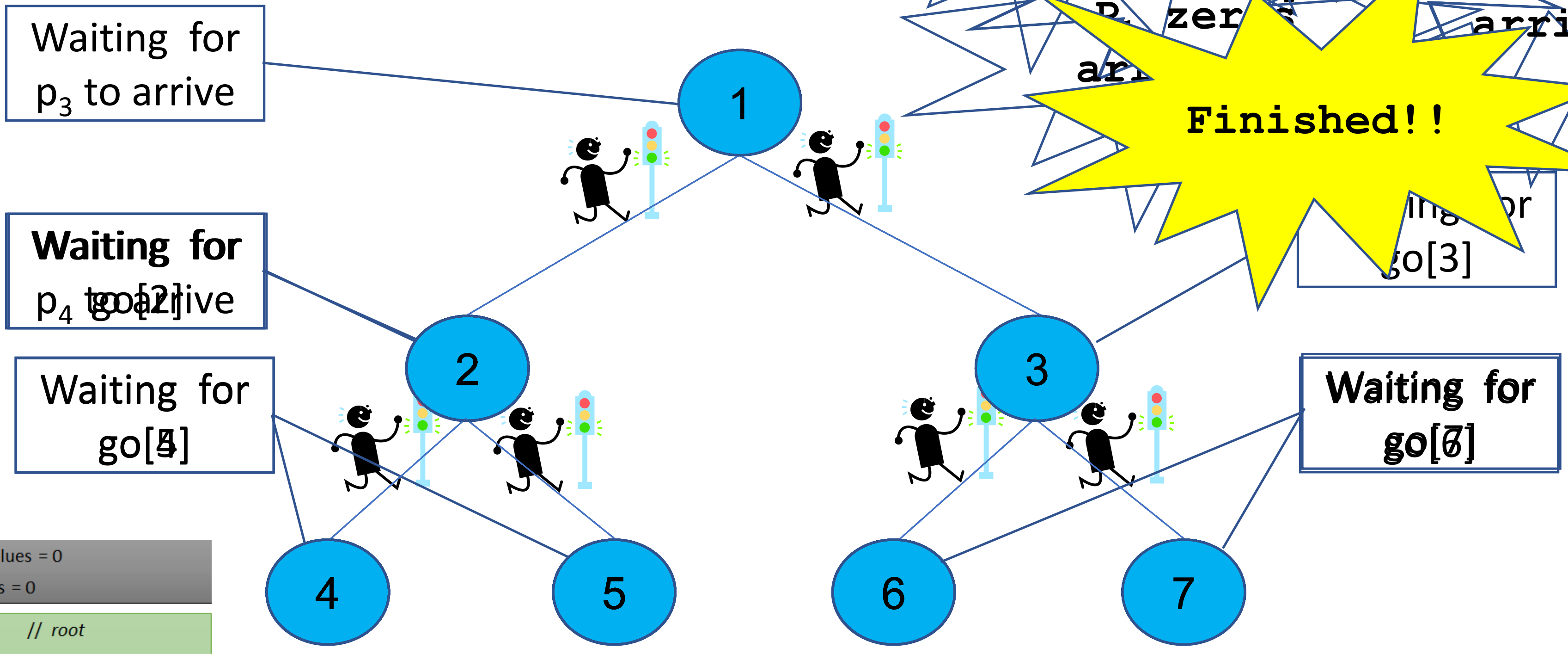
Root:
- Wait for arriving children
- Tell children to go

Internal:
- Wait for arriving children
- Wait for parent go signal
- Tell children to go

Child:
- arrive
- Wait for parent go signal

A Tree-based Barrier
Example Run for n=7 threads





shared	arrive[2..n]: array of atomic bits, initial values = 0
	go[2..n]: array of atomic bits, initial values = 0

```
if i=1 then                                   // root
    await(arrive[2] = 1); arrive[2] := 0
    await(arrive[3] = 1); arrive[3] := 0
    go[2] = 1; go[3] = 1
else if i ≤ (n-1)/2 then                      // internal node
    await(arrive[2i] = 1); arrive[2i] := 0
    await(arrive[2i+1] = 1); arrive[2i+1] := 0
    arrive[i] := 1
    await(go[i] = 1); go[i] := 0
    go[2i] = 1; go[2i+1] := 1
else                                          // leaf
    arrive[i] := 1
    await(go[i] = 1); go[i] := 0 fi
fi
```

	2	3	4	5	6	7
arrive	0	0	0	0	0	0
go	1	1	1	1	1	1

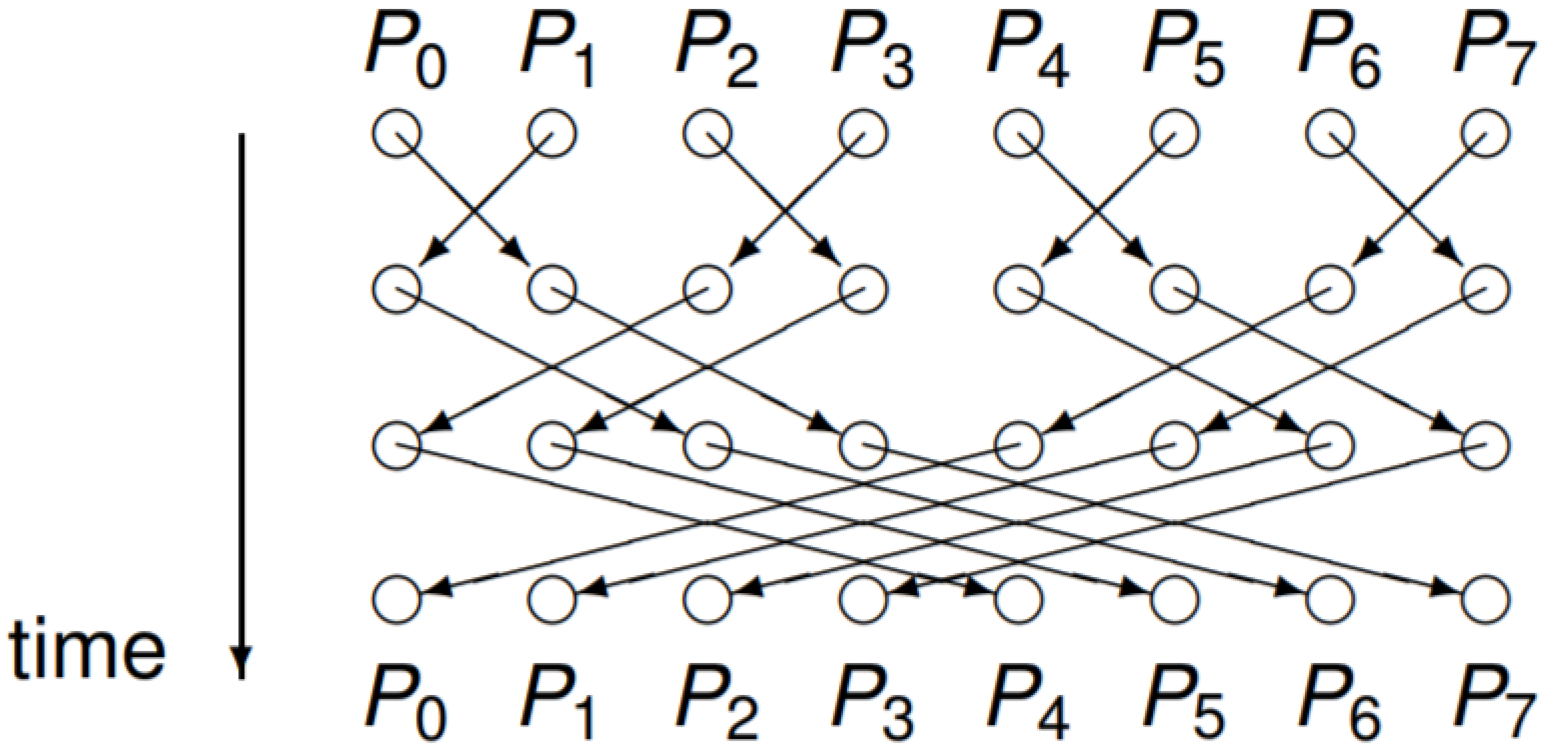
At this point all non-root threads in some await(go) case

Tree Barrier Tradeoffs

- Pros:

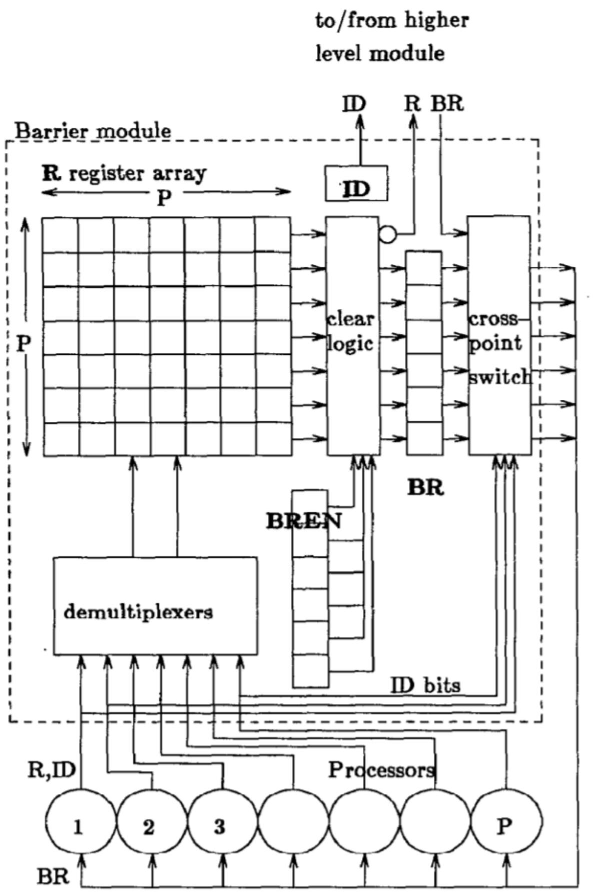
- Cons:

Butterfly Barrier



- When would this be preferable?

Hardware Supported Barriers



CPU

Barriers Summary

Seen:

- Semaphore-based barrier
- Simple barrier
  - Based on atomic fetch-and-increment counter
- Local spinning barrier
  - Based on atomic fetch-and-increment counter and go array
- Tree-based barrier

Not seen:

- Test-and-Set barriers
  - Based on test-and-test-and-set objects
  - One version without memory initialization
- See-Saw barrier

Questions?