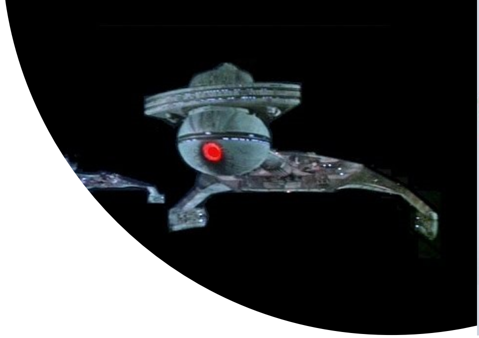
Synchronization: Semaphores, Monitors, Barriers

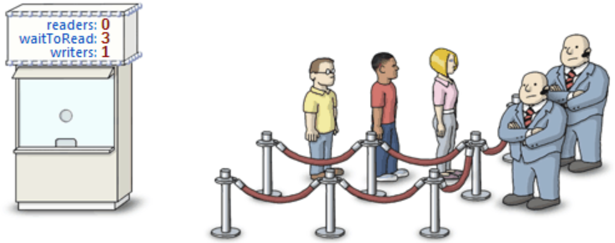
Chris Rossbach

CS378H

Today



- Questions?
- Administrivia
  - Start looking at Lab 2!
- Material for the day
  - Lab 1 discussion
  - Semaphores
  - Monitors
  - Barriers

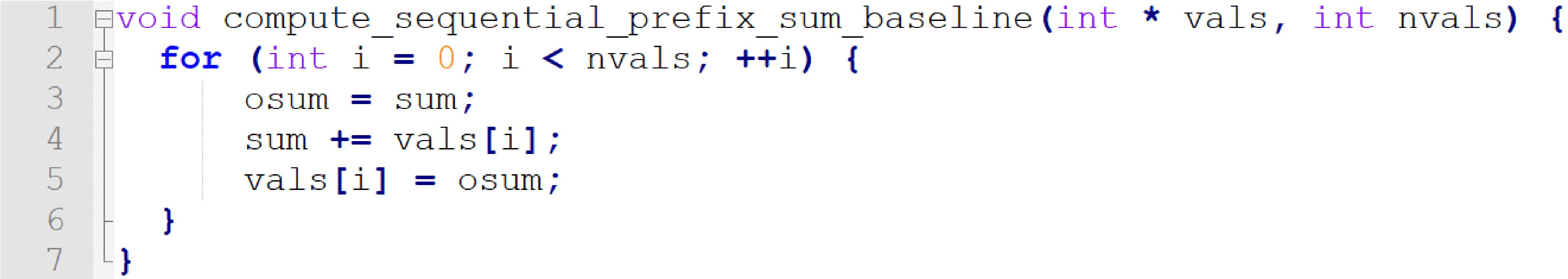


- Acknowledgements
  - Thanks to Gadi Taubenfield: I borrowed and modified some of his slides on barriers
- Image credits
  - https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjxi4uip8LdAhWFq1MKHbBeD4sQjRx6BAgBEAU&url=http%3A%2F%2Fpreshing.com%2F20150316%2Fsemaphores-are-surprisingly-versatile&psig=AOvVaw20Zw2eU9WAmbX8qxDSLSRd&ust=1537282884760655
  - https://images-na.ssl-images-amazon.com/images/I/31EcIPmMniL.jpg
  - https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjBivLOp8LdAhWF0VMKHdMvAnwQjRx6BAgBEAU&url=https%3A%2F%2Fprocastproducts.com%2Falaska-barriers-10-tall&psig=AOvVaw24KBCgTpBd7ynNpqcwcaqO&ust=1537282983281741

Faux Quiz (answer any 2, 5 min)

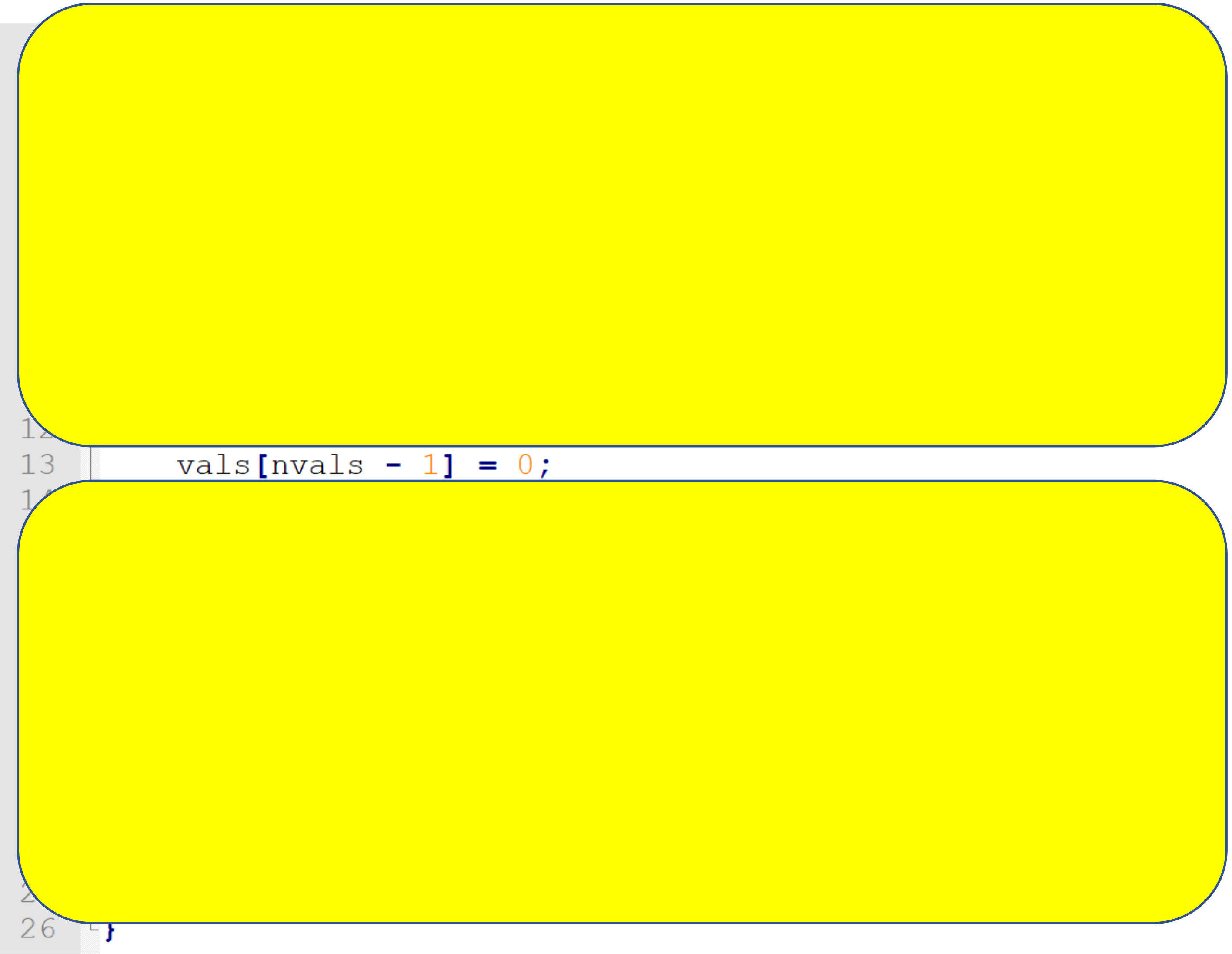
- What is the difference between Mesa and Hoare monitors?
- Why recheck the condition on wakeup from a monitor wait?
- How can you build a barrier with spinlocks?
- How can you build a barrier with monitors?
- How can you build a barrier without spinlocks or monitors?
- What is the difference between mutex and semaphores?
- How are monitors and semaphores related?
- Why does pthread_cond_init accept a pthread_mutex_t parameter? Could it use a pthread_spinlock_t? Why [not]?
- Why do modern CPUs have both coherence and HW-supported RMW instructions? Why not just one or the other?
- What is priority inheritance?

Lab 1: Baseline



```
void compute_sequential_prefix_sum_baseline(int * vals, int nvals) {
  for (int i = 0; i < nvals; ++i) {
      osum = sum;
      sum += vals[i];
      vals[i] = osum;
  }
}
```

Lab 1: Algorithm in Sequential Context

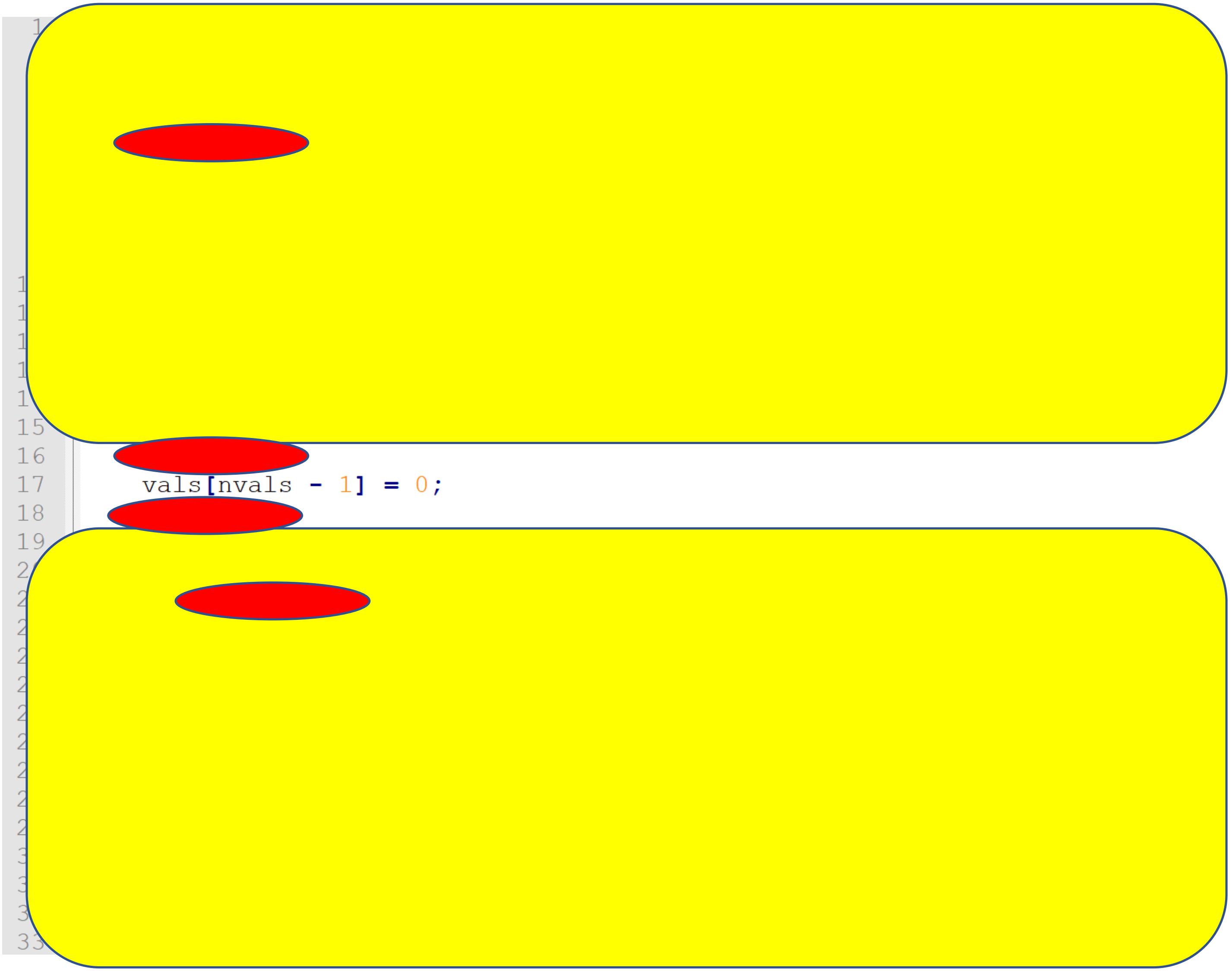


```
    vals[nvals - 1] = 0;
}
```

Upsweep

Downsweep

Lab 1: Parallel



```
vals[nvals - 1] = 0;
```

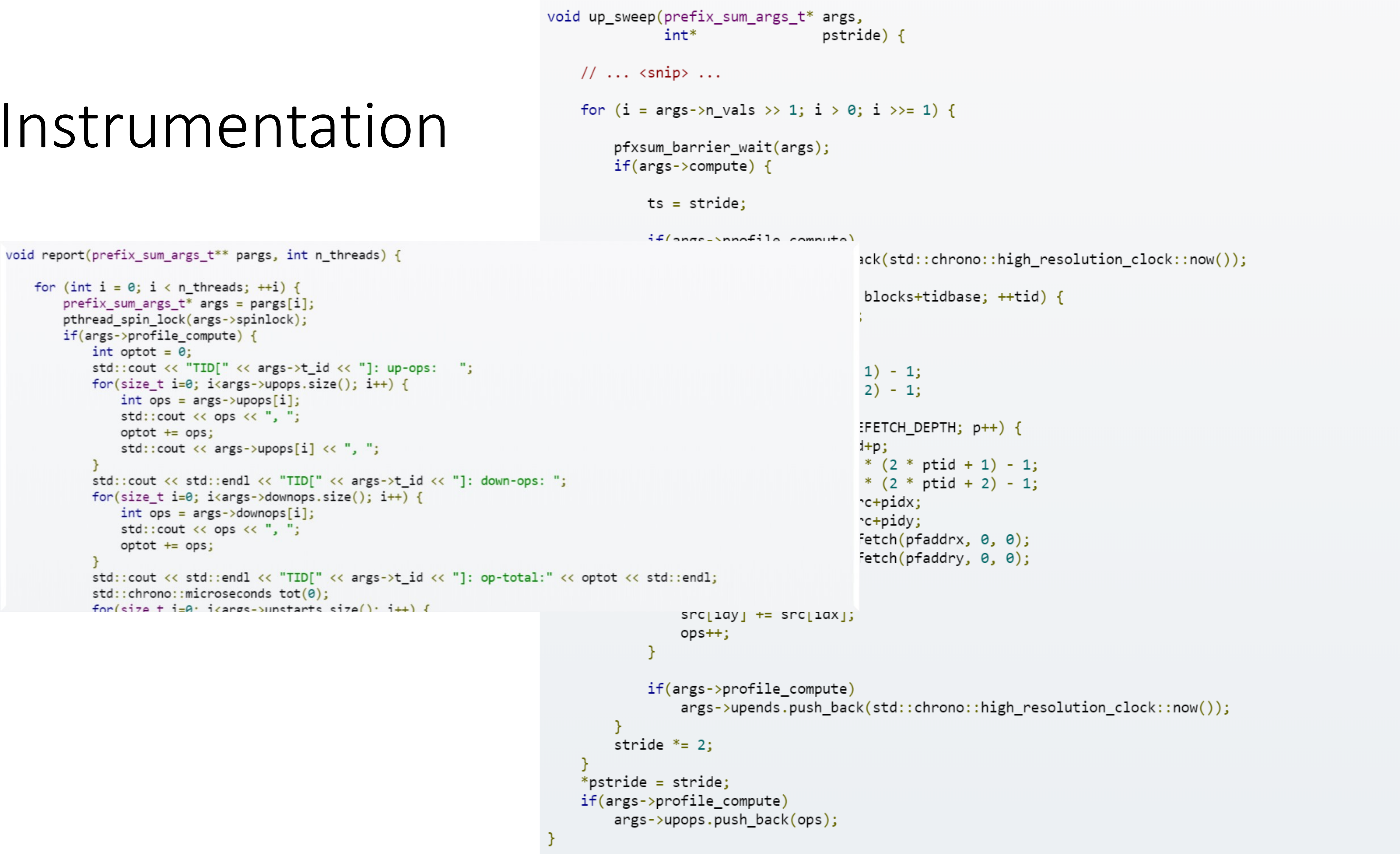
Upsweep

Downsweep

Instrumentation

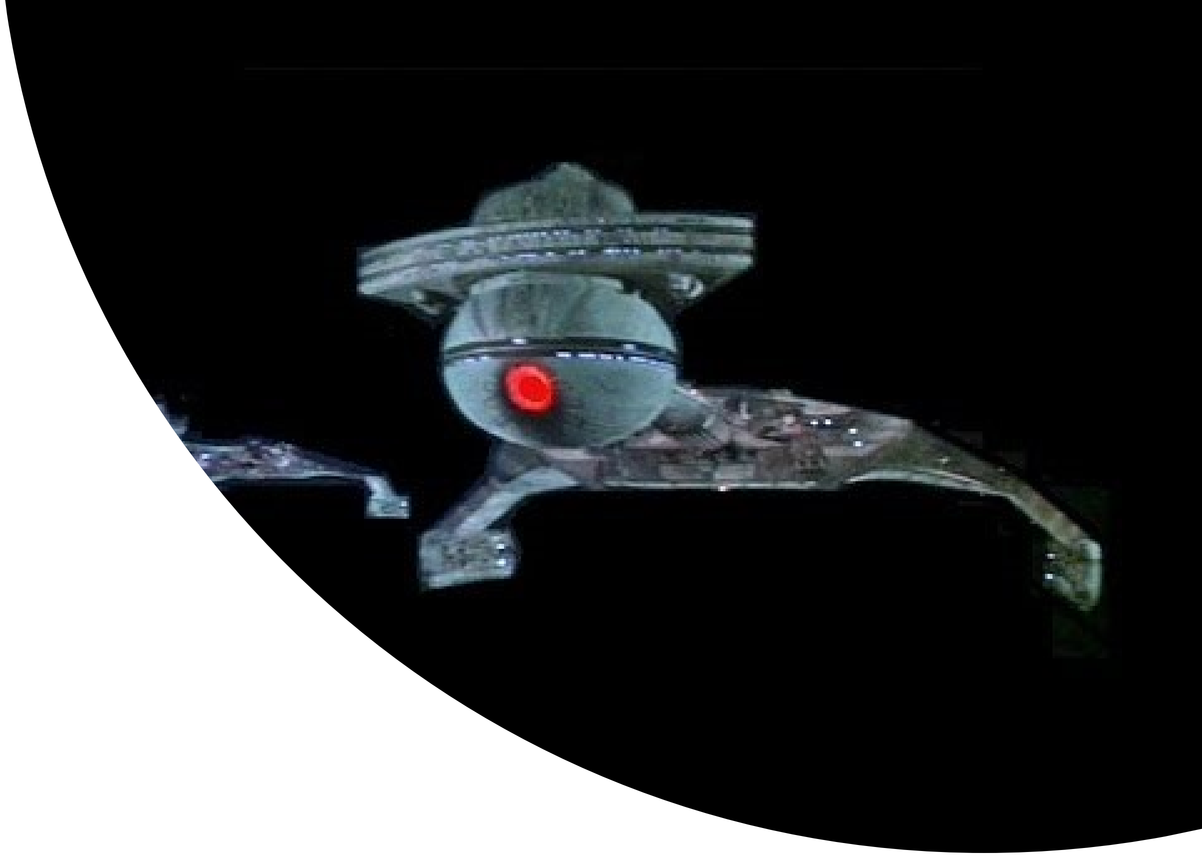
```
struct prefix_sum_args_t {
    int*               input_vals;
    int*               output_vals;
    int*               vals_padded;
    bool               spin;
    bool               compute;
    bool               profile_compute;
    bool               profile_barriers;
    bool               no_barrier;
    bool               sequential_sweep;
    bool               prefetch;
    bool               affinity;
    bool               syncwake;
    pthread_barrier_t* barrier;
    pthread_barrier_t* wakebarrier;
    pthread_spinlock_t* spinlock;
    spin_barrier*      s_barrier;
    int                n_vals;
    int                n_vals_padded;
    int                n_blocks;
    int                n_threads;
    int                n_chunk_size;
    int                t_id;
    std::vector<int> upops;
    std::vector<int> downops;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> upstarts;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> upends;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> downstarts;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> downends;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> barrierin;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> barrierout;

    prefix_sum_args_t() {
        compute = true;
        spin = false;
        no_barrier = false;
        profile_compute = false;
        profile_barriers = false;
        sequential_sweep = false;
        prefetch = false;
        affinity = false;
        syncwake = true;

        upops.reserve(2000);
        downops.reserve(2000);
```

Instrumentation



```
void report(prefix_sum_args_t** pargs, int n_threads) {

    for (int i = 0; i < n_threads; ++i) {
        prefix_sum_args_t* args = pargs[i];
        pthread_spin_lock(args->spinlock);
        if(args->profile_compute) {
            int optot = 0;
            std::cout << "TID[" << args->t_id << "]: up-ops:   ";
            for(size_t i=0; i<args->upops.size(); i++) {
                int ops = args->upops[i];
                std::cout << ops << ", ";
                optot += ops;
                std::cout << args->upops[i] << ", ";
            }
            std::cout << std::endl << "TID[" << args->t_id << "]: down-ops: ";
            for(size_t i=0; i<args->downops.size(); i++) {
                int ops = args->downops[i];
                std::cout << ops << ", ";
                optot += ops;
            }
            std::cout << std::endl << "TID[" << args->t_id << "]: op-total:" << optot << std::endl;
            std::chrono::microseconds tot(0);
            for(size_t i=0; i<args->upstarts.size(); i++) {
```

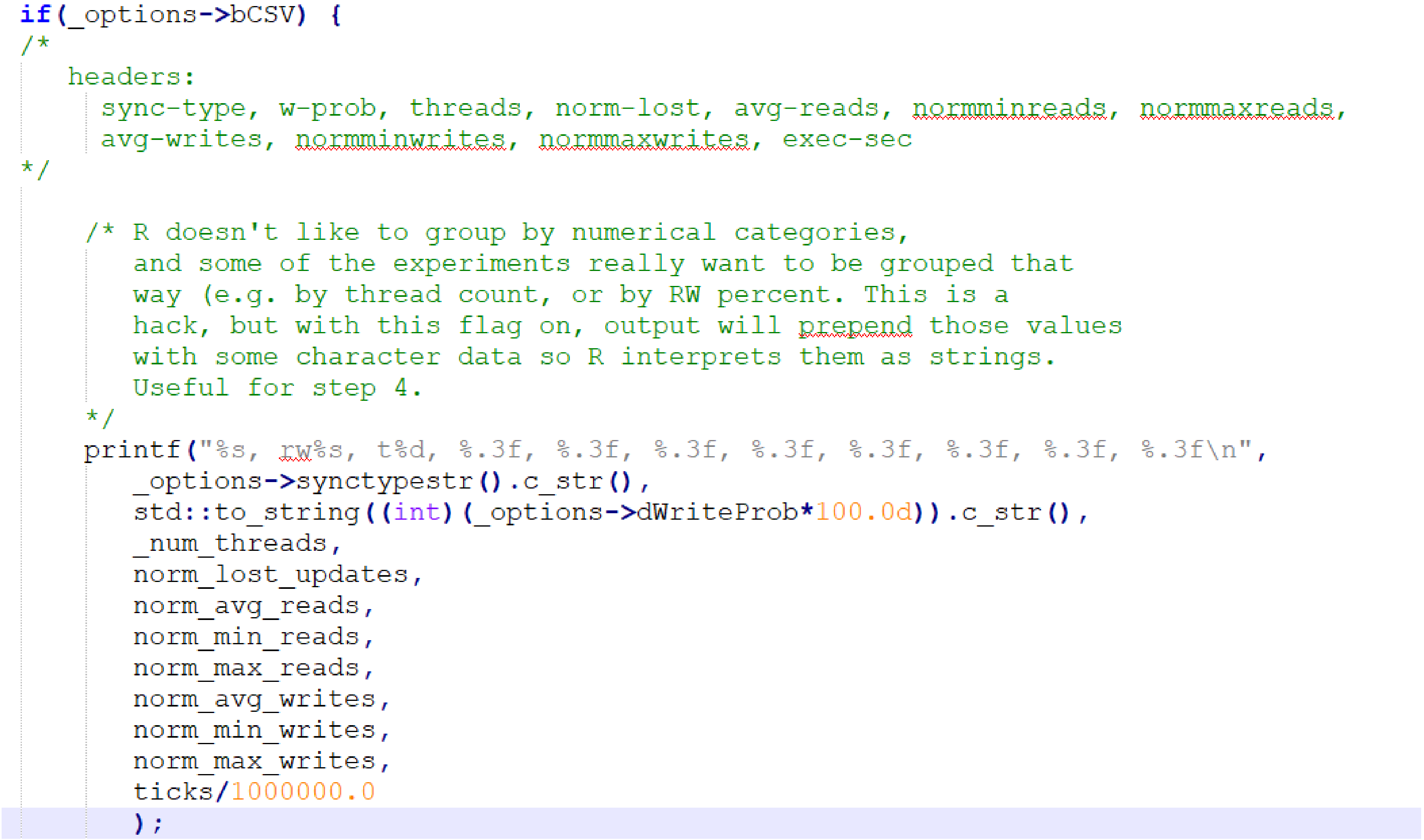
```
void up_sweep(prefix_sum_args_t* args,
              int*               pstride) {

    // ... <snip> ...

    for (i = args->n_vals >> 1; i > 0; i >>= 1) {

        pfxsum_barrier_wait(args);
        if(args->compute) {

            ts = stride;

            if(args->profile_compute)
                                      ack(std::chrono::high_resolution_clock::now());

                                      blocks+tidbase; ++tid) {
                                      ;


                                      1) - 1;
                                      2) - 1;

                                      EFETCH_DEPTH; p++) {
                                      d+p;
                                      * (2 * ptid + 1) - 1;
                                      * (2 * ptid + 2) - 1;
                                      c+pidx;
                                      c+pidy;
                                      fetch(pfaddrx, 0, 0);
                                      fetch(pfaddry, 0, 0);


                src[idy] += src[idx];
                ops++;
            }

            if(args->profile_compute)
                args->upends.push_back(std::chrono::high_resolution_clock::now());
        }
        stride *= 2;
    }
    *pstride = stride;
    if(args->profile_compute)
        args->upops.push_back(ops);
}
```



Discussion

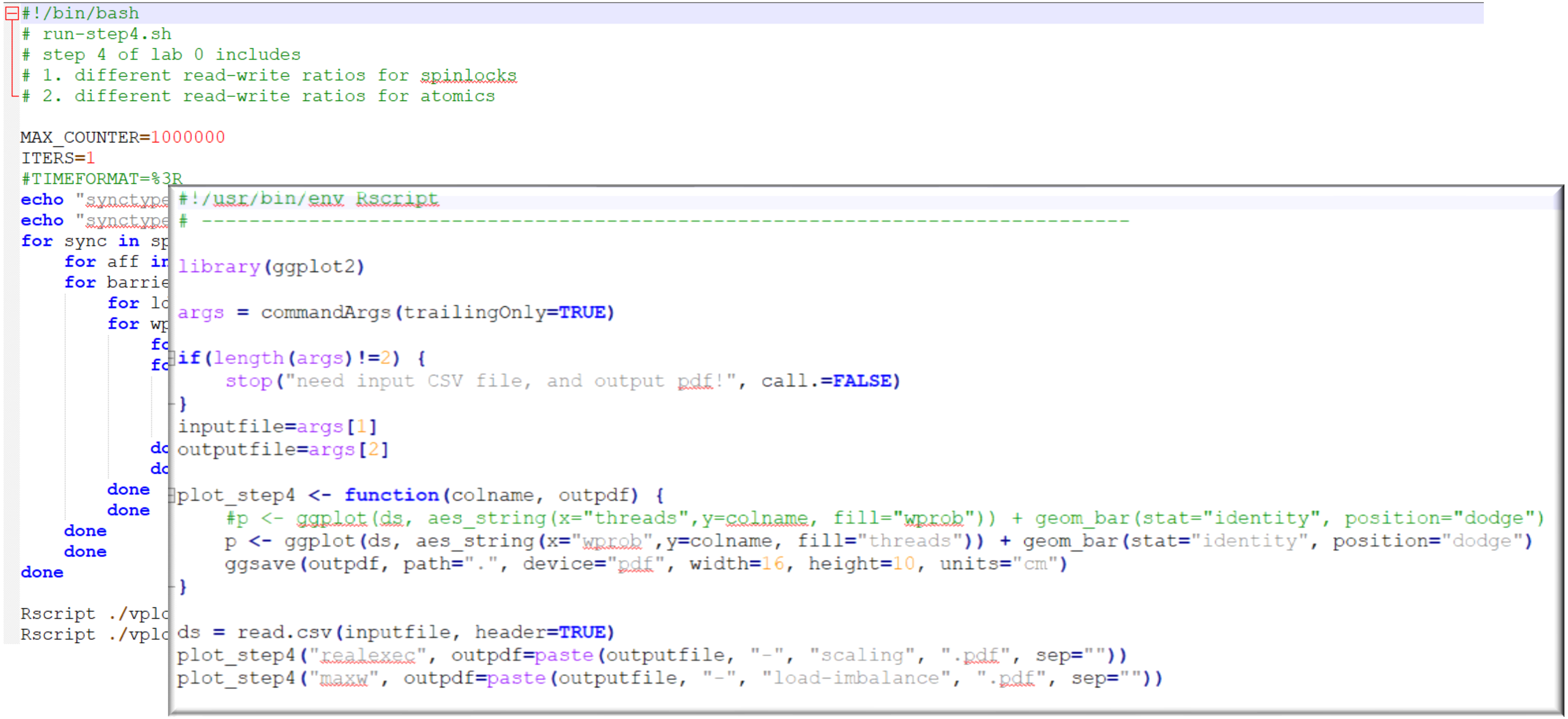
Could you make it scale?

Lab Tricks: Output CSV



```
if(_options->bCSV) {
/*
   headers:
     sync-type, w-prob, threads, norm-lost, avg-reads, normminreads, normmaxreads,
     avg-writes, normminwrites, normmaxwrites, exec-sec
*/

    /* R doesn't like to group by numerical categories,
       and some of the experiments really want to be grouped that
       way (e.g. by thread count, or by RW percent. This is a
       hack, but with this flag on, output will prepend those values
       with some character data so R interprets them as strings.
       Useful for step 4.
    */
    printf("%s, rw%s, t%d, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f, %.3f\n",
       _options->synctypestr().c_str(),
       std::to_string((int)(_options->dWriteProb*100.0d)).c_str(),
       _num_threads,
       norm_lost_updates,
       norm_avg_reads,
       norm_min_reads,
       norm_max_reads,
       norm_avg_writes,
       norm_min_writes,
       norm_max_writes,
       ticks/1000000.0
       );
```

Lab Tricks: scripting your experiments

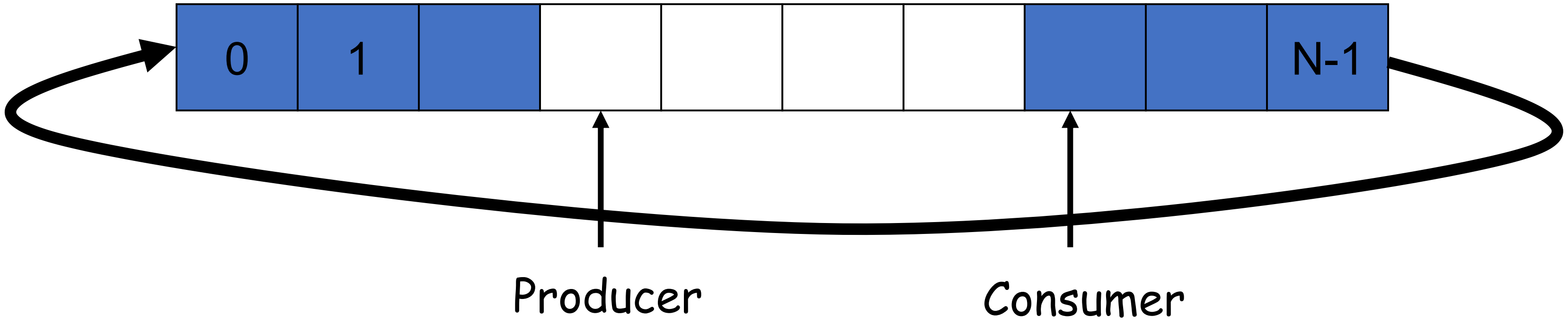


```
#!/bin/bash
# run-step4.sh
# step 4 of lab 0 includes
# 1. different read-write ratios for spinlocks
# 2. different read-write ratios for atomics

MAX_COUNTER=1000000
ITERS=1
#TIMEFORMAT=%3R
echo "synctype
echo "synctype
for sync in sp
    for aff in
    for barrie
        for lo
        for wp
            fo
            fo


            do
            do
        done
        done
    done
    done
done

Rscript ./vplo
Rscript ./vplo
```

```
#!/usr/bin/env Rscript
# ----------------------------------------------------------------------------

library(ggplot2)

args = commandArgs(trailingOnly=TRUE)

if(length(args)!=2) {
    stop("need input CSV file, and output pdf!", call.=FALSE)
}
inputfile=args[1]
outputfile=args[2]

plot_step4 <- function(colname, outpdf) {
    #p <- ggplot(ds, aes_string(x="threads",y=colname, fill="wprob")) + geom_bar(stat="identity", position="dodge")
    p <- ggplot(ds, aes_string(x="wprob",y=colname, fill="threads")) + geom_bar(stat="identity", position="dodge")
    ggsave(outpdf, path=".", device="pdf", width=16, height=10, units="cm")
}

ds = read.csv(inputfile, header=TRUE)
plot_step4("realexec", outpdf=paste(outputfile, "-", "scaling", ".pdf", sep=""))
plot_step4("maxw", outpdf=paste(outputfile, "-", "load-imbalance", ".pdf", sep=""))
```

Producer-Consumer (Bounded-Buffer) Problem

- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “consumes”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data

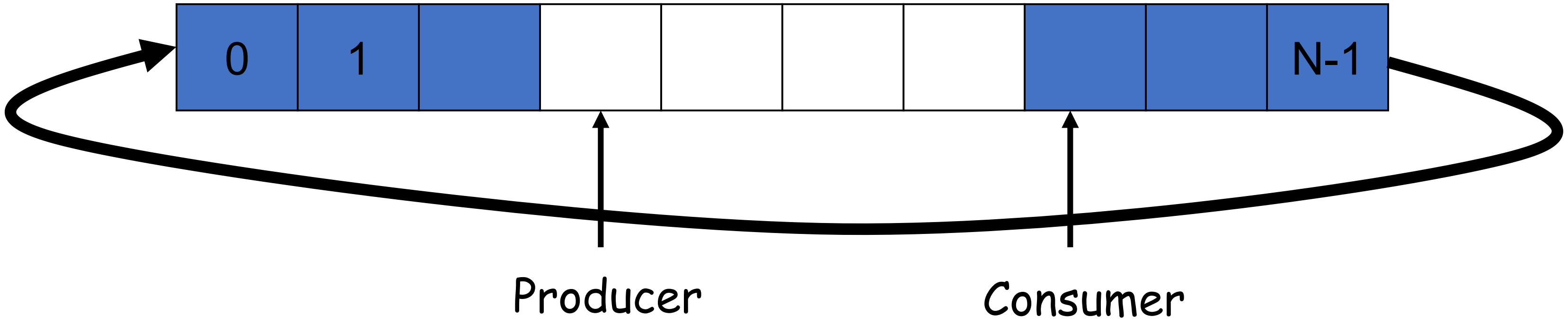


OK, let’s write some code for this

(using locks only)

- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer writes data
- Consumer reads data

```
object array[N]
void enqueue(object x);
object dequeue();
```

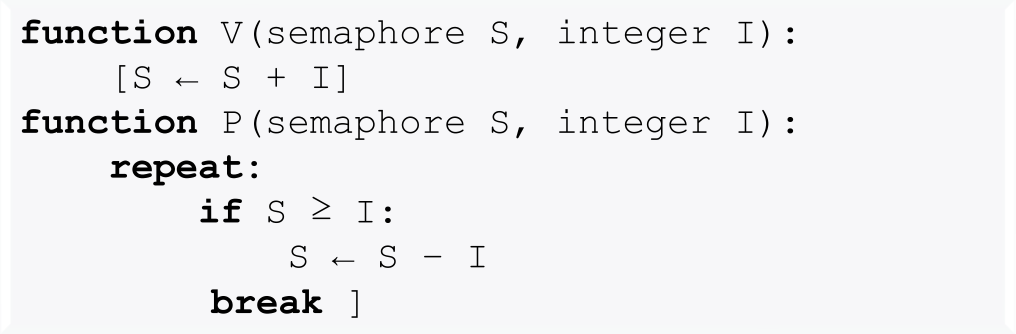


Semaphore Motivation

- Problem with locks: mutual exclusion, but *no ordering*
- Inefficient for producer-consumer (and lots of other things)
  - Producer: creates a resource
  - Consumer: uses a resource
  - bounded buffer between them
  - You need synchronization for correctness, *and…*
  - Scheduling order:
    - producer waits if buffer full, consumer waits if buffer empty

Semaphores

- Synchronization variable
  - Integer value
    - Can’t access value directly
    - Must initialize to some value
      - sem_init(sem_t *s, int pshared, unsigned int value)
  - Two operations
    - sem_wait, or down(), P()
    - sem_post, or up(), V()



```
function V(semaphore S, integer I):
    [S ← S + I]
function P(semaphore S, integer I):
    repeat:
        if S ≥ I:
            S ← S - I
         break ]
```

```
int sem_wait(sem_t *s) {
    wait until value of semaphore s
          is greater than 0
    decrement the value of
          semaphore s by 1
}
```

```
int sem_post(sem_t *s) {
    increment the value of
          semaphore s by 1
    if there are 1 or more
          threads waiting, wake 1
}
```

Semaphore Uses

```
// initialize to X
sem_init(s, 0, X)

sem_wait(s);
// critical section
sem_post(s);
```

- Mutual exclusion
  - Semaphore as mutex
  - What should initial value be?
    - Binary semaphore: X=1
    - (Counting semaphore: X>1)
- Scheduling order
  - One thread waits for another
  - What should initial value be?

```
// thread 0
… // 1st half of computation
sem_post(s);
```

```
// thread 1

sem_wait(s);
… //2nd half of computation
```

Producer-Consumer with semaphores

- Two semaphores
  - sem_t full; // # of filled slots
  - sem_t empty; // # of empty slots
- Problem: mutual exclusion?

Is this correct?

```
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);
```

```
producer() {
    sem_wait(empty);
    … // fill a slot
    sem_post(full);
}
```

```
consumer() {
    sem_wait(full);
    … // empty a slot
    sem_post(empty);
}
```

Producer-Consumer with semaphores

- Three semaphores
  - sem_t full; // # of filled slots
  - sem_t empty; // # of empty slots
  - sem_t mutex; // mutual exclusion

```
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);
sem_init(&mutex, 0, 1);
```

```
producer() {
    sem_wait(empty);
    sem_wait(&mutex);
    … // fill a slot
    sem_post(&mutex);
    sem_post(full);
}
```

```
consumer() {
    sem_wait(full);
    sem_wait(&mutex);
    … // empty a slot
    sem_post(&mutex);
    sem_post(empty);
}
```

Pthreads and Semaphores

- No pthread_semaphore_t!
-
-

- int **sem_wait**(sem_t ***sem**)

y sem is greater
e count

hore pointed to
he semaphore,

signed int

between threads

- else shared between processes

- Type: `pthread_semaphore_t`

```
int pthread_semaphore_init(pthread_spinlock_t *lock);
int pthread_semaphore_destroy(pthread_spinlock_t *lock);
```

…

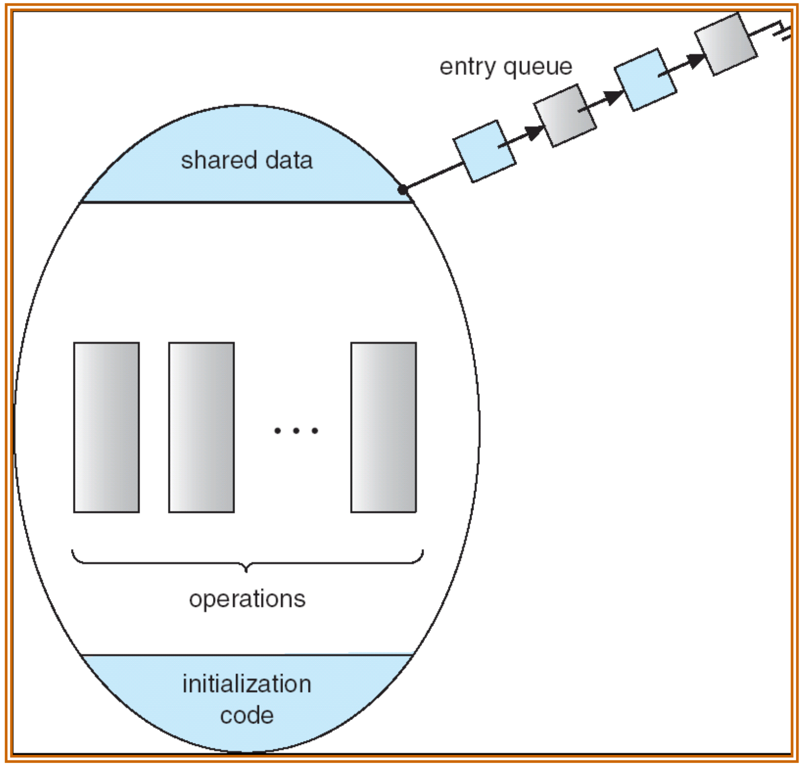
- ?????

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…



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_mu
int pthread_cond_signal(pthread_
int pthread_cond_broadcast(pthrea
```

Java:

`synchronized keyword`

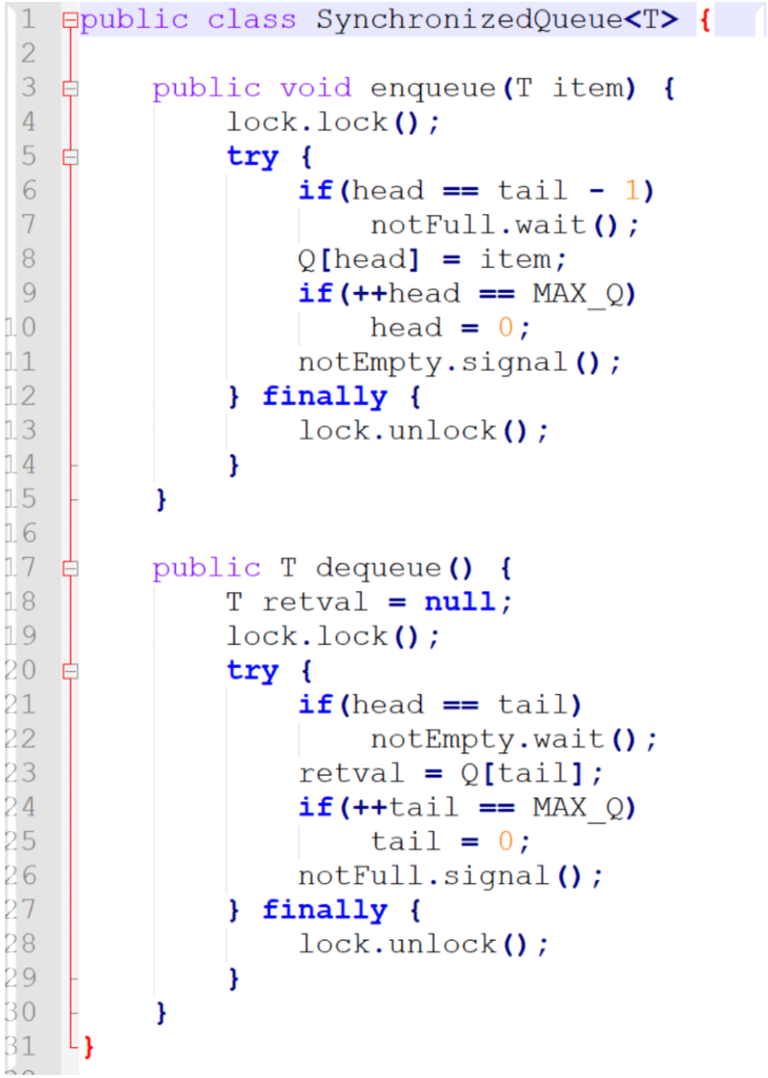
`wait()/notify()/notifyAll()`

C#: Monitor class

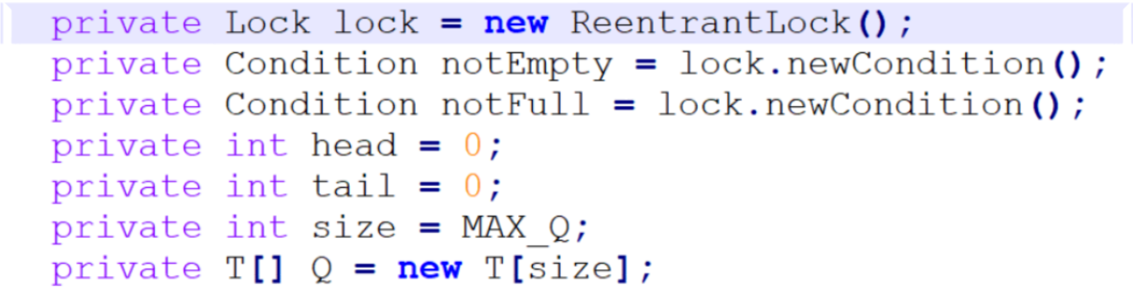
`Enter()/Exit()/`

`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
```

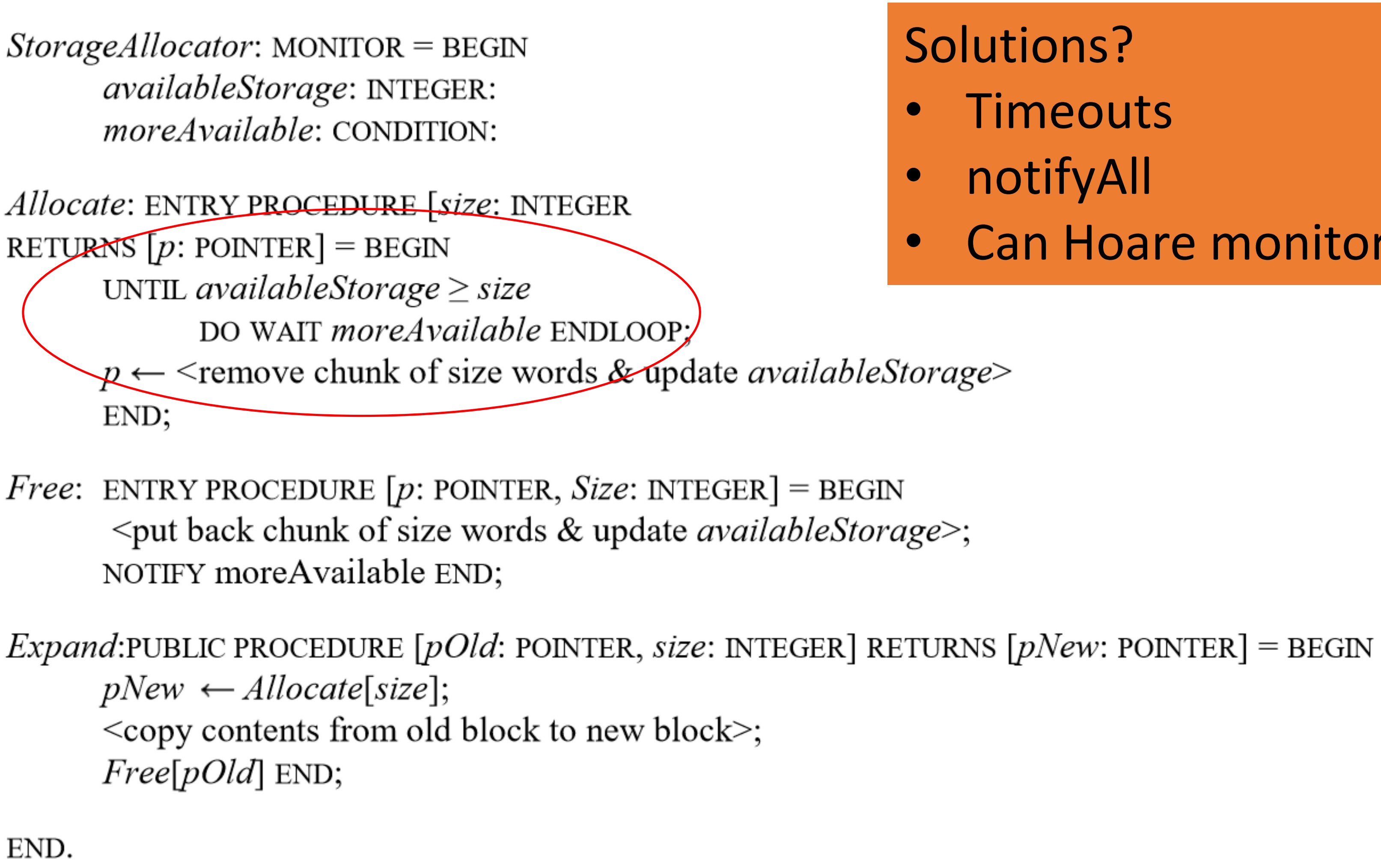
```
schedule:
    if e.any()
        t ← e.pop_front
        t. run
    else
        unlock
```

```
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?



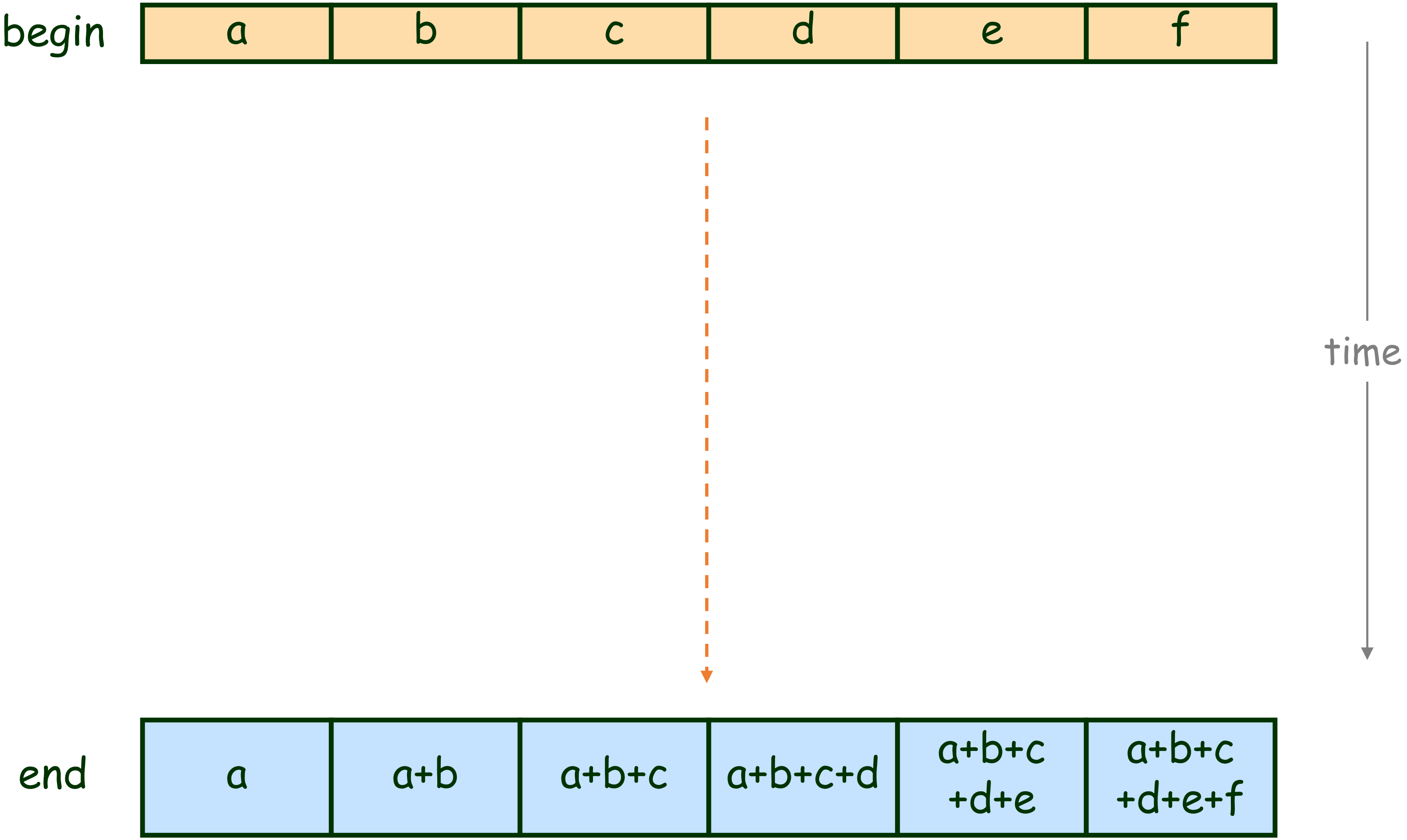
```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

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?

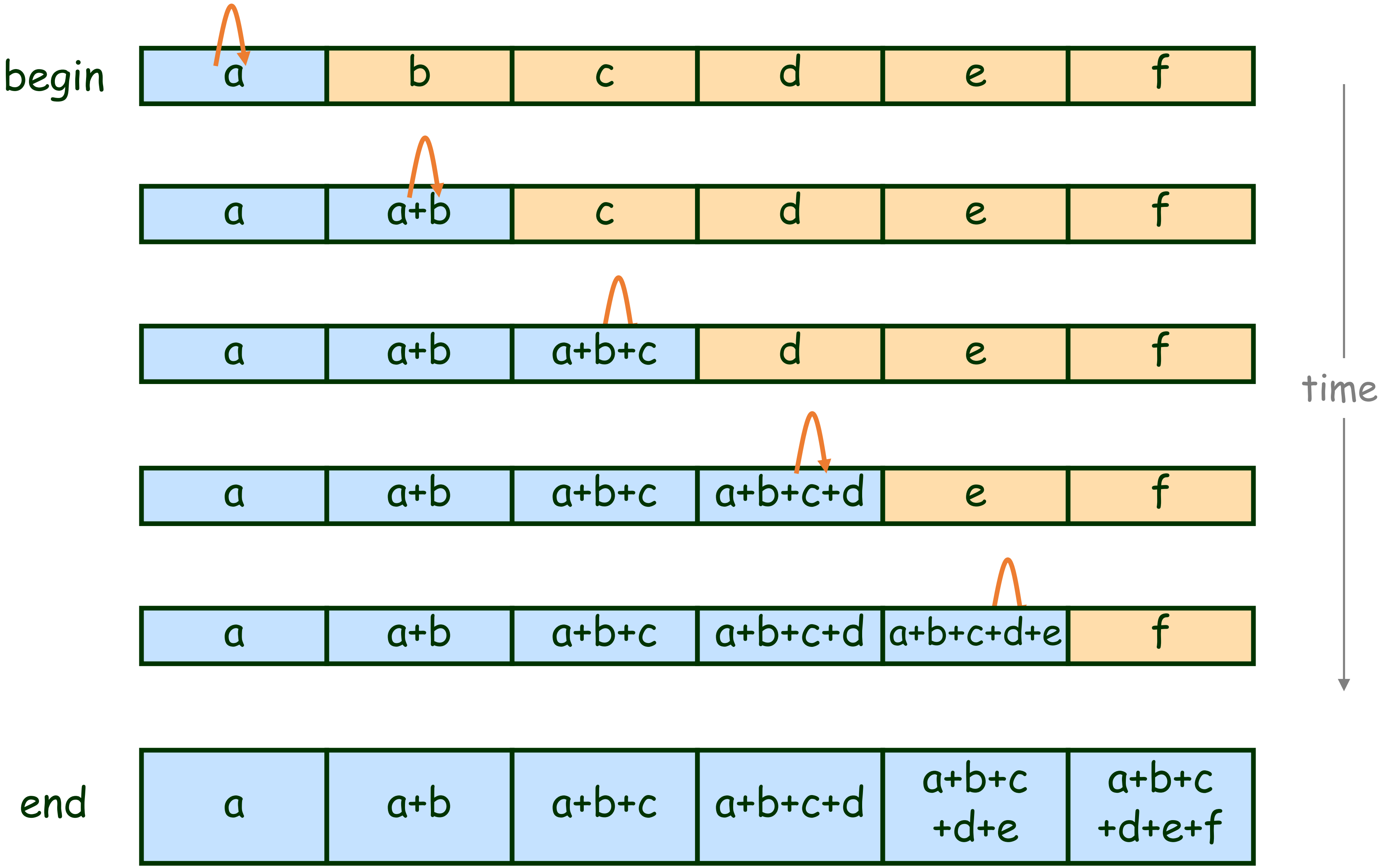
Barriers

Prefix Sum



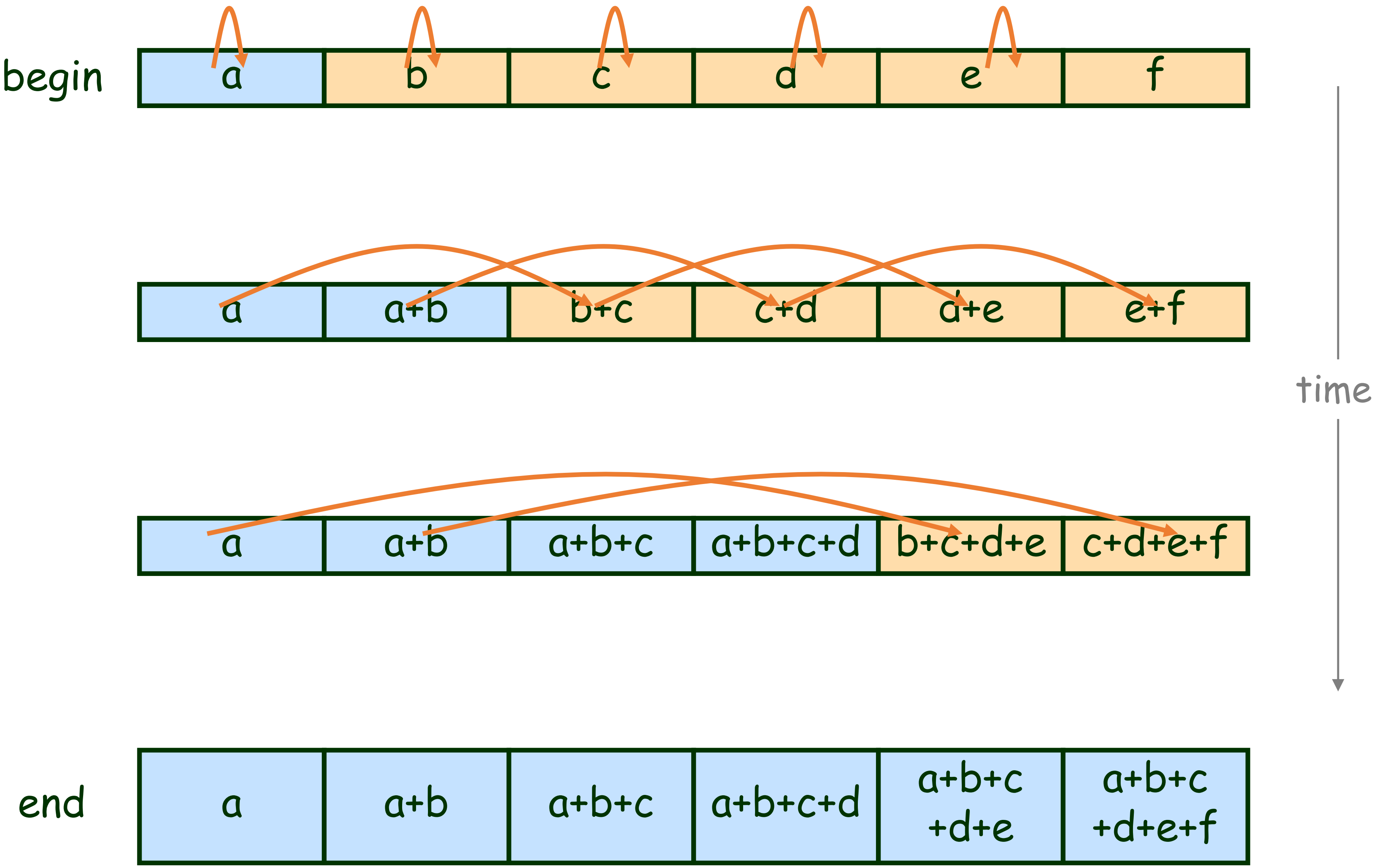
begin	a	b	c	d	e	f
end	a	a+b	a+b+c	a+b+c+d	a+b+c +d+e	a+b+c +d+e+f

Prefix Sum



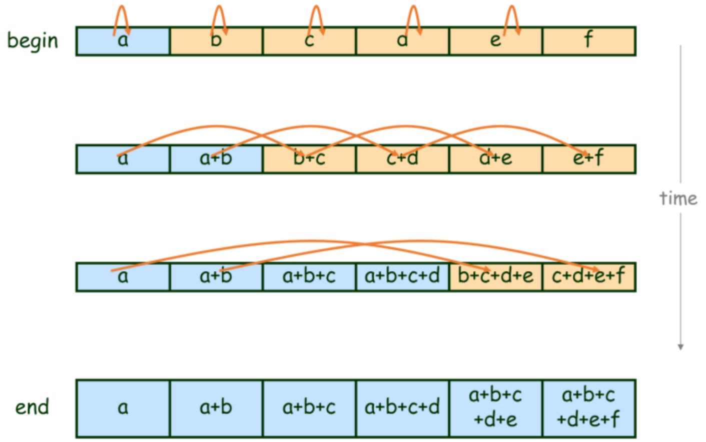
begin	a	b	c	d	e	f
	a	a+b	c	d	e	f
	a	a+b	a+b+c	d	e	f
	a	a+b	a+b+c	a+b+c+d	e	f
	a	a+b	a+b+c	a+b+c+d	a+b+c+d+e	f
end	a	a+b	a+b+c	a+b+c+d	a+b+c +d+e	a+b+c +d+e+f

Parallel Prefix Sum



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

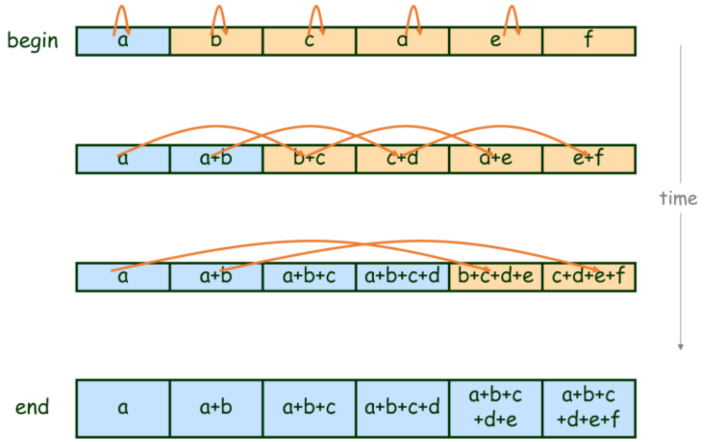
Pthreads Parallel Prefix Sum



```
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?

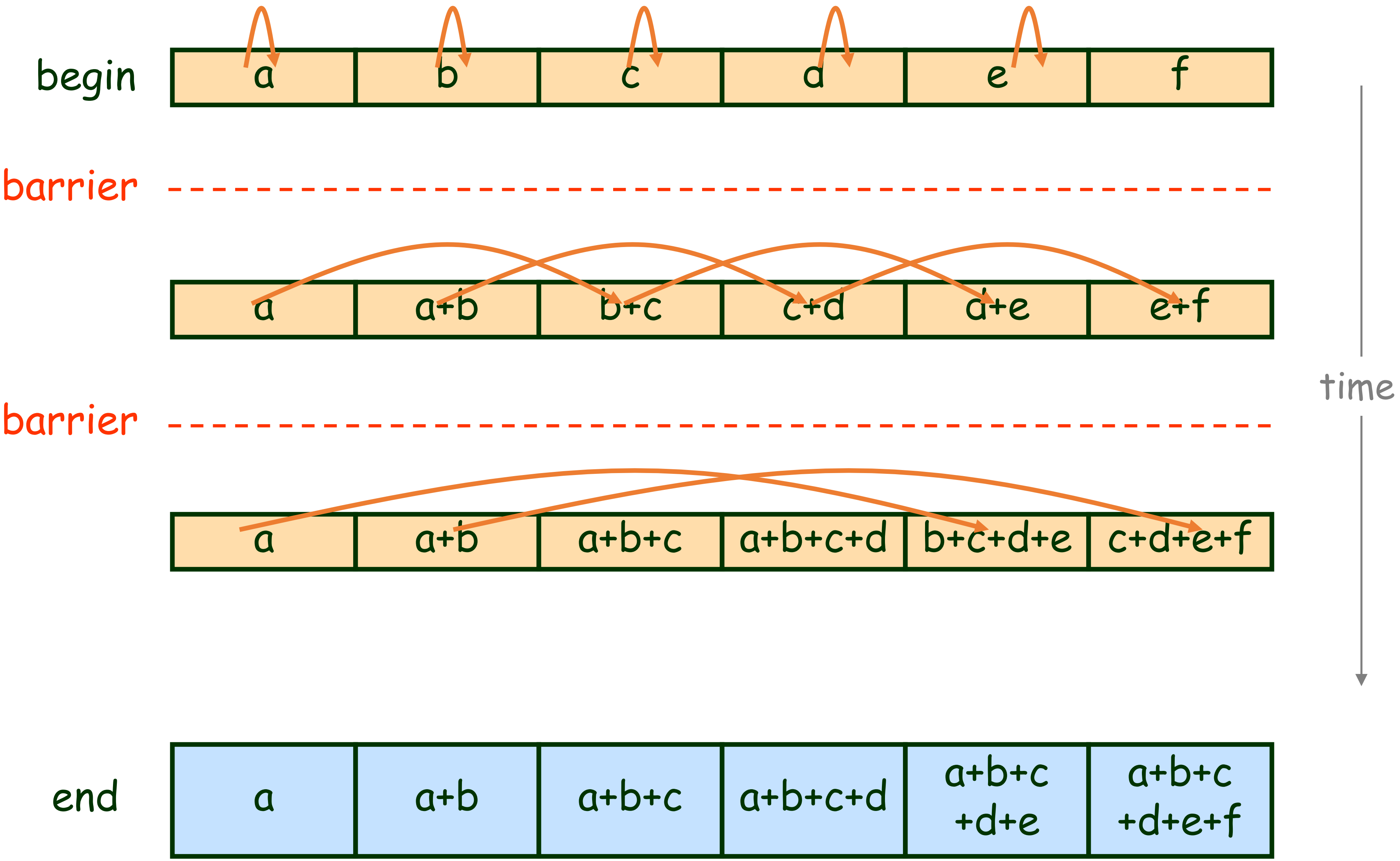
Pthreads Parallel Prefix Sum



```
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]);
  }

}
```

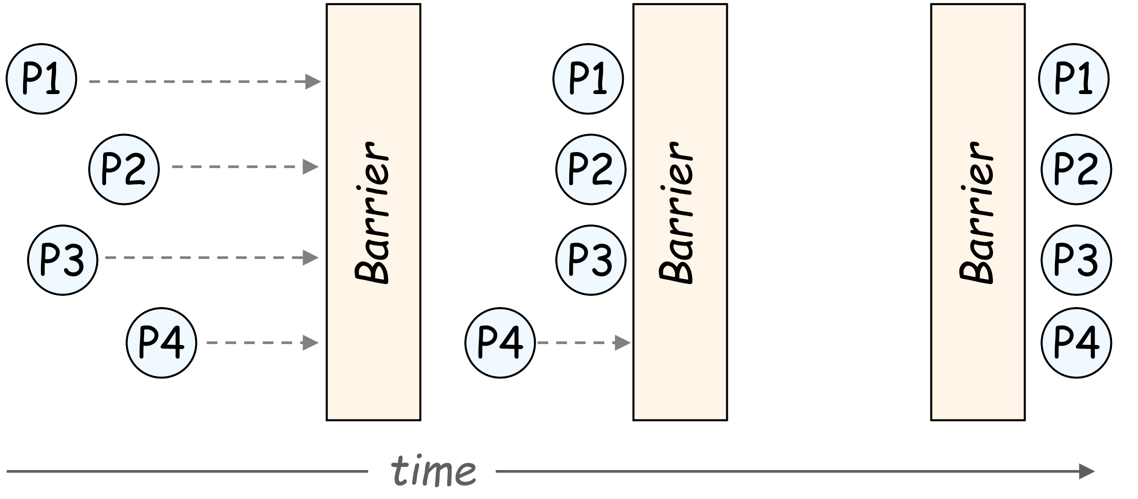
fixed?

Parallel Prefix Sum



What is a Barrier ?

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



four processes approach the barrier

all except P4 arrive

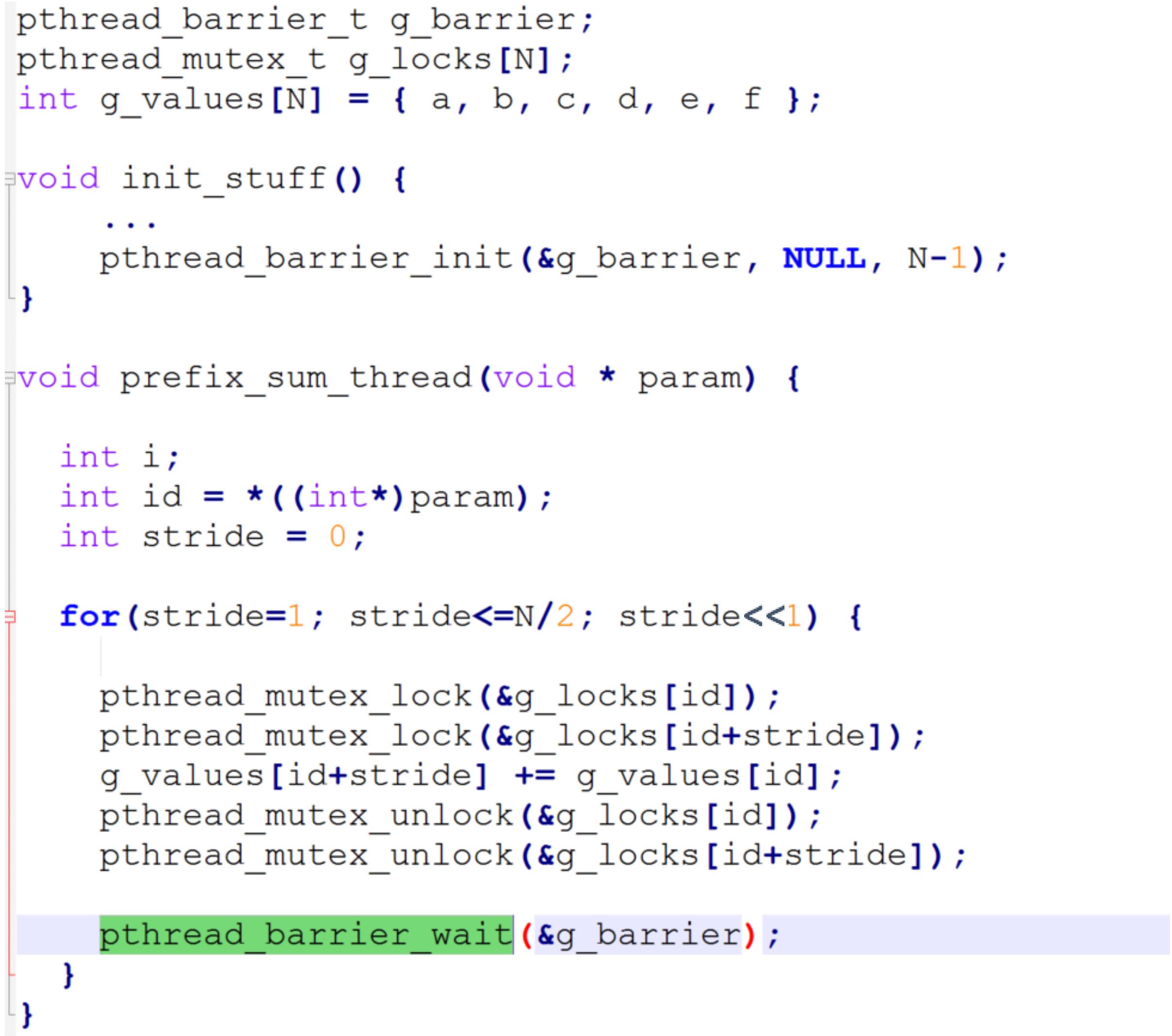
Once all arrive, they continue

Pthreads and barriers

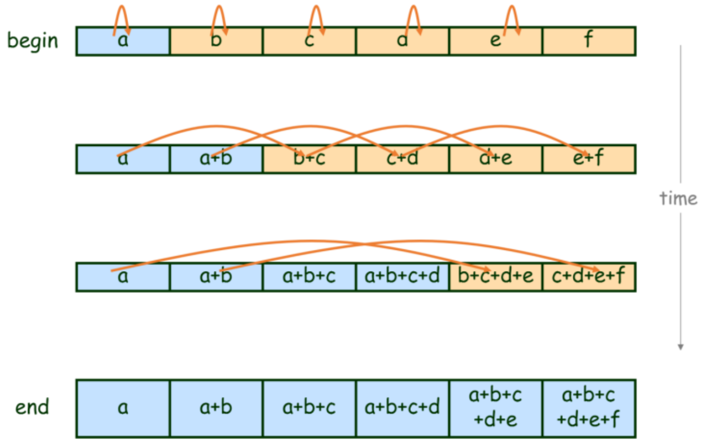
Type `pthread_barrier_t`

```
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
- Low shared memory references per process
- No need for shared memory initialization
- Symmetric: same amount of work for all processes
- Algorithm simplicity
- Simple basic primitive
- Minimal propagation time
- Reusability of the barrier (must!)

Barrier Building Blocks

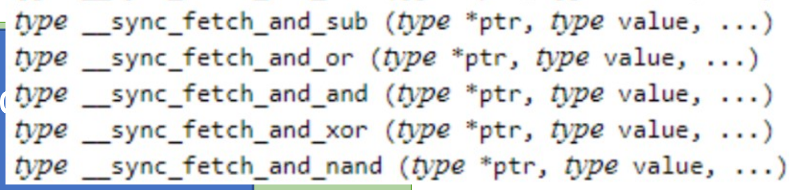
- Conditions
- Semaphores
- Atomic Bit
- Atomic Register
- Fetch-and-increment register
- Test and set bits
- Read-Modify-Write register

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
```

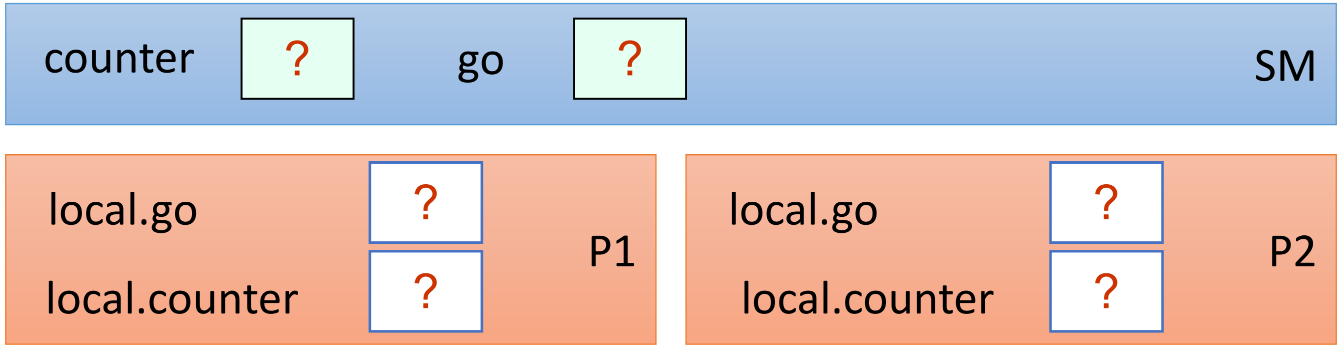
Simple Barrier Using an Atomic Counter

shared counter: fetch and increment reg. – {0,..n}, initially = 0
go: atomic bit, initial value is immaterial
local local.go: a bit, initial value is immaterial
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



counter ? go ? SM

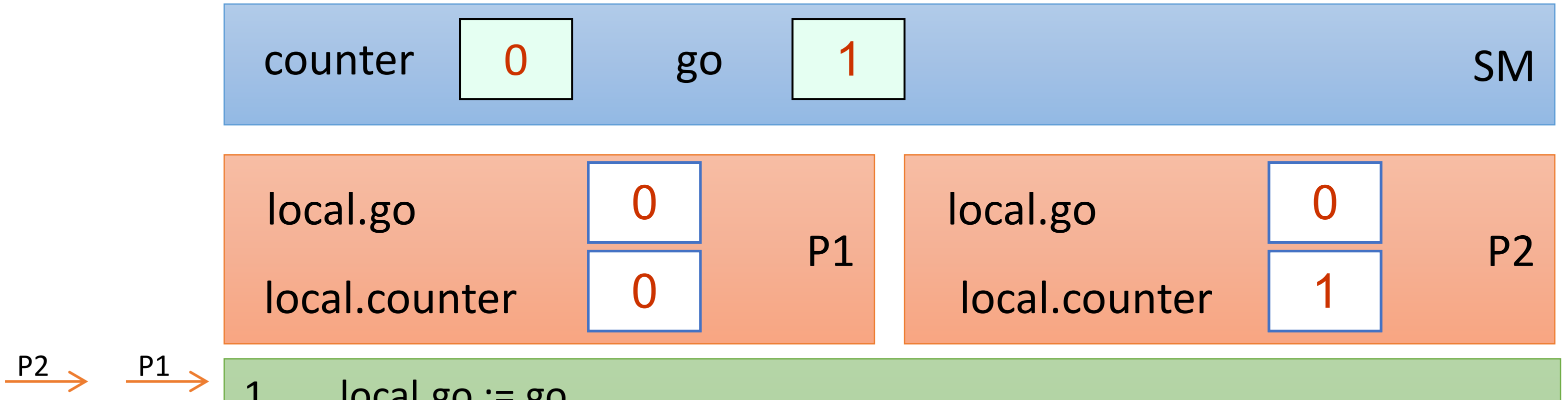
local.go ?
local.counter ?
P1

local.go ?
local.counter ?
P2

```
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
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

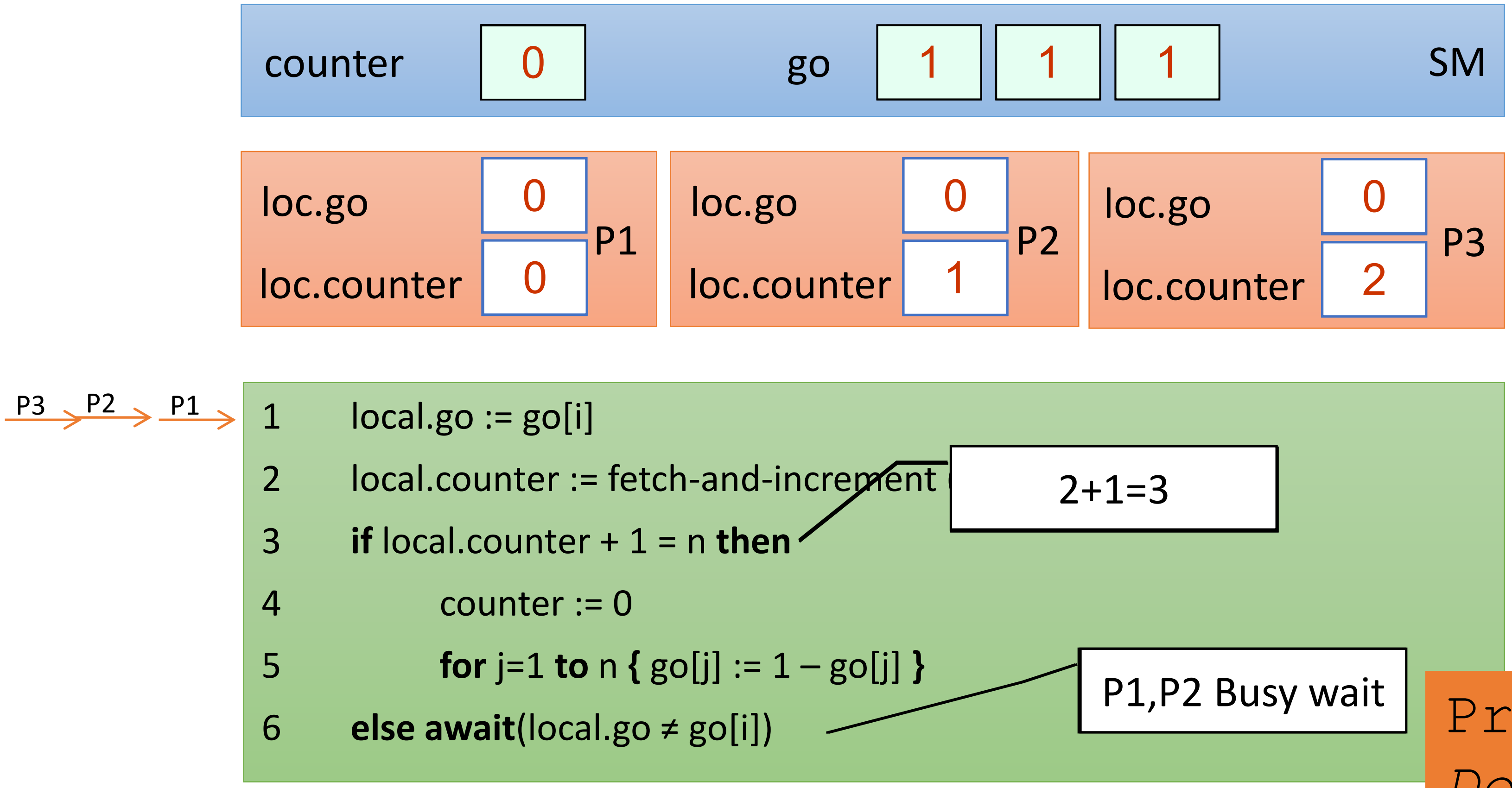
Program of a Thread i

```
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



```
local.go := go[i]
local.counter := fetch-and-increment
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 (*)

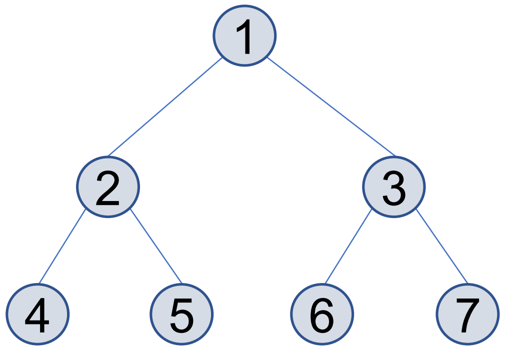
Simple Barrier with go array

- 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

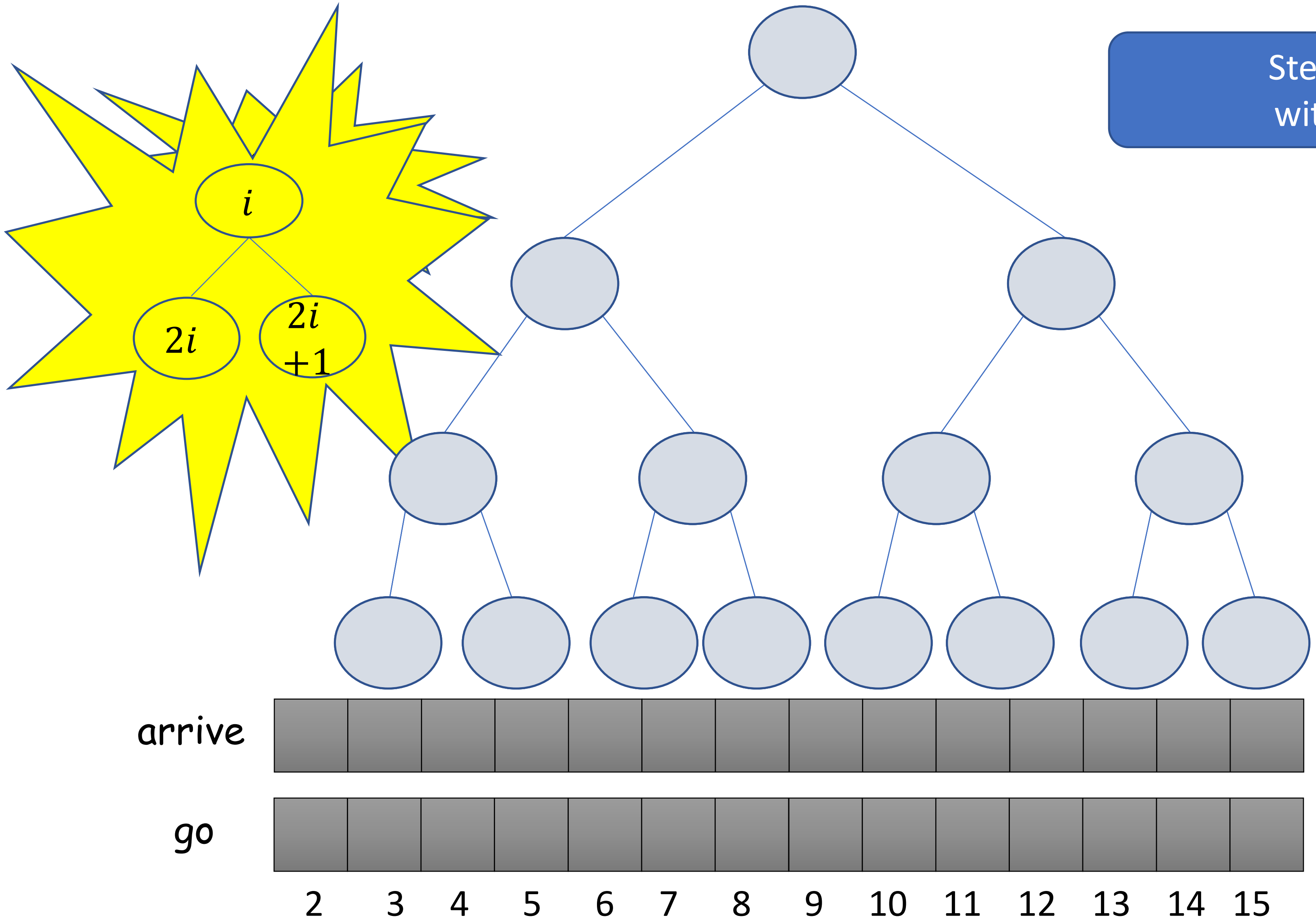
Tree Barriers

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
```

Root

Internal

Leaf

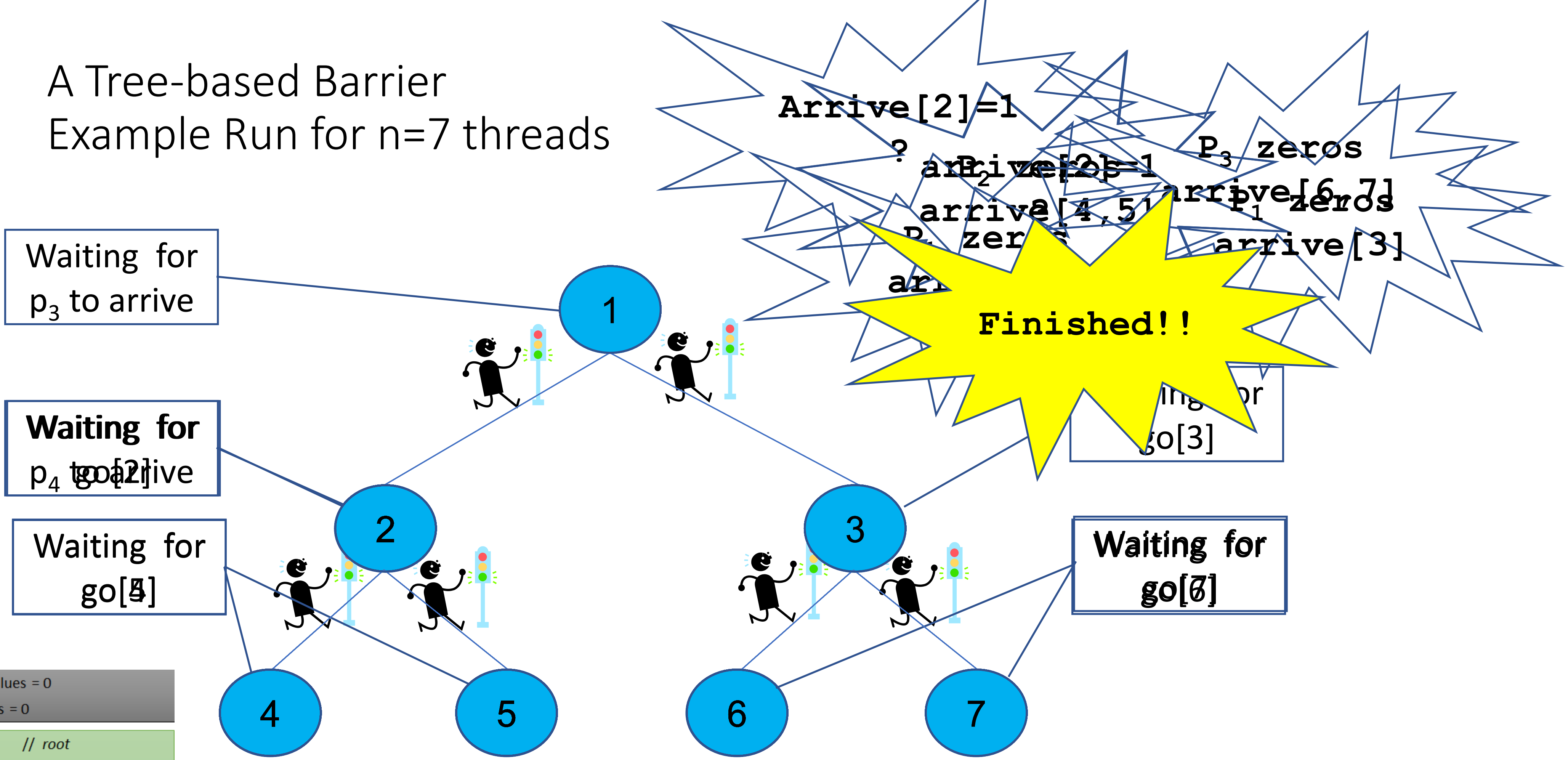
```
if i=1 then                                              // root
        await(arrive[2] = 1); arrive[2] := 0
        await(arrive[3] = 1); arrive[3] := 0
        go[2] = 1; go[3] = 1
else if i ≤ (n-1)/2 then                                 // internal node
        await(arrive[2i] = 1); arrive[2i]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

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

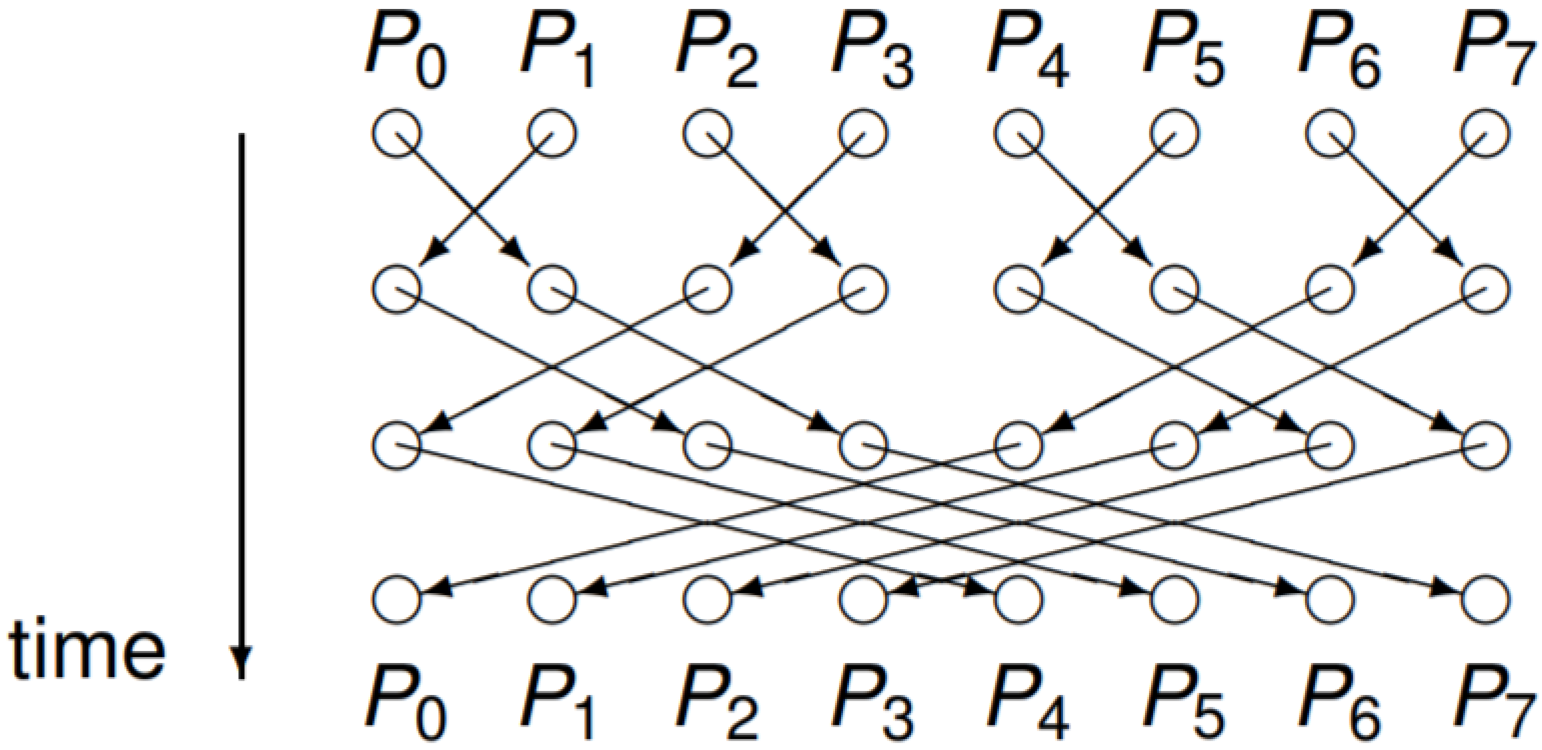
At this point all non-root threads in some await(go) case

Tree Barrier Tradeoffs

- Pros:

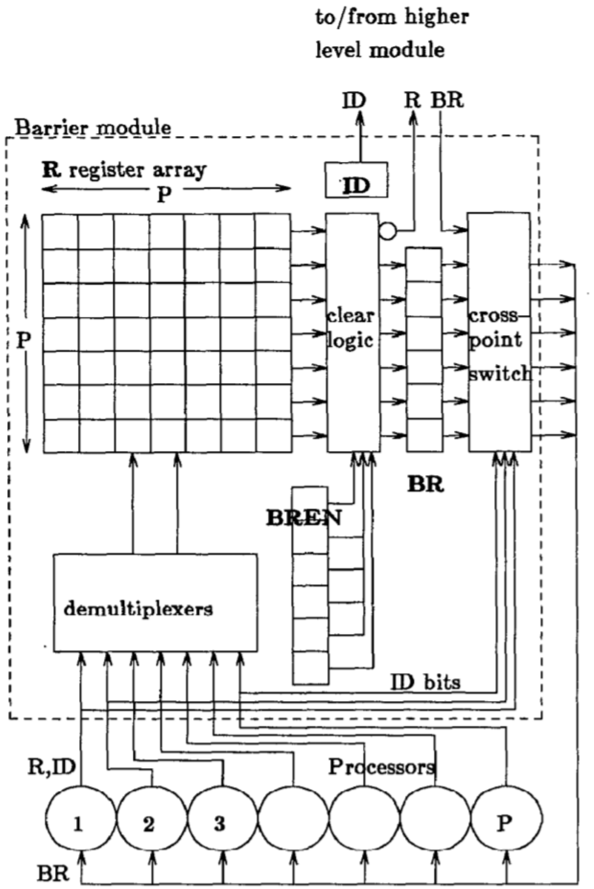
- Cons:

Butterfly Barrier

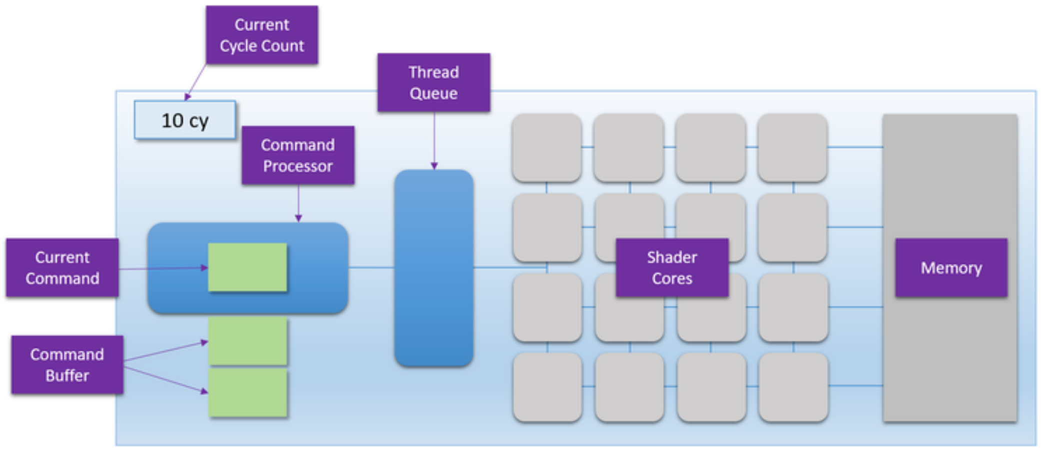


- When would this be preferable?

Hardware Supported Barriers



CPU



GPU

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?