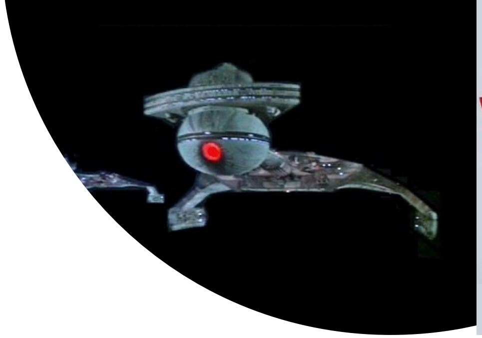
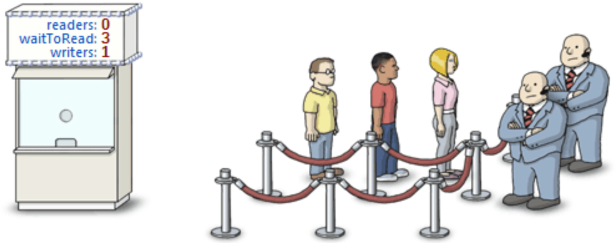
Synchronization: Monitors, Barriers

Chris Rossbach

Today



- Questions?
- Administrivia
  - Lab 1 due date moved LAST YEAR, AND just wanted to bring it up for nostalgia. (haha)
  - Lab 1 not actually moved at all. Due tonight!
  - Start looking at Lab 2 anyway, esp if you’re done with Lab 1
- Material for the day
  - Coherence redux
  - Some thoughts on work efficiency and instrumentation
  - Monitors
  - Barriers

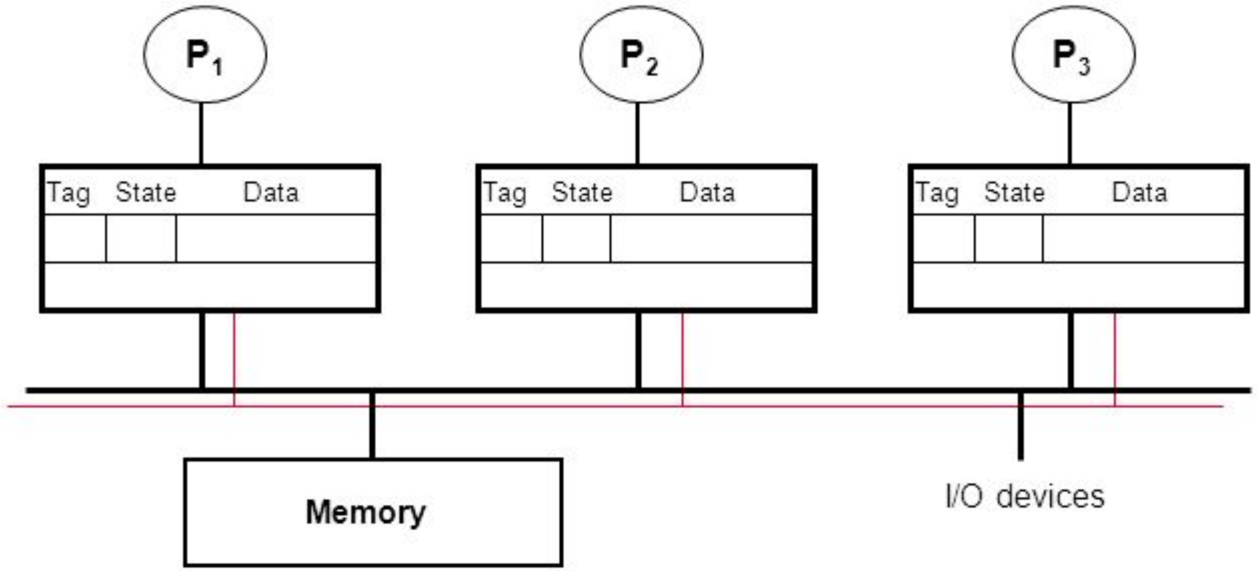


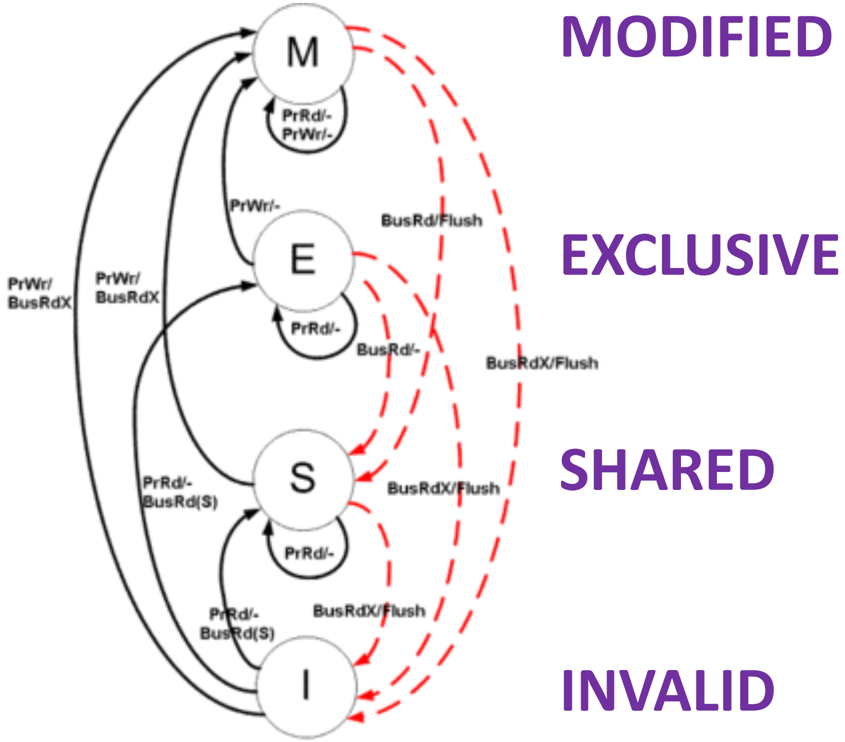
- Acknowledgements
  - Thanks to Gadi Taubenfield: I borrowed and modified some of his slides on barriers
- Image credits
  - https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjxi4uip8LdAhWFq1MKHbBeD4sQjRx6BAgBEAU&url=http%3A%2F%2Fpreshing.com%2F20150316%2Fsemaphores-are-surprisingly-versatile&psig=AOvVaw20Zw2eU9WAmbX8qxDSLSRd&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?

Review: Basic MESI Cache Coherence





Each cache line has a state (M, E, S, I)

- Processors “snoop” bus to maintain states
- Initially → ‘I’ → Invalid
- Read one → ‘E’ → exclusive
- Reads → ‘S’ → multiple copies possible
- Write → ‘M’ → single copy → lots of cache coherence traffic

How can we improve over busy-wait?

```
Lock::Acquire() {
while(1) {
  while (*lock == 1) ; // spin just reading
  if (test&set(lock) == 0)  break;
}
```

Mutex

- Same abstraction as spinlock
- But is a “blocking” primitive
  - Lock available → same behavior
  - Lock held → yield/block
- Many ways to yield
- Simplest case of semaphore

```
void cm3_lock(u8_t* M) {
  u8_t LockedIn = 0;
  do {
   if (__LDREXB(Mutex) == 0) {
     // unlocked: try to obtain lock
     if ( __STREXB(1, Mutex)) { // got lock
       __CLREX(); // remove __LDREXB() lock
       LockedIn = 1;
     }
     else task_yield(); // give away cpu
   }
   else task_yield();    // give away cpu
} while(!LockedIn);
```

- Is it better to use a spinlock or mutex on a uni-processor?
- Is it better to use a spinlock or mutex on a multi-processor?
- How do you choose between spinlock/mutex on a multi-processor?

futex: Fast Userspace Mutex

```
int futex(int *uaddr, int futex_op, int val,
          const struct timespec *timeout );
```

uaddr points to a 32-bit value in user space

futex_op

- FUTEX_WAIT – if val == *uaddr sleep till FUTEX_WAKE
- FUTEX_WAKE – wake up at most val waiting threads

timeout

- *timespec* structure to specify a timeout for the op

- Interface to the kernel sleep()
- Let thread deschedule itself – conditionally!
- Can be used to implement locks, semaphores, monitors, etc…

Test&Set and `futex`

```
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);
```

```
acquire(int *thelock) {
   while (test&set(thelock)) {
      futex(thelock, FUTEX_WAIT, 1);
   }
}
```

```
release(int *thelock) {
   thelock = 0; // unlock
   futex(&thelock, FUTEX_WAKE, 1);

}
```

- Properties:
  - Sleep interface by using futex – no busywaiting
- Pros: low overhead to acquire lock
- Cons:
  - Unlock calls kernel to potentially wake someone up – even if none
  - Ideally, we have no-kernel crossings when uncontended

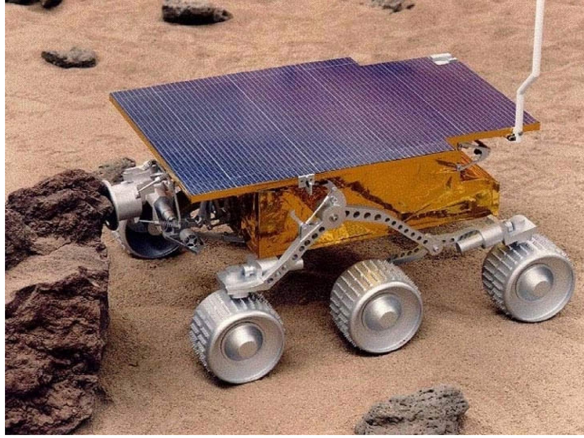
Improved Test&Set with futex

```
bool maybe_waiters = false;
int mylock = 0; // Interface: acquire(&mylock,&maybe_waiters);
                //            release(&mylock,&maybe_waiters);

acquire(int *thelock, bool *maybe) {
   while (test&set(thelock)) {
      // Sleep, since lock busy!
      *maybe = true;
      futex(thelock, FUTEX_WAIT, 1);

      // Make sure other sleepers not stuck
      *maybe = true;
   }
}
```

```
release(int*thelock, bool *maybe) {
   thelock = 0;
   if (*maybe) {
      *maybe = false;
      // Try to wake up someone
      futex(&value, FUTEX_WAKE, 1);
   }
}
```

- Pros: syscall-free in the uncontended case
  - Uses syscalls if multiple waiters, or concurrent acquire/release
- But it can be considerably optimized!
  - See "Futexes are Tricky" by Ulrich Drepper

Lock Pitfalls…



A(prio-0) → `lock(my_lock);`

B(prio-100) → `lock(my_lock);`

ACK! Priority Inversion!

Solution?

Priority inheritance: A runs at B's priority

MARS pathfinder failure:

http://wiki.csie.ncku.edu.tw/embedded/priority-inversion-on-Mars.pdf

Other ideas?

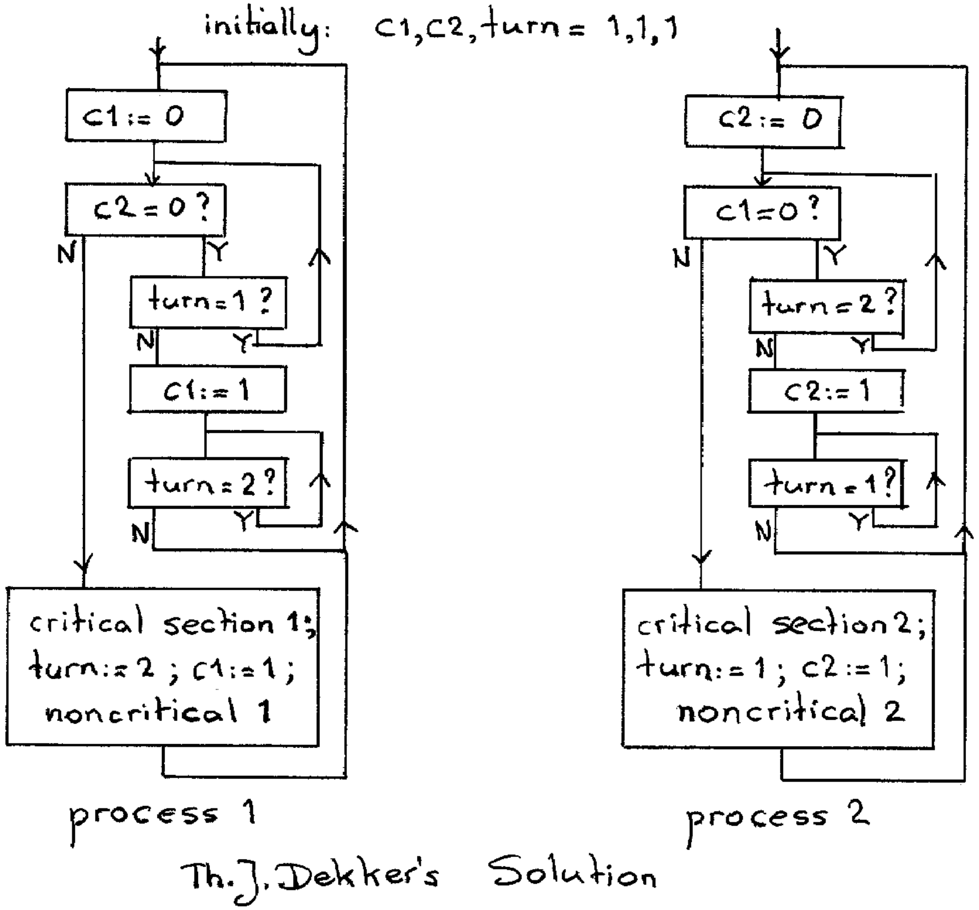
Can you build a lock without HW RMW?

Dekker's Algorithm

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

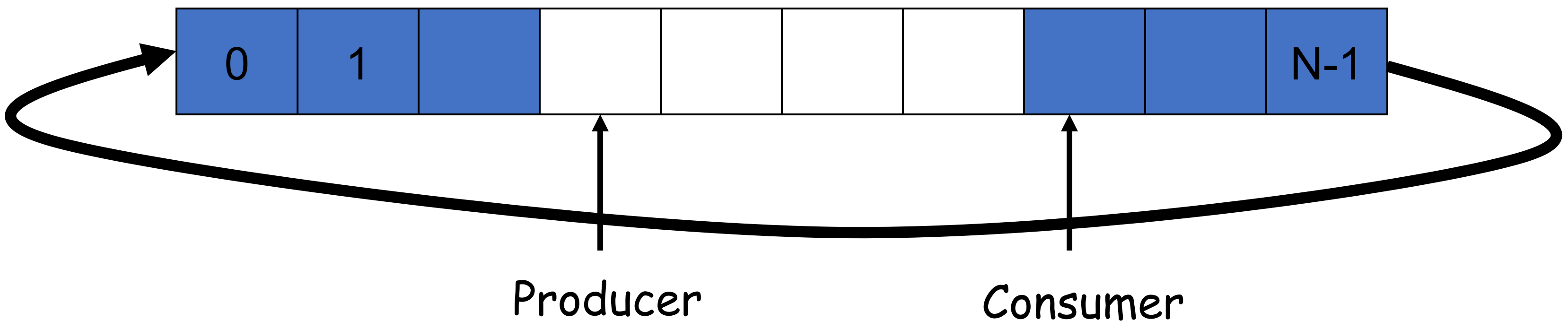
```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```



Producer-Consumer (Bounded-Buffer) Problem

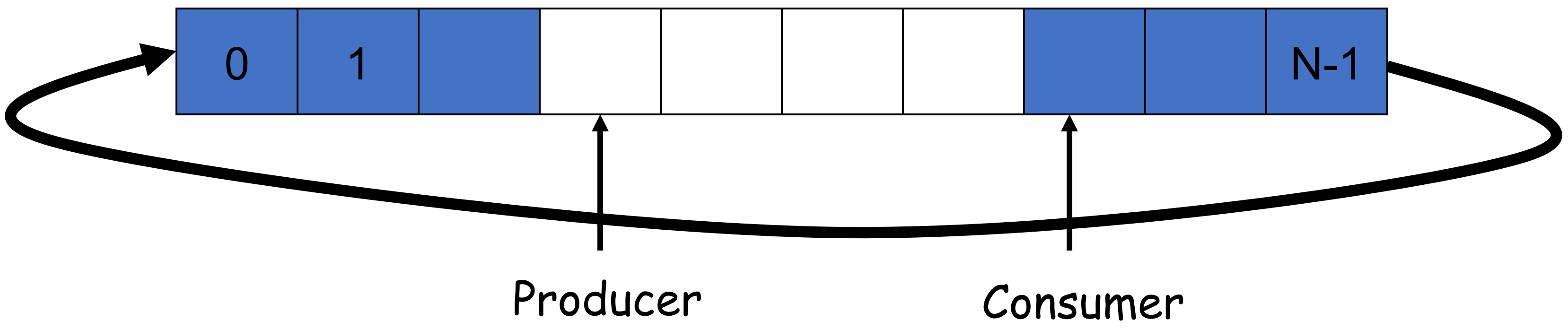
- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “consumes”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data



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

- 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?

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

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

```
//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

Is this correct?

- Problem: mutual exclusion?

```
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!
-
-

- Type: `pthread_semaphore_t`

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

- ?????

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

…y sem is greater
…e count

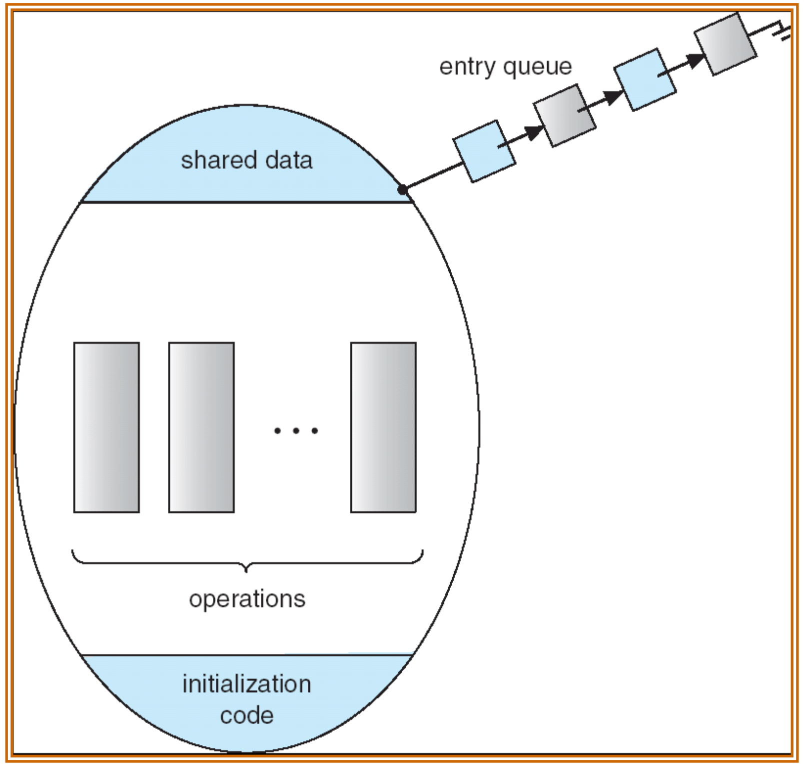
…phore pointed to
…he semaphore,

…**signed int**

…between threads
- else shared between processes

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/monitors

Why the pthread_mutex_t parameter for pthread_cond_wait?

- 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(pthre
```

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
```

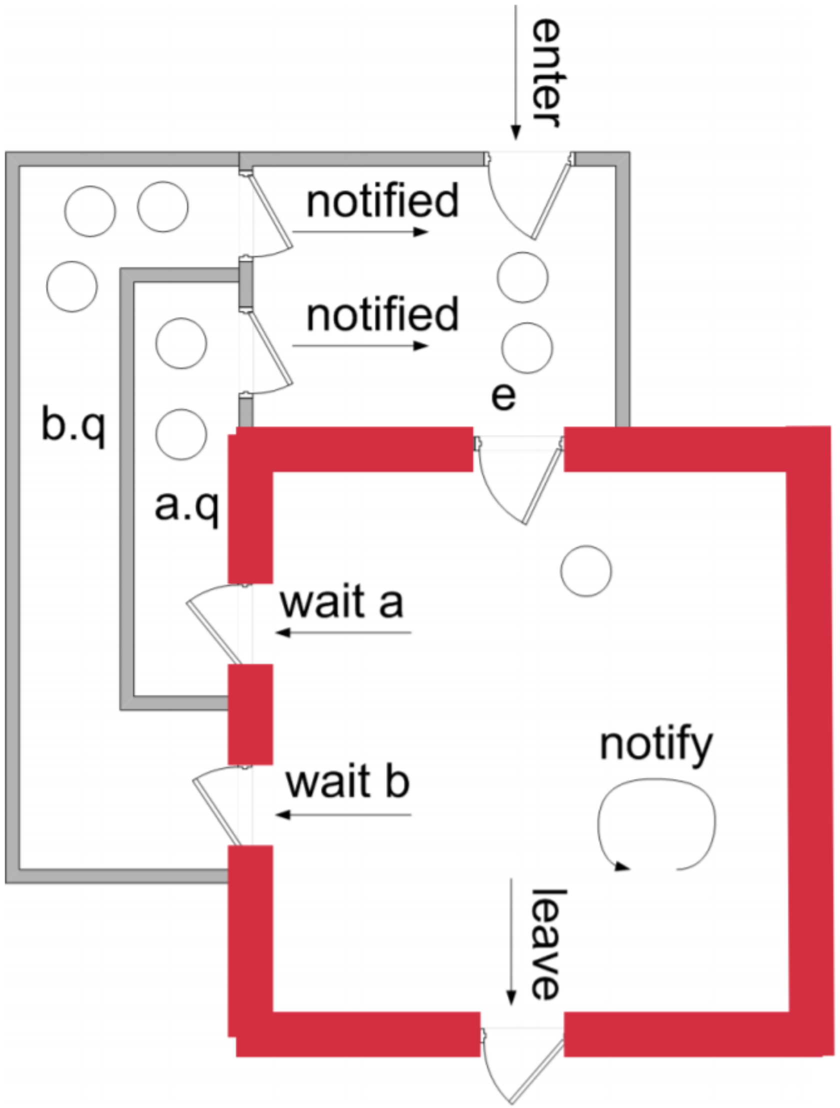
```
schedule:
    if e.any()
        t ← e.pop_front
        t. run
    else
        unlock
```

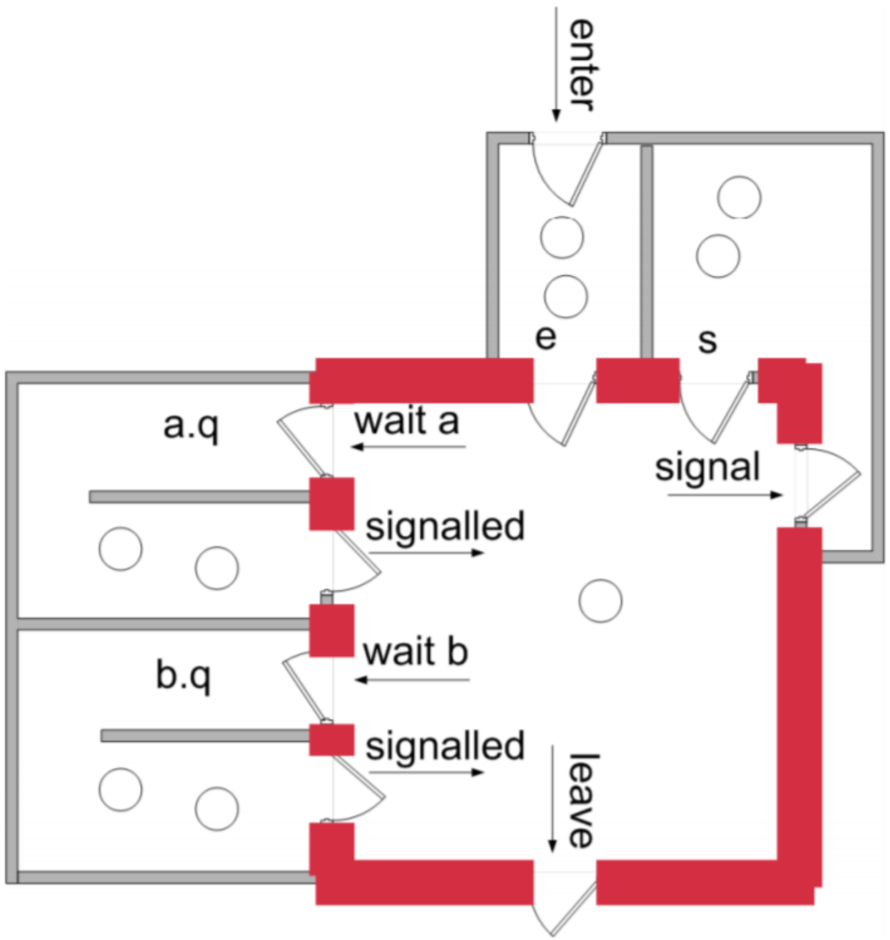
```
notify C:

    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?

Mesa, Hansen, Hoare





Example: anyone see a bug?

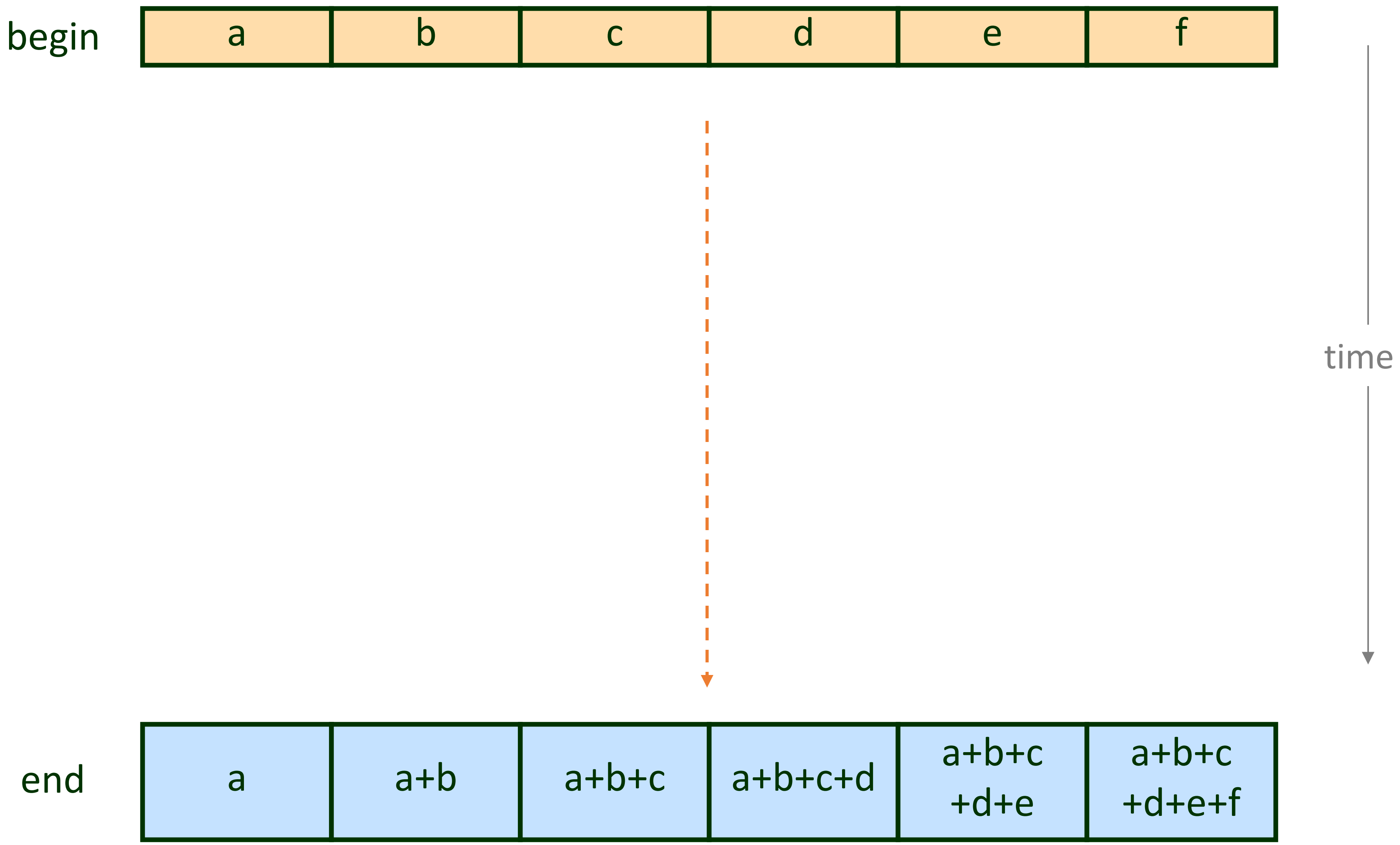
```
StorageAllocator: MONITOR = BEGIN
        availableStorage: INTEGER:
        moreAvailable: CONDITION:

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?

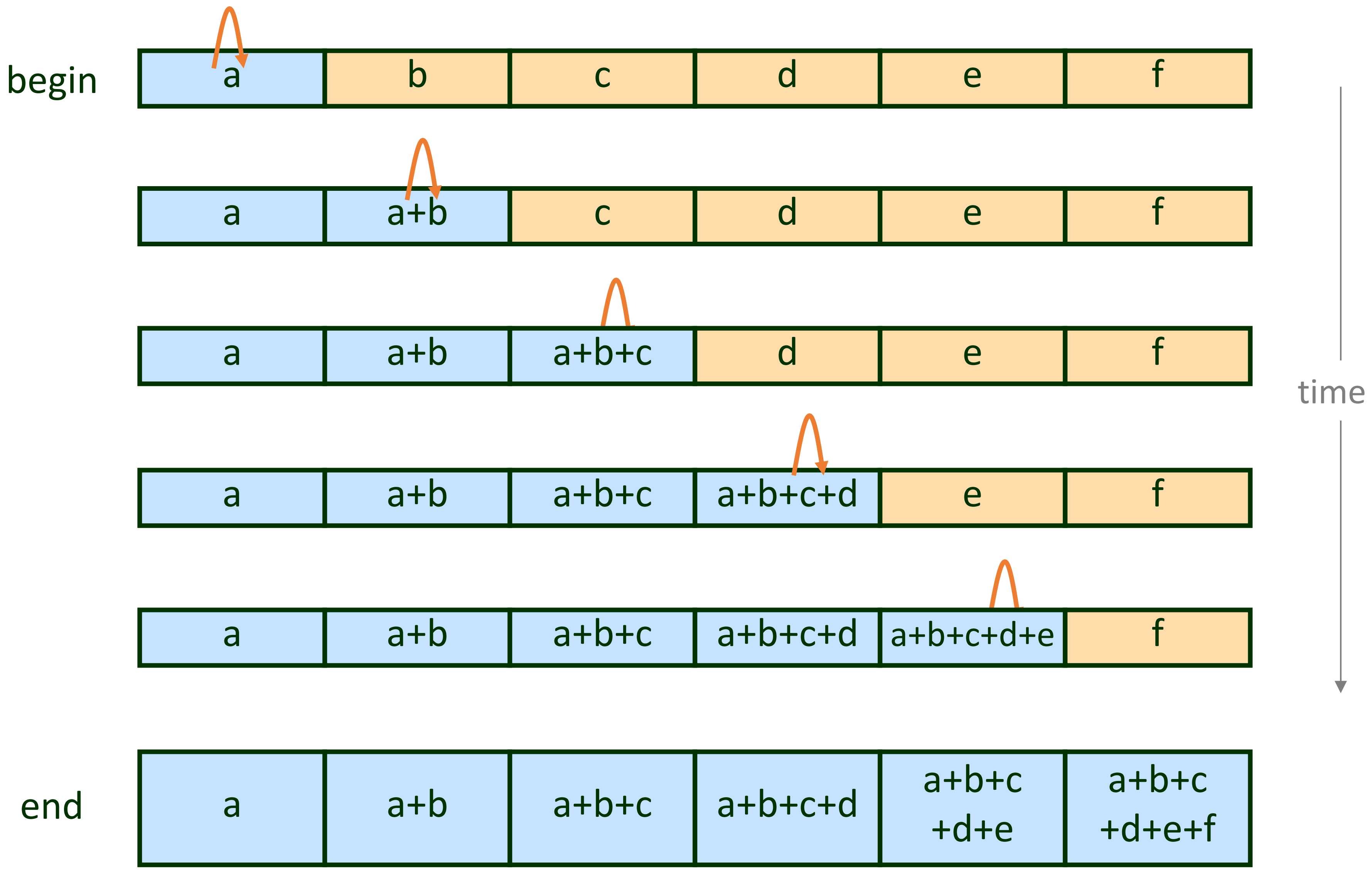
- Timeouts
- notifyAll
- Can Hoare monitors support notifyAll?

Barriers

Prefix Sum

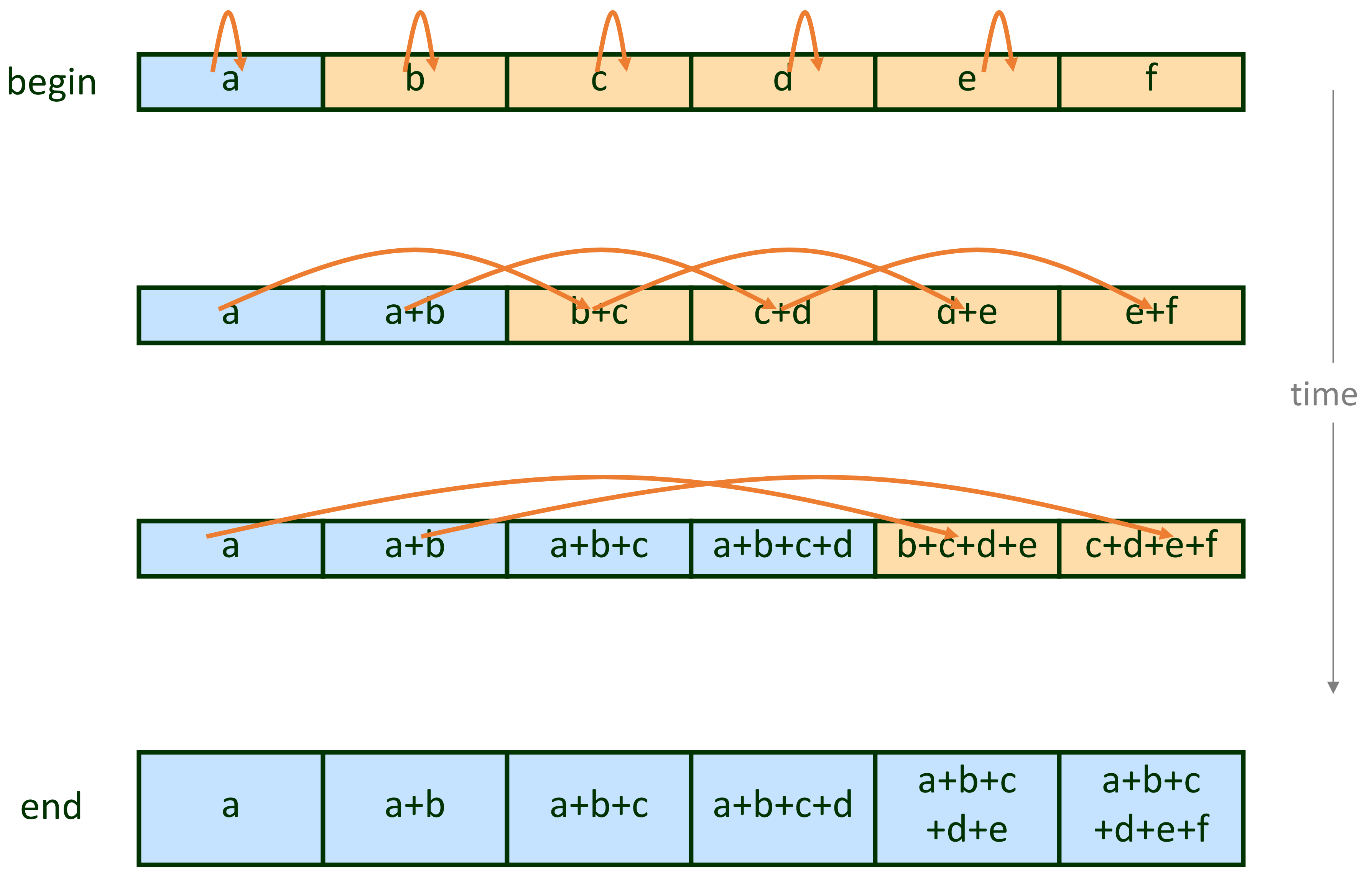


Prefix Sum

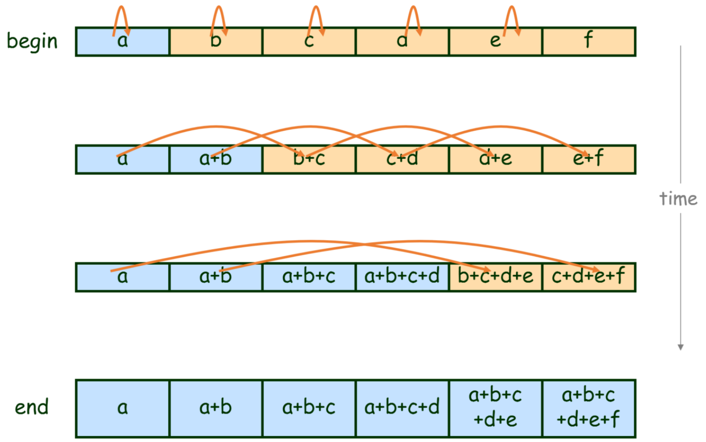


begin
a
b
c
d
e
f
a
a+b
c
d
e
f
a
a+b
a+b+c
d
e
f
a
a+b
a+b+c
a+b+c+d
e
f
a
a+b
a+b+c
a+b+c+d
a+b+c+d+e
f
end
a
a+b
a+b+c
a+b+c+d
a+b+c +d+e
a+b+c +d+e+f
time

Parallel Prefix Sum



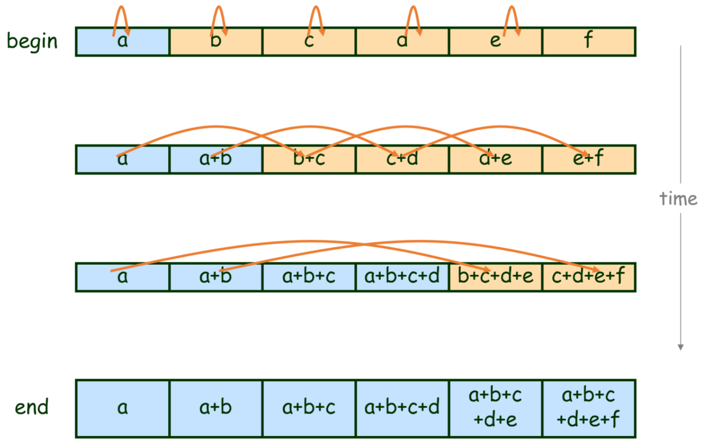
Pthreads Parallel Prefix Sum




```c
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?

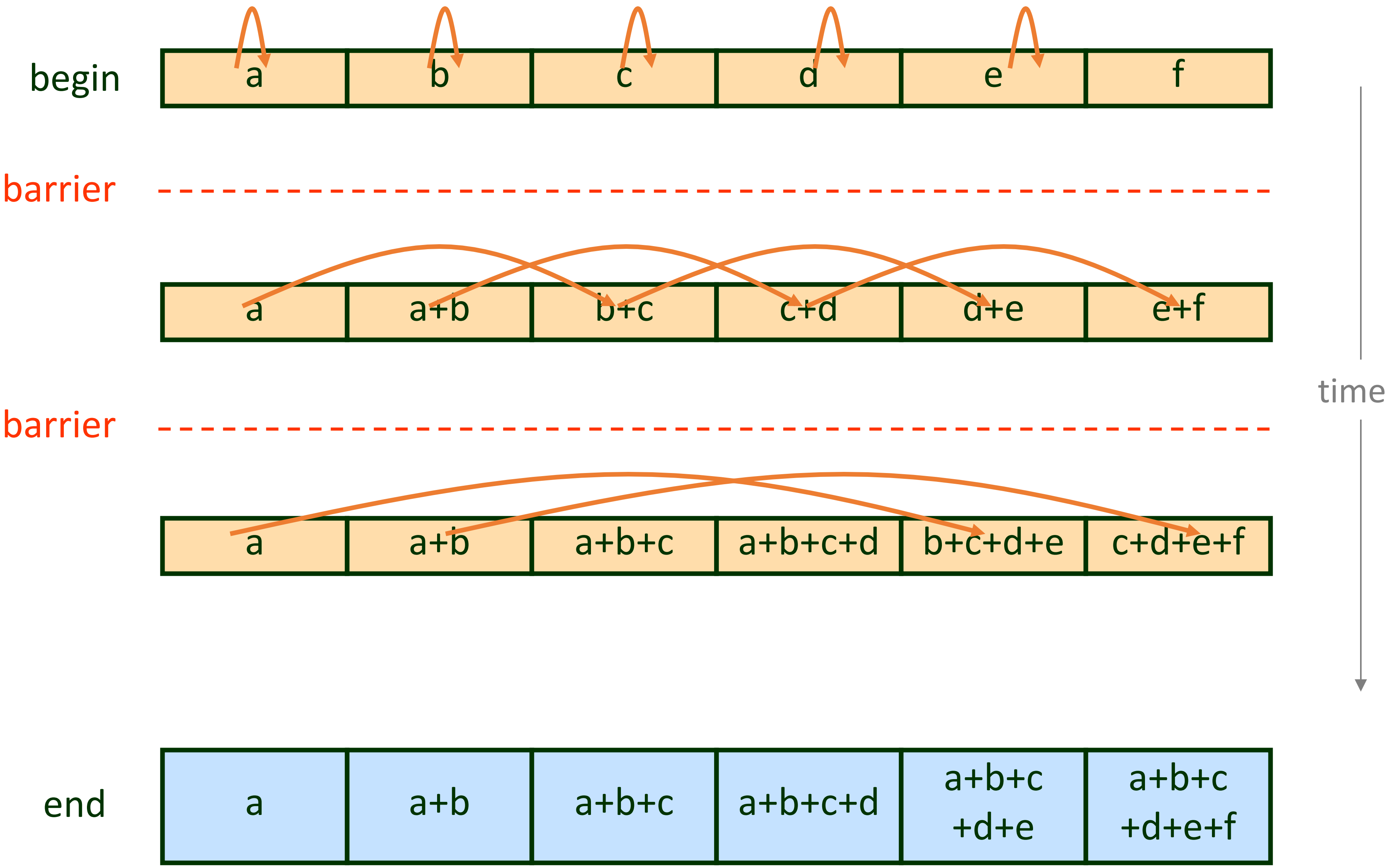
Pthreads Parallel Prefix Sum

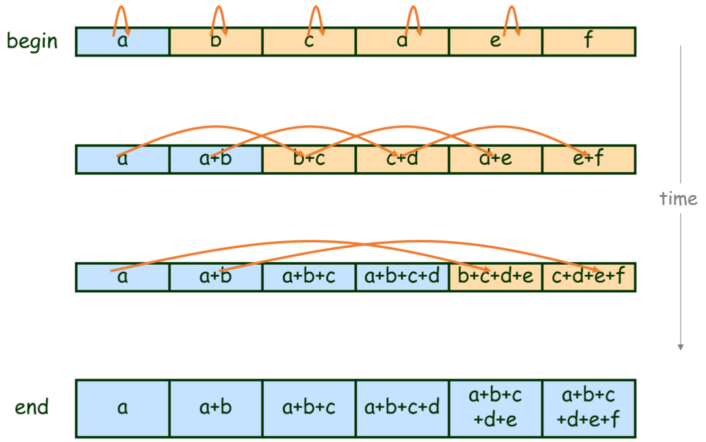



```
pthread_mutex_t g_locks[N] = { MUTEX_INITIALIZER, ...};
int g_values[N] = { a, b, c, d, e, f };

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





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

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

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
```

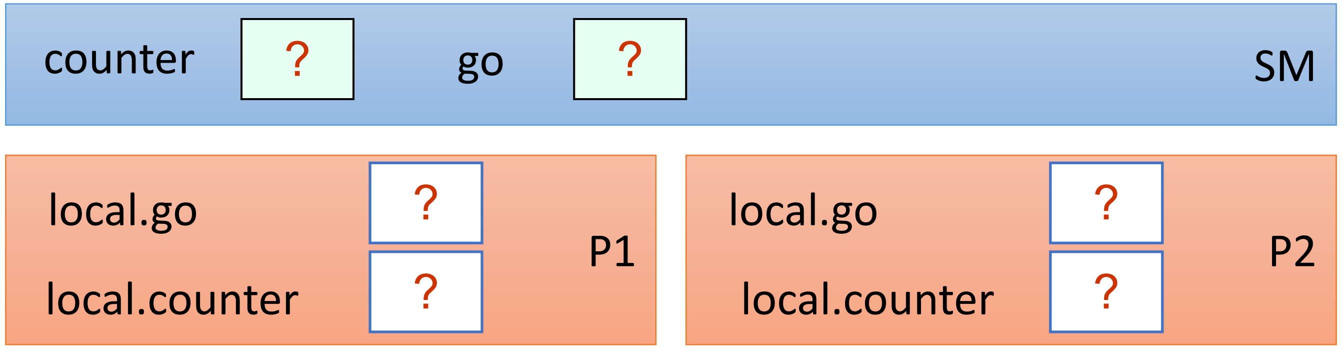
Simple Barrier Using an Atomic Counter

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




```
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

SM	
counter	0
go	1

P1		P2	
local.go	0	local.go	0
local.counter	0	local.counter	1

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

counter	0	go	1 1 1		SM

	loc.go	loc.counter
P1	0	0
P2	0	1
P3	0	2

P3 → P2 → P1 →

```
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 (*)

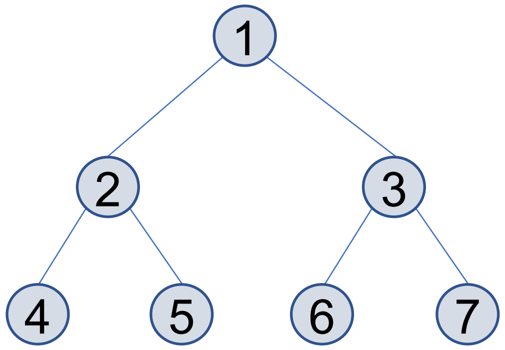
Simple Barrier with go array

- 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

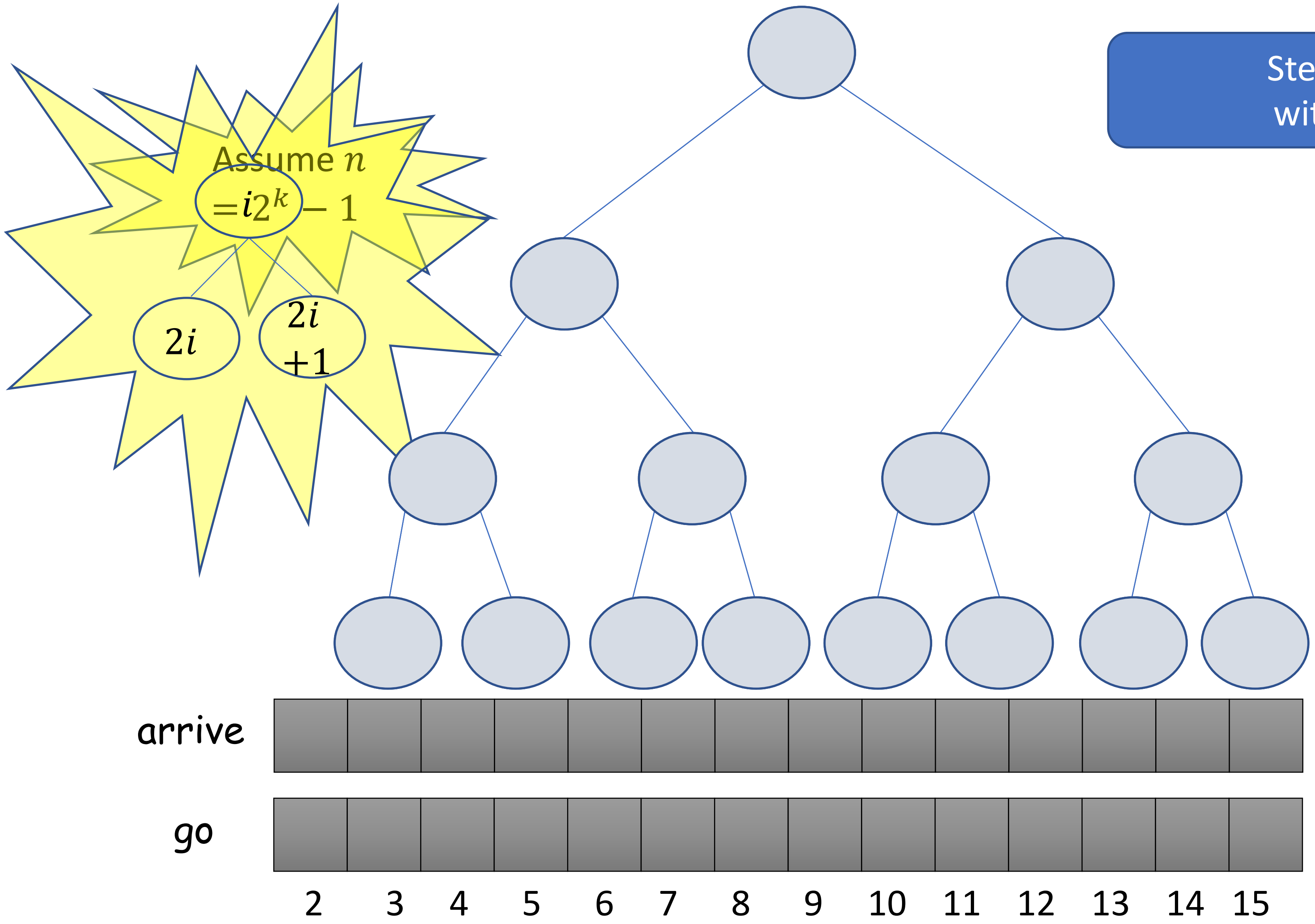
A Tree-based Barrier

- 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:

```
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
```

Internal:

```
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
```

Leaf:

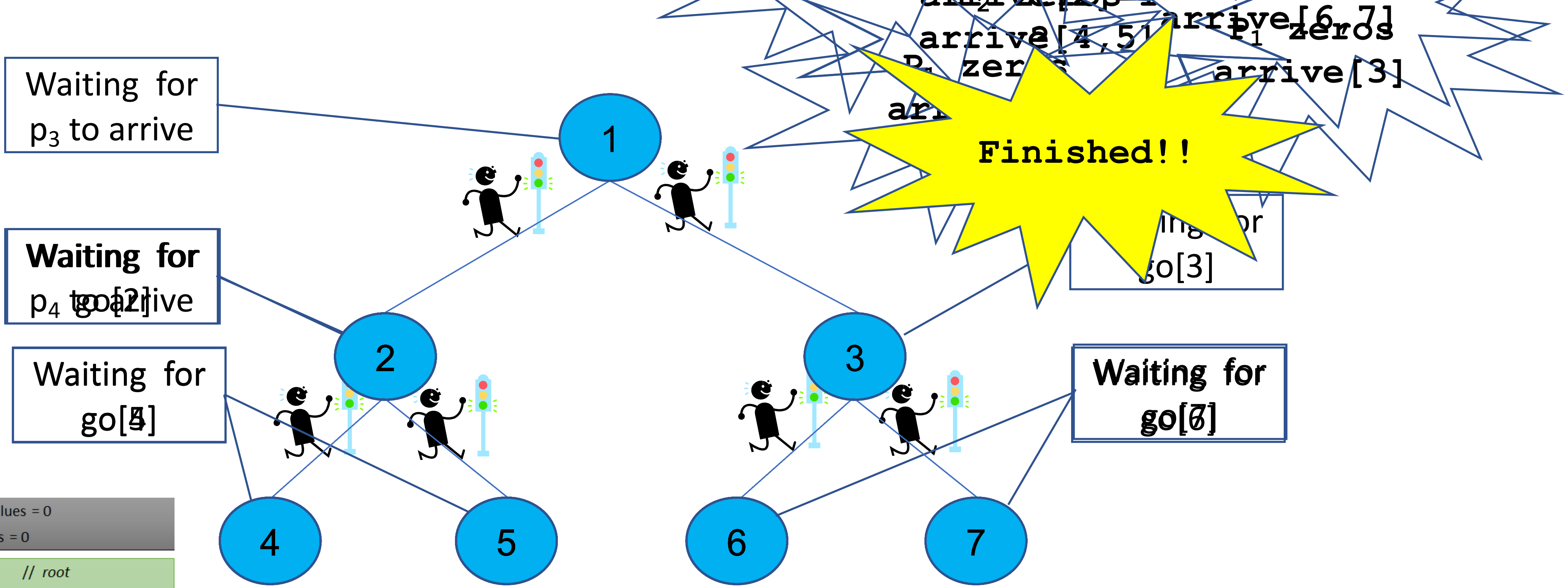
```
else                                               // leaf
    arrive[i] := 1
    await(go[i] = 1); go[i] := 0 fi
fi
```

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
```

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

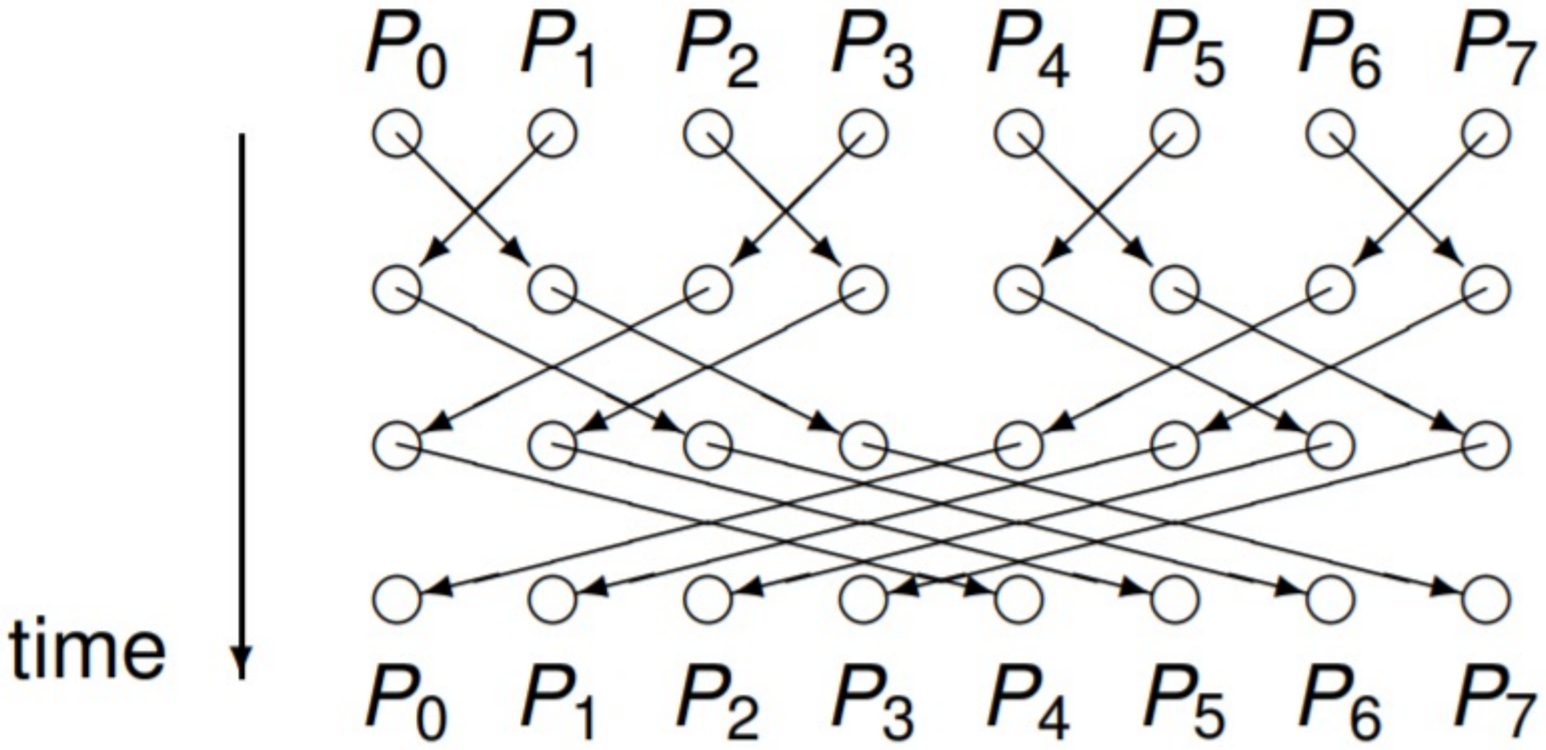
At this point all non-root threads in some await(go) case

Tree Barrier Tradeoffs

- Pros:

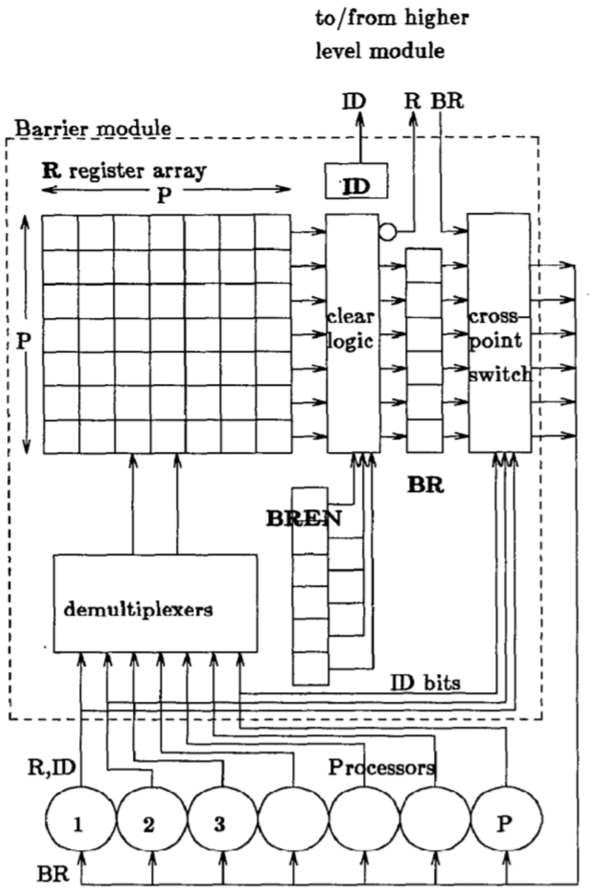
- Cons:

Butterfly Barrier

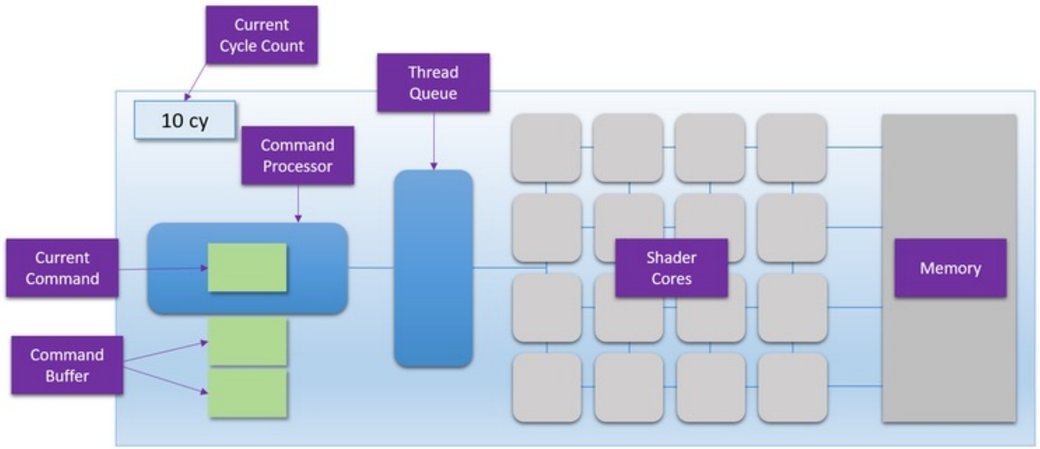


- When would this be preferable?

Hardware Supported Barriers



CPU



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?