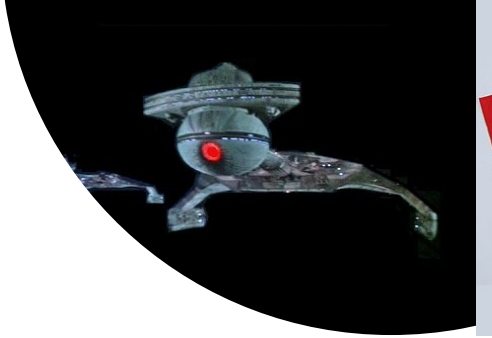
Synchronization: Monitors, Barriers

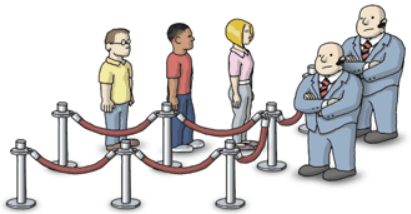
Chris Rossbach

CS378H

Today



- Questions?
- Administrivia
  - My office hours changed: T 3:30-4:30
  - Start looking at Lab 2!
- Material for the day
  - Lab 1 discussion
  - 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?

Instrumentation

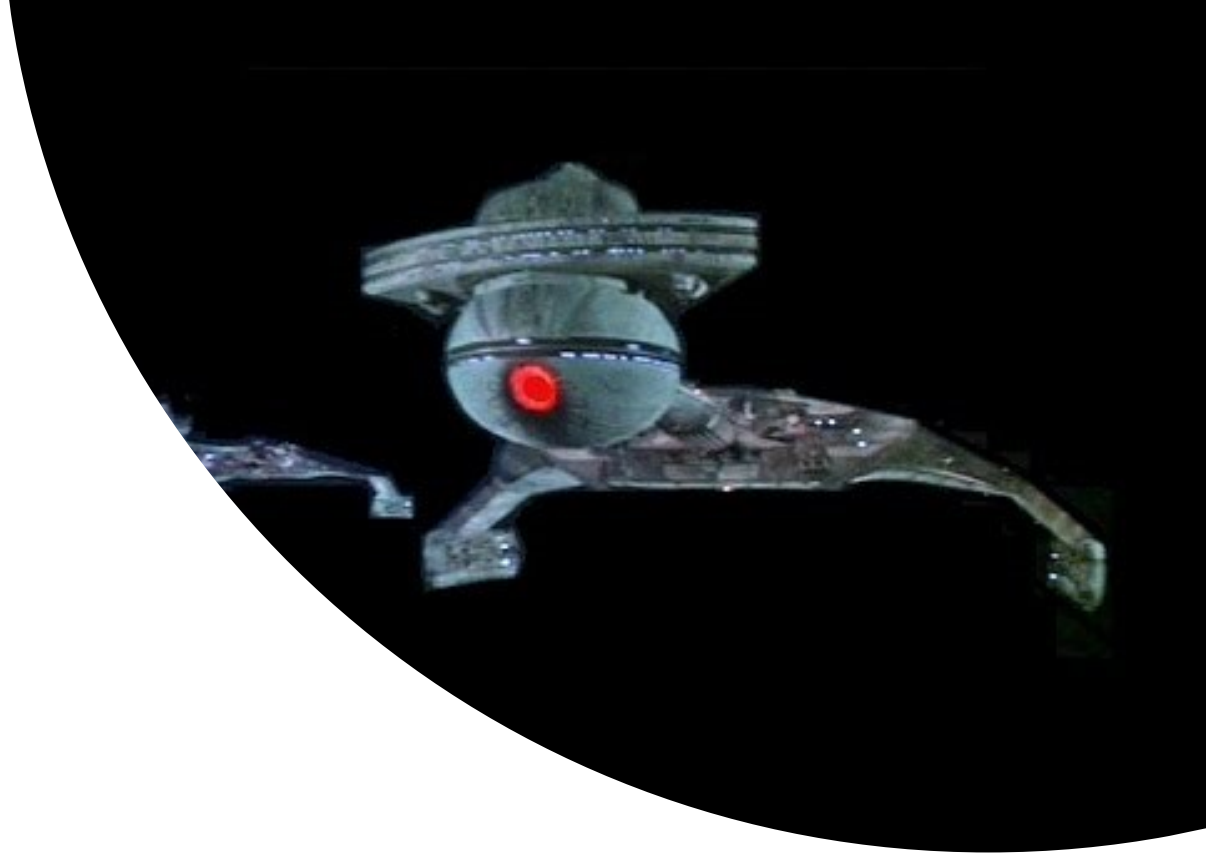
Instrumentation

```
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

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++) {
```



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, wprob, threads, normlost, avgr, minr, maxr, avgw, minw, maxw, parexec, realexec" > step4-spinlock.csv
echo "synctype, wprob, threads, normlost, avgr, minr, maxr, avgw, minw, maxw, parexec, realexec" > step4-atomic.csv
for sync in spinlock atomic; do
    for aff in true; do
    for barrier in false; do
        for ld in true; do
        for wprob in .5 .1 .01; do
            for threads in 1 2 4 8 16; do
            for iter in `seq 1 $ITERS`; do
                output=`/usr/bin/time -f %e -o timing ./locks --iterations $MAX_COUNTER --workers $threads --sync-type $sync --csv true
                realtime=`cat timing`
                echo "$output, $realtime" >> step4-$sync.csv
            done
            done
        done
        done
    done
    done
done

Rscript ./vplot-step4.R step4-spinlock.csv step4-spinlock
Rscript ./vplot-step4.R step4-atomic.csv step4-atomic
```

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=""))
```

Hoare-style Monitors
(aka blocking condition variables)

Hoare-style Monitors

(aka blocking condition variables)

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

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
```

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
```

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
```

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
```

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

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
```

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?

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?

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
```

Must run signaled thread immediately

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

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:

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)

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)

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)

Mesa-style monitors

(aka non-blocking condition variables)

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

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
```

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)
```

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
```

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

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

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

Mesa-style monitors

(aka non-blocking condition variables)

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

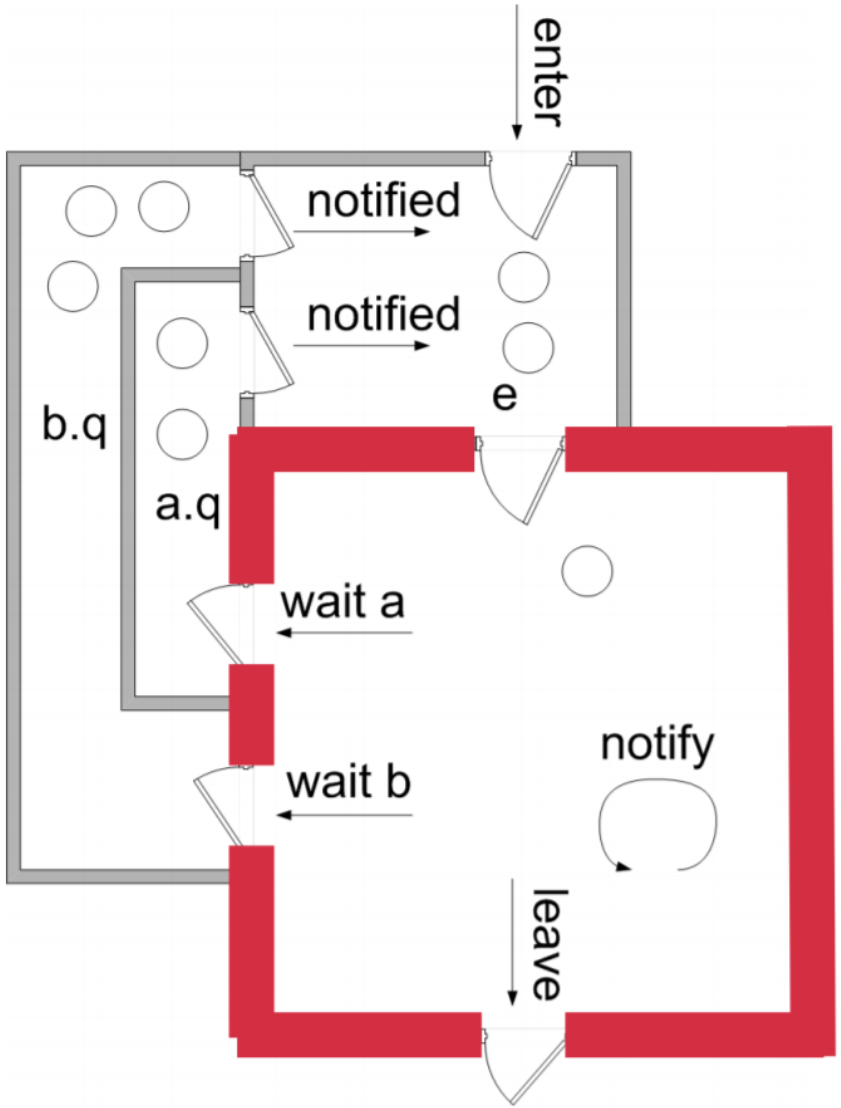
```
schedule:
    if e.any()
        t ← e.pop_front
        t. run
    else
        unlock
```

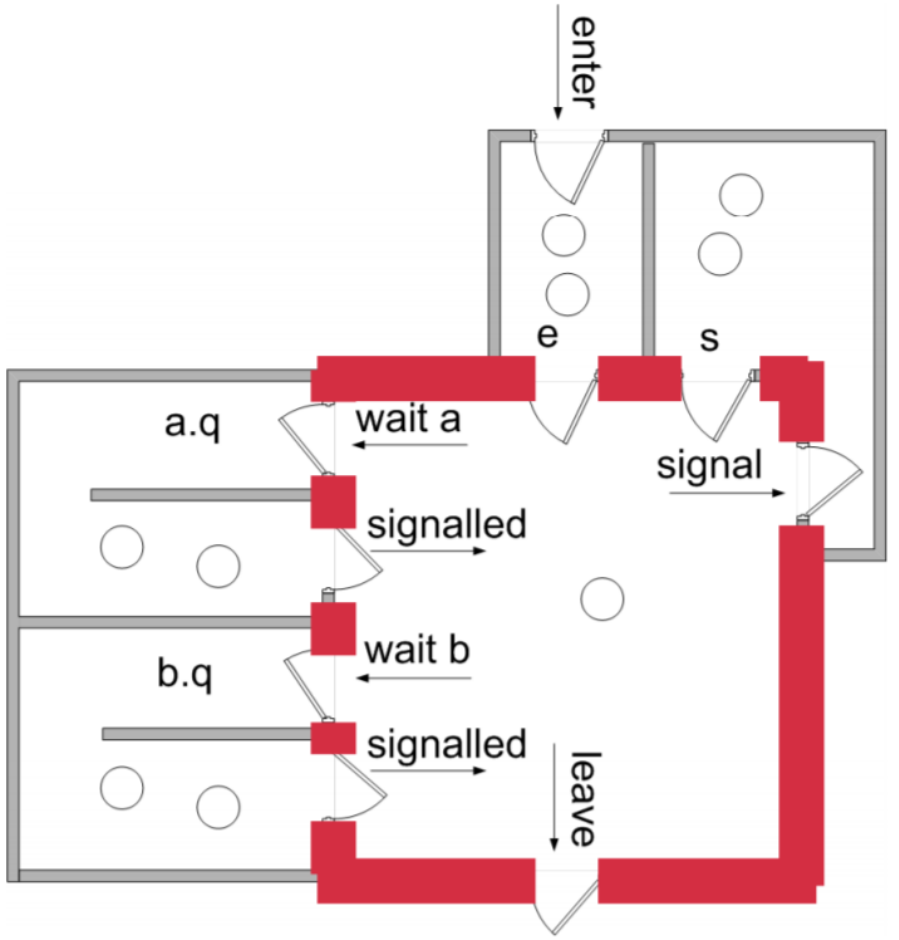
```
notify C:
    if C.q.any()
        t ← C.q.pop_front() // t is "notified "
        e.push_back(t)
```

```
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.
```

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.
```

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?

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

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

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?

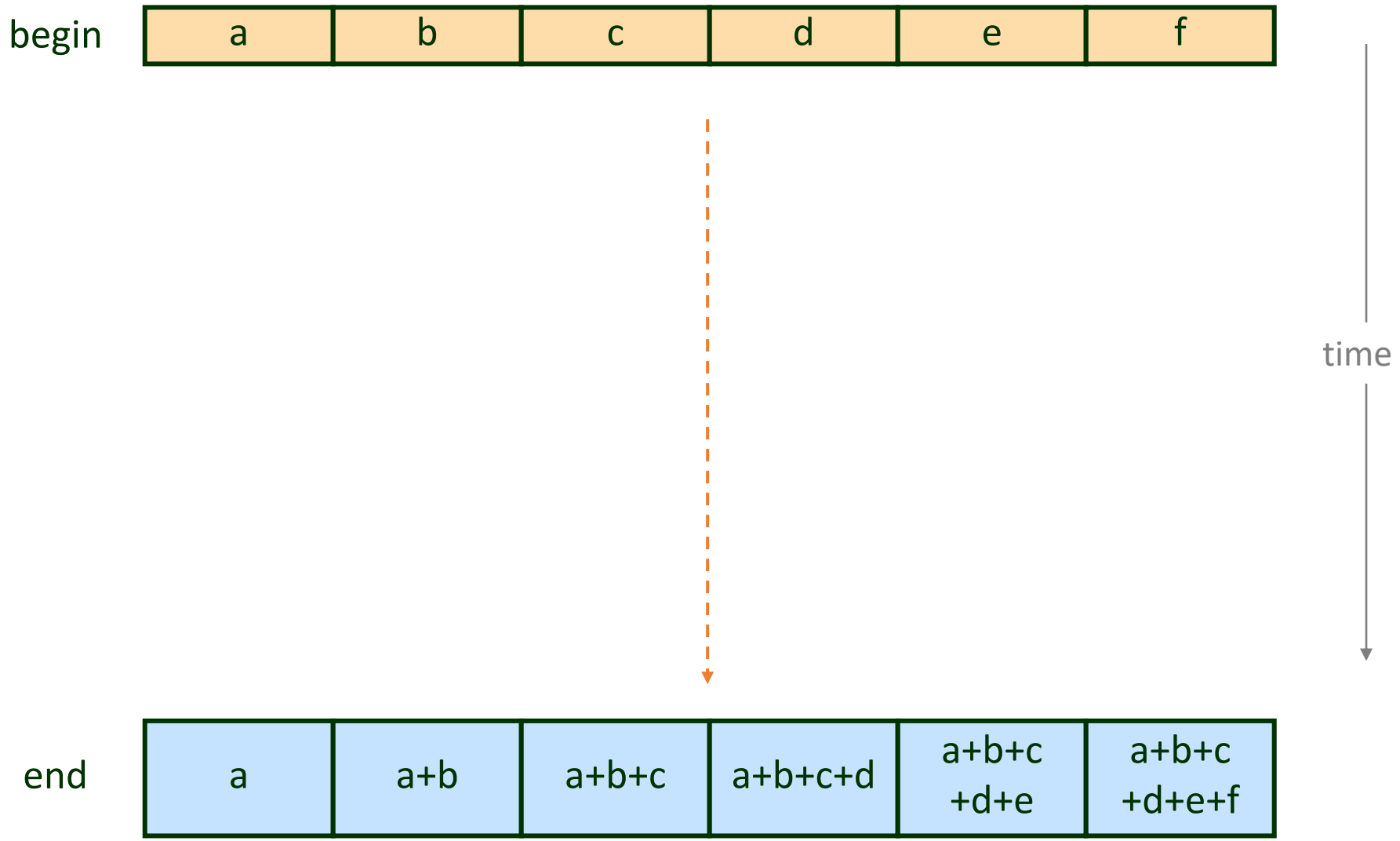
Barriers

Barriers

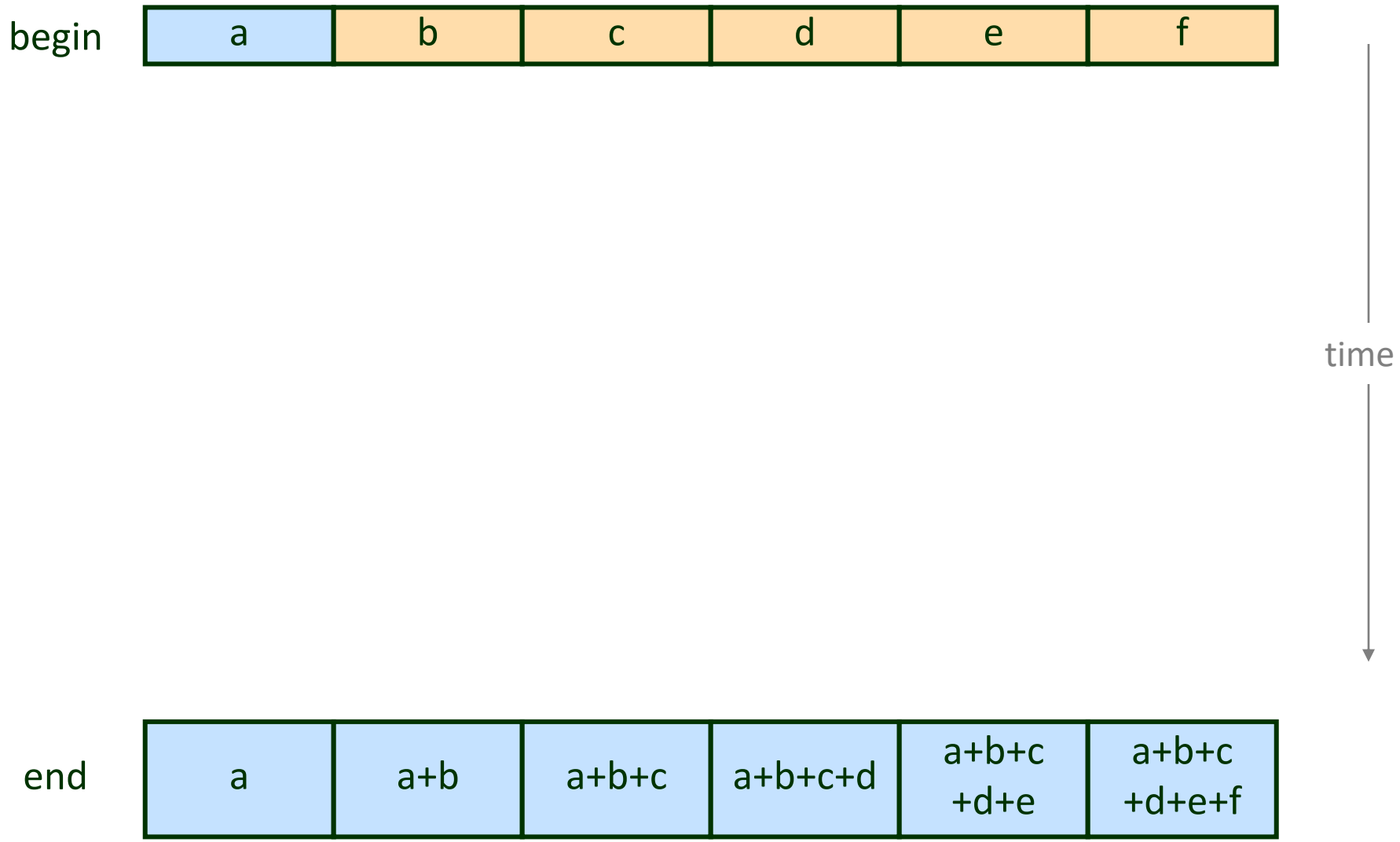
Prefix Sum

Prefix Sum

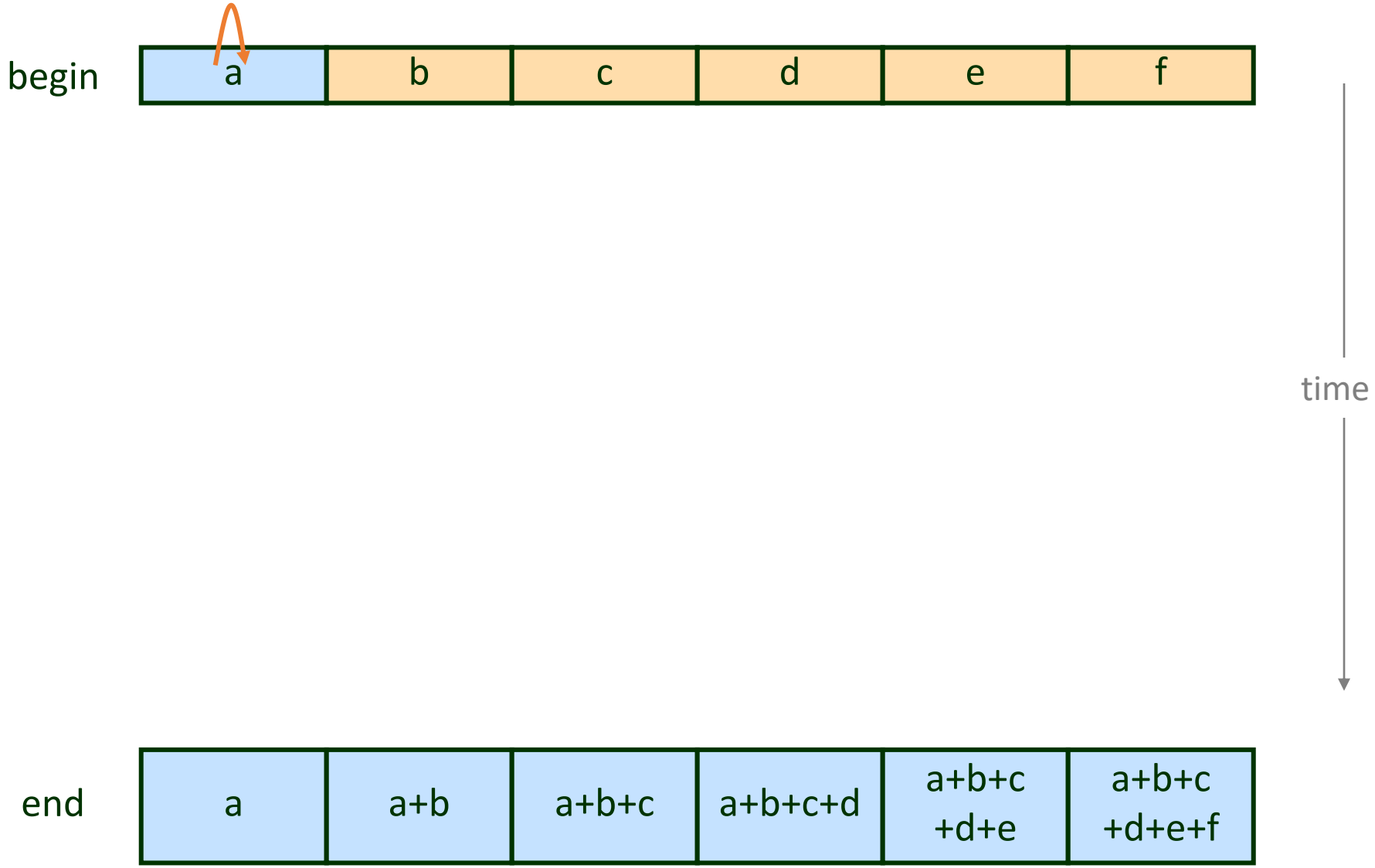
Prefix Sum



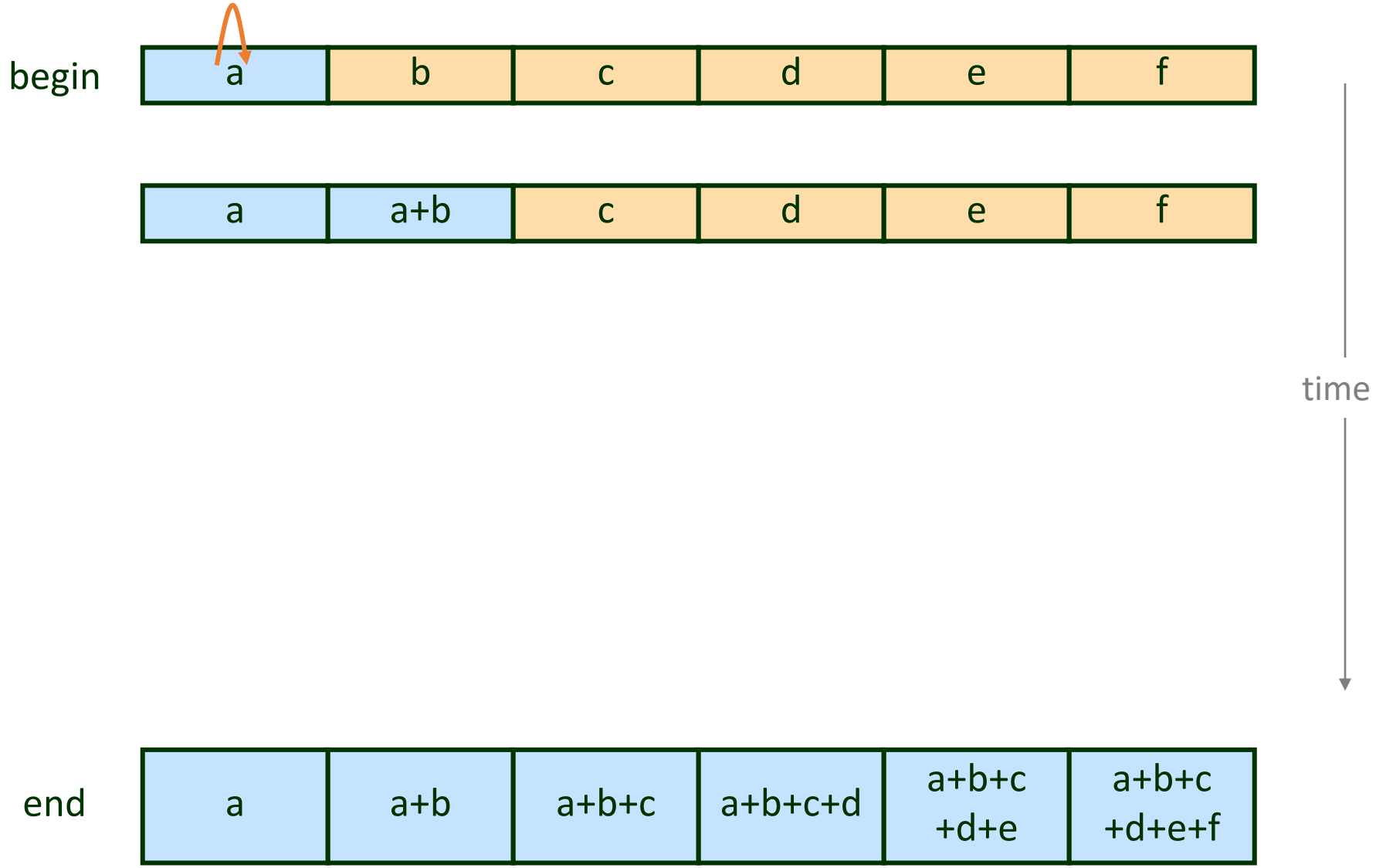
Prefix Sum



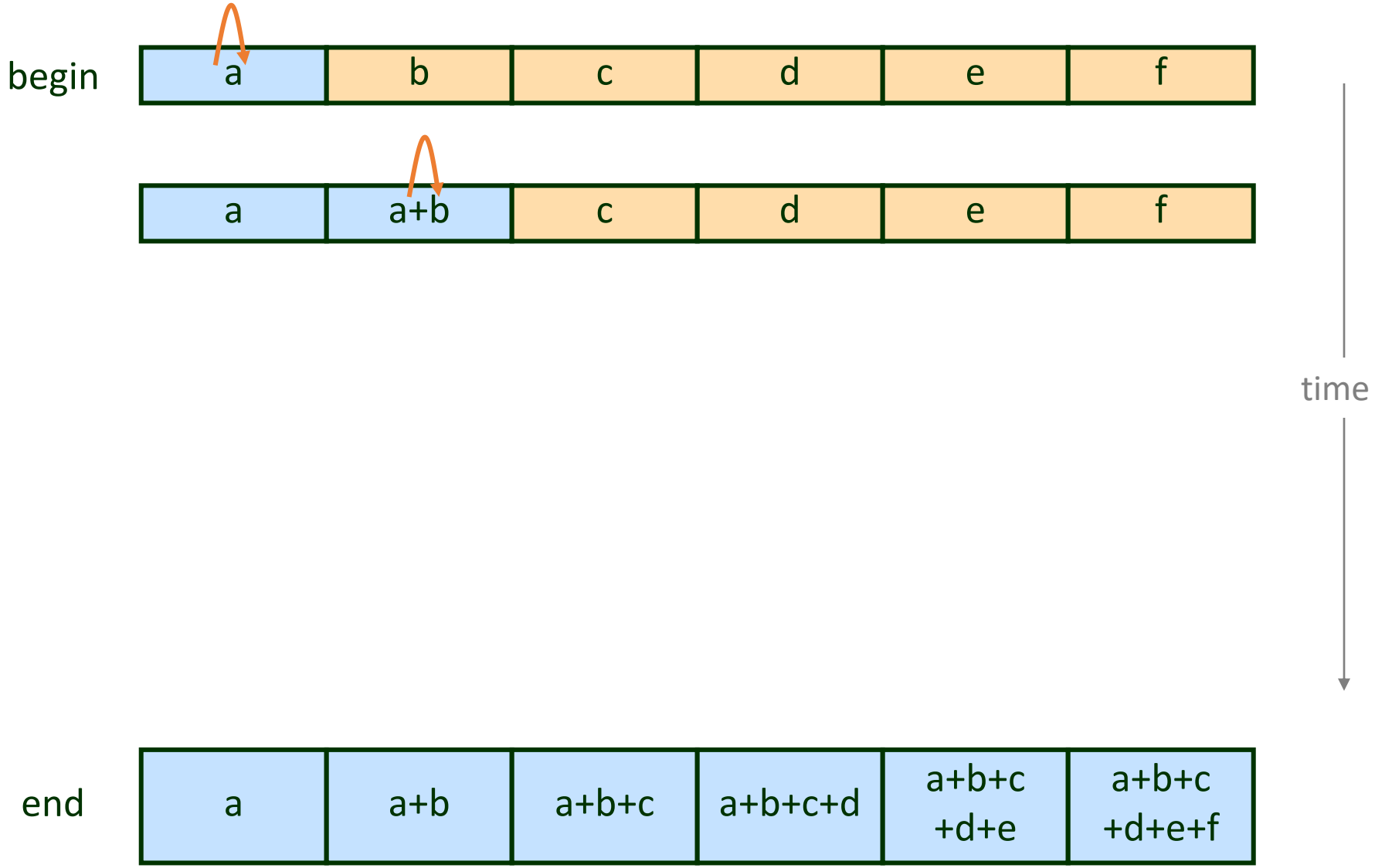
Prefix Sum



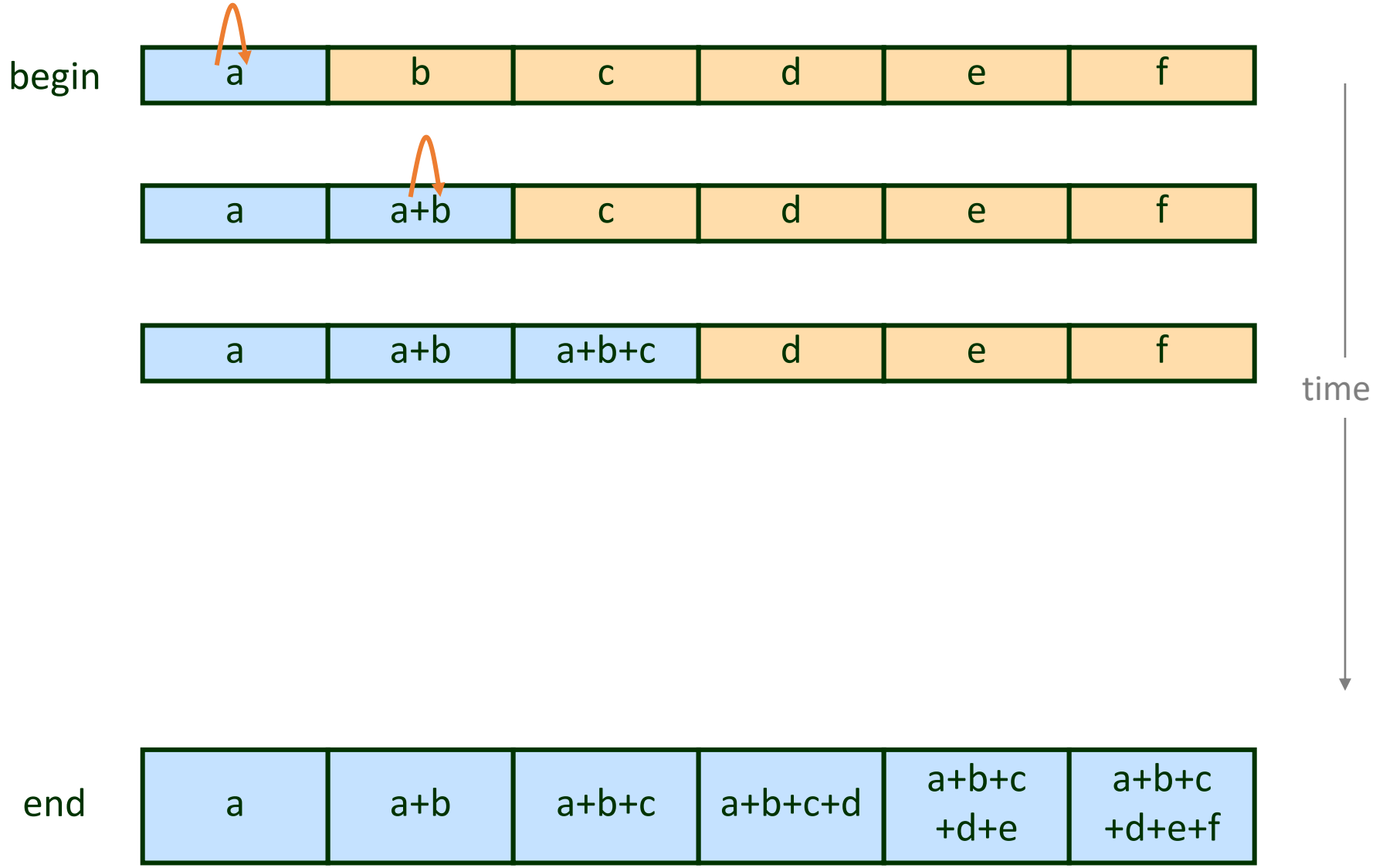
Prefix Sum



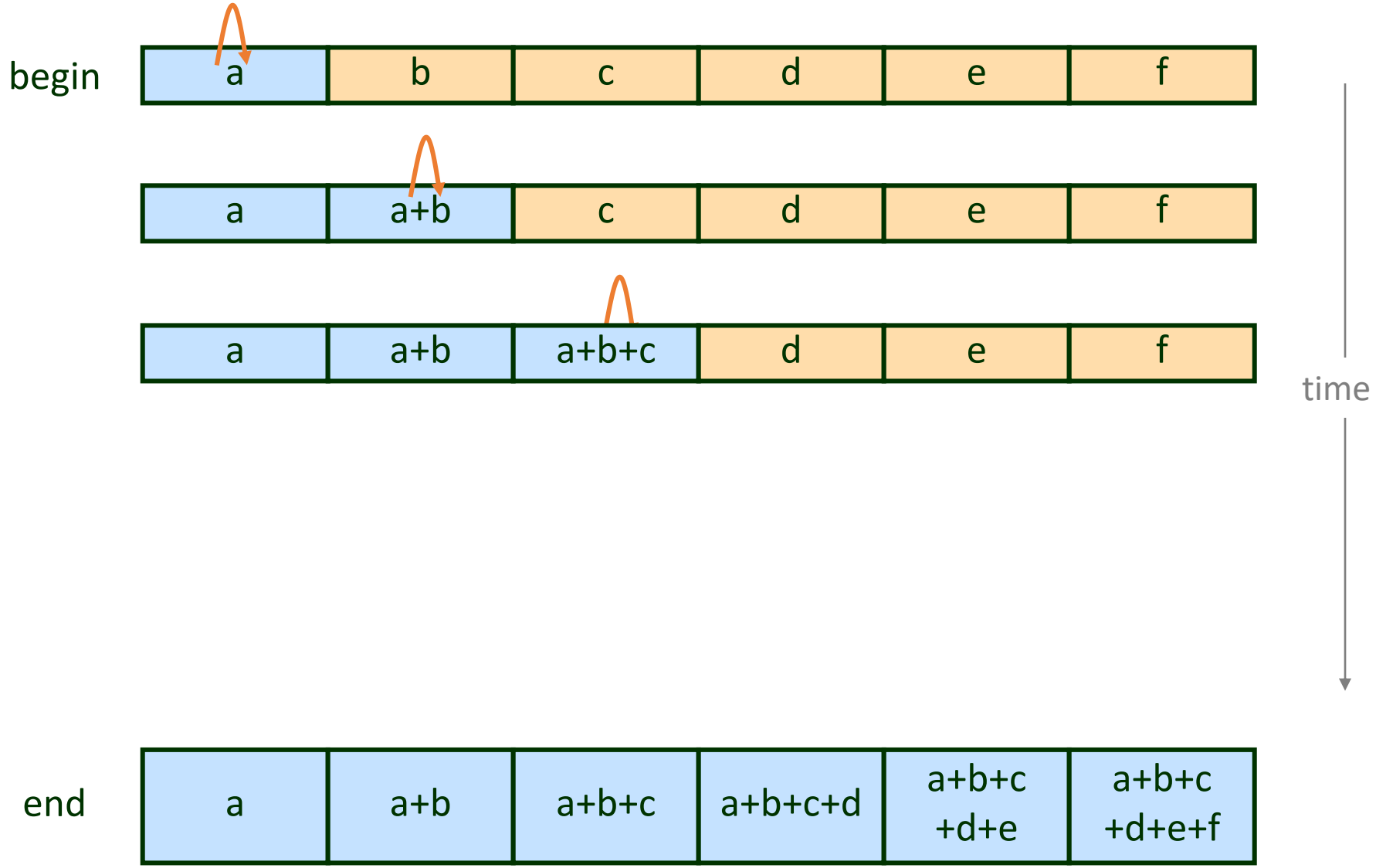
Prefix Sum



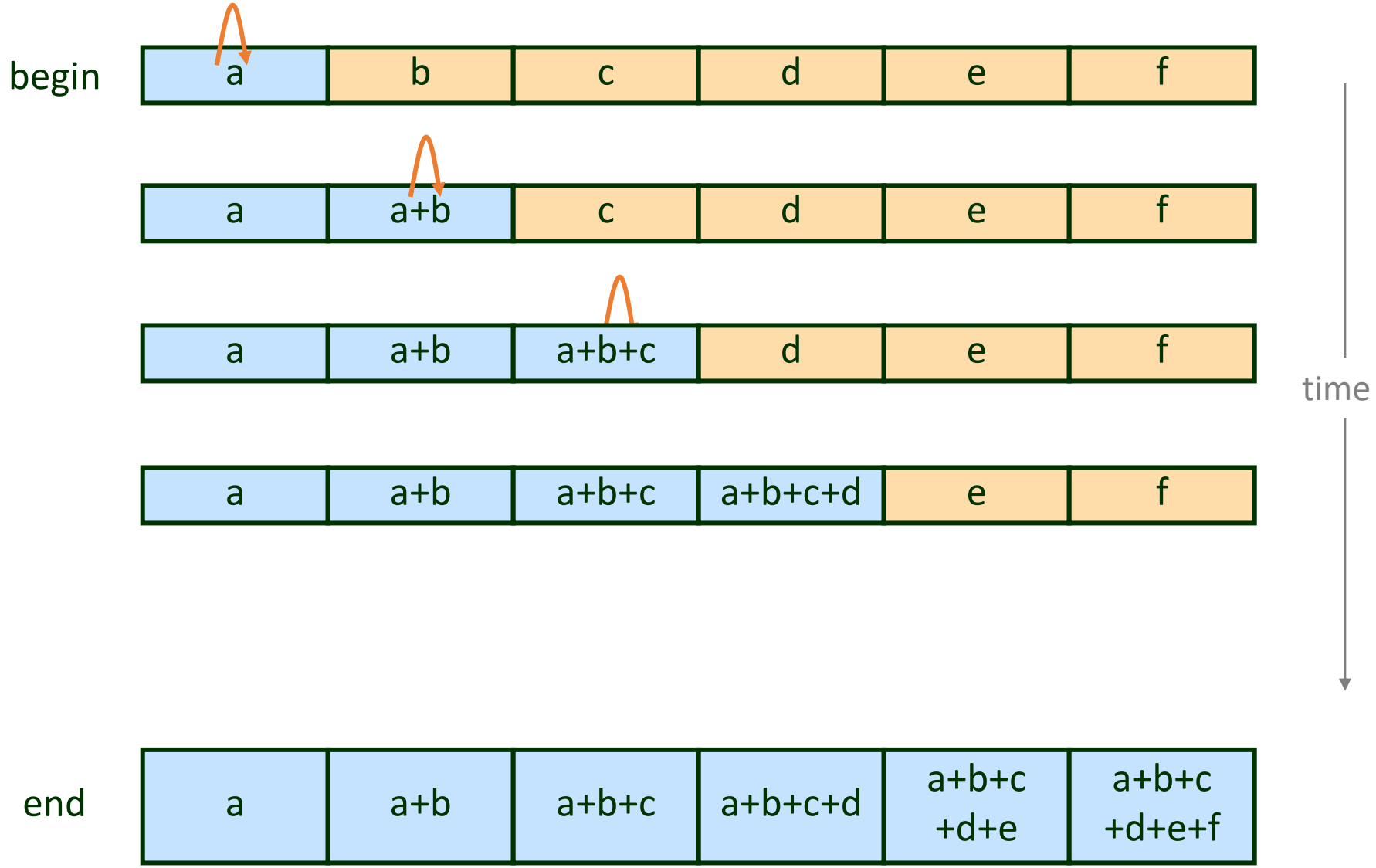
Prefix Sum


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

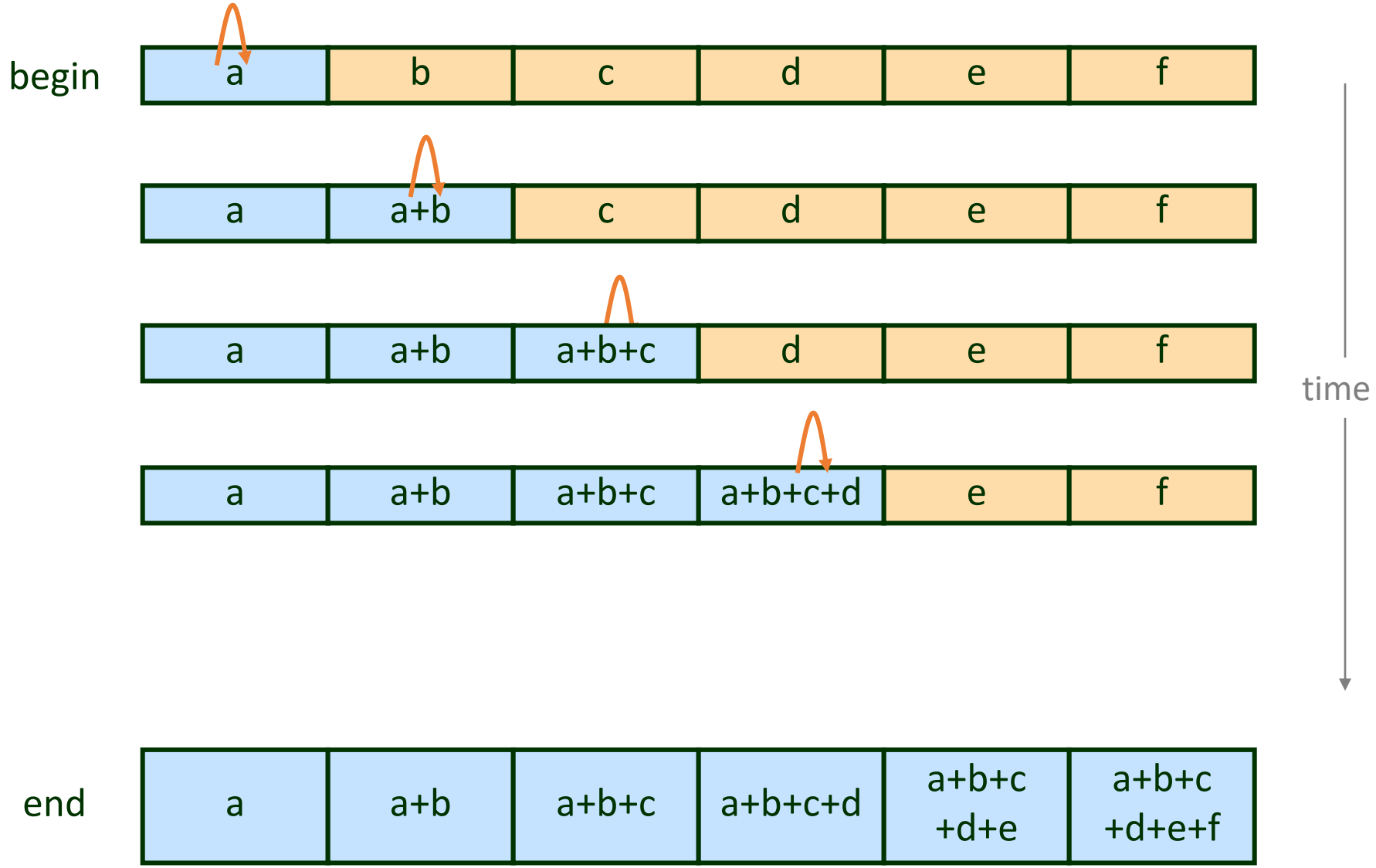
Prefix Sum


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

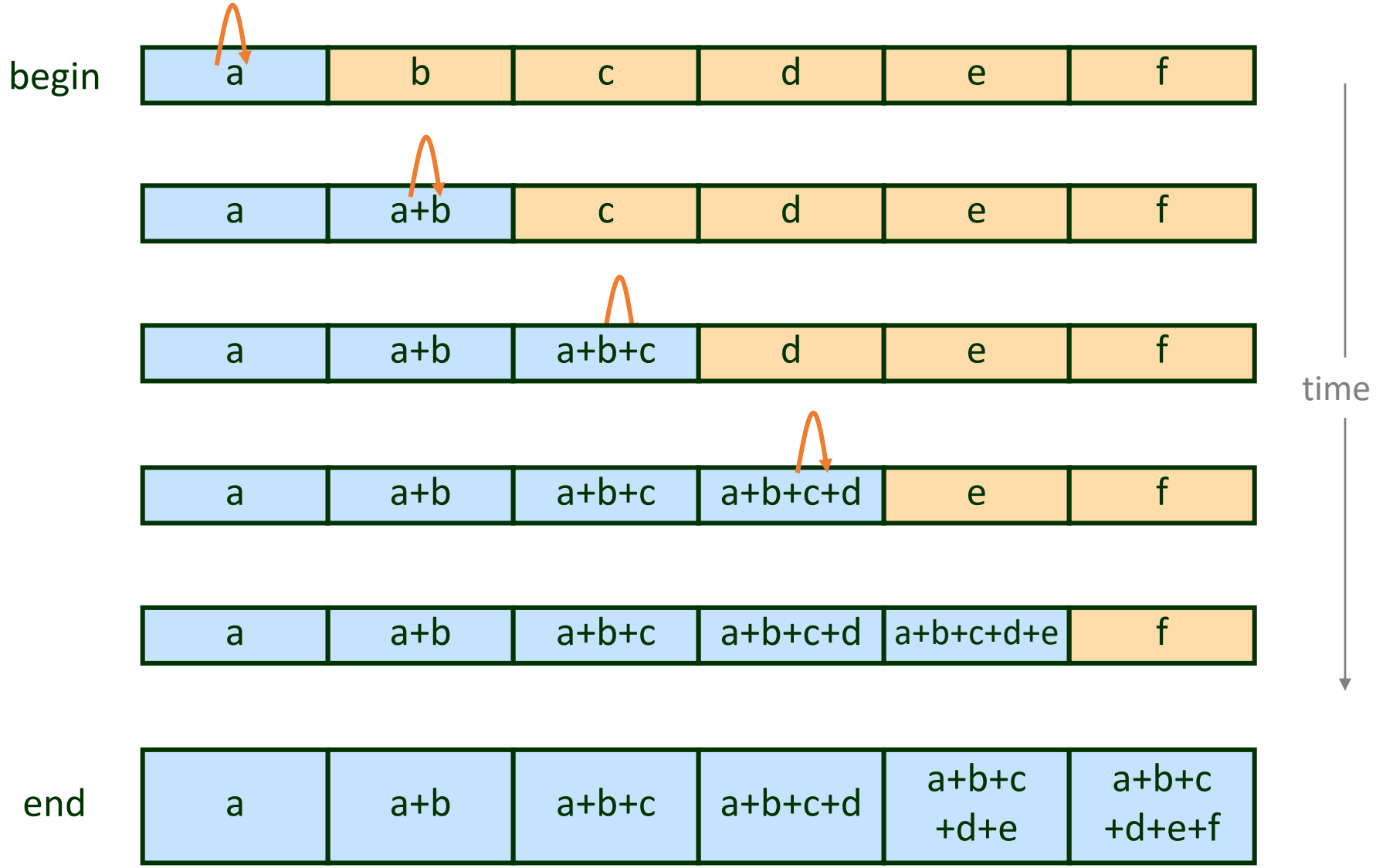
Prefix Sum


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

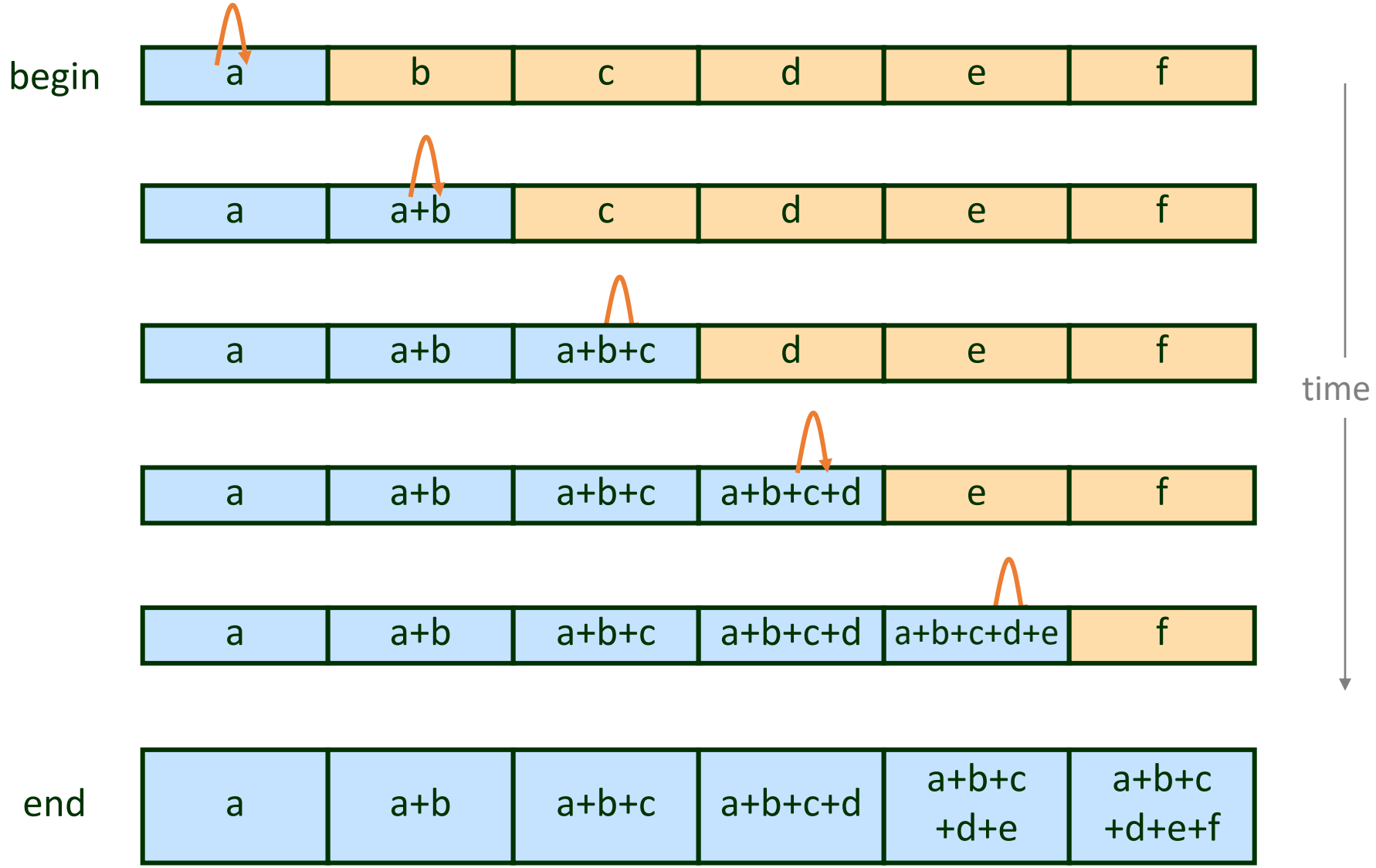
Prefix Sum


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

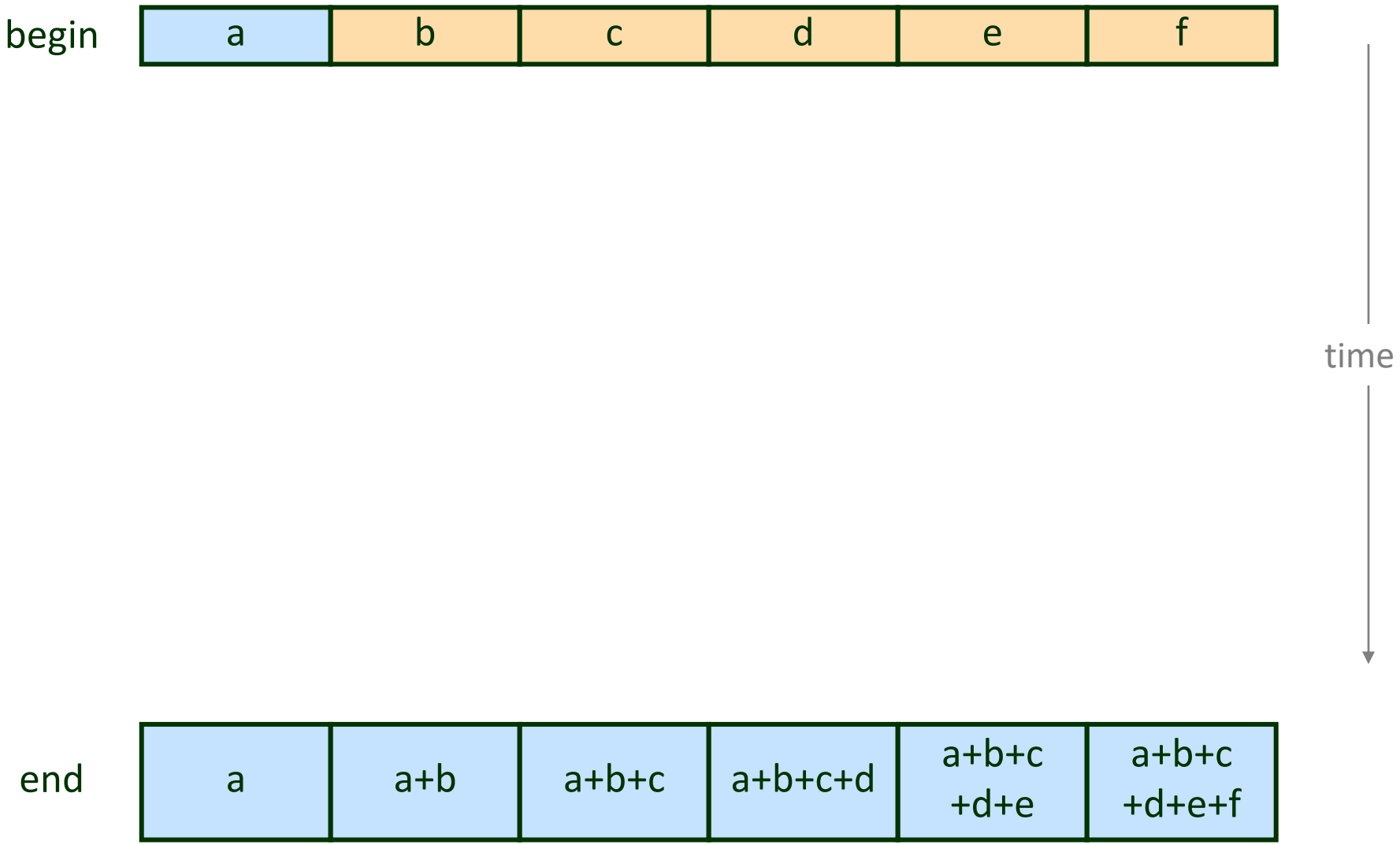
Prefix Sum


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
time

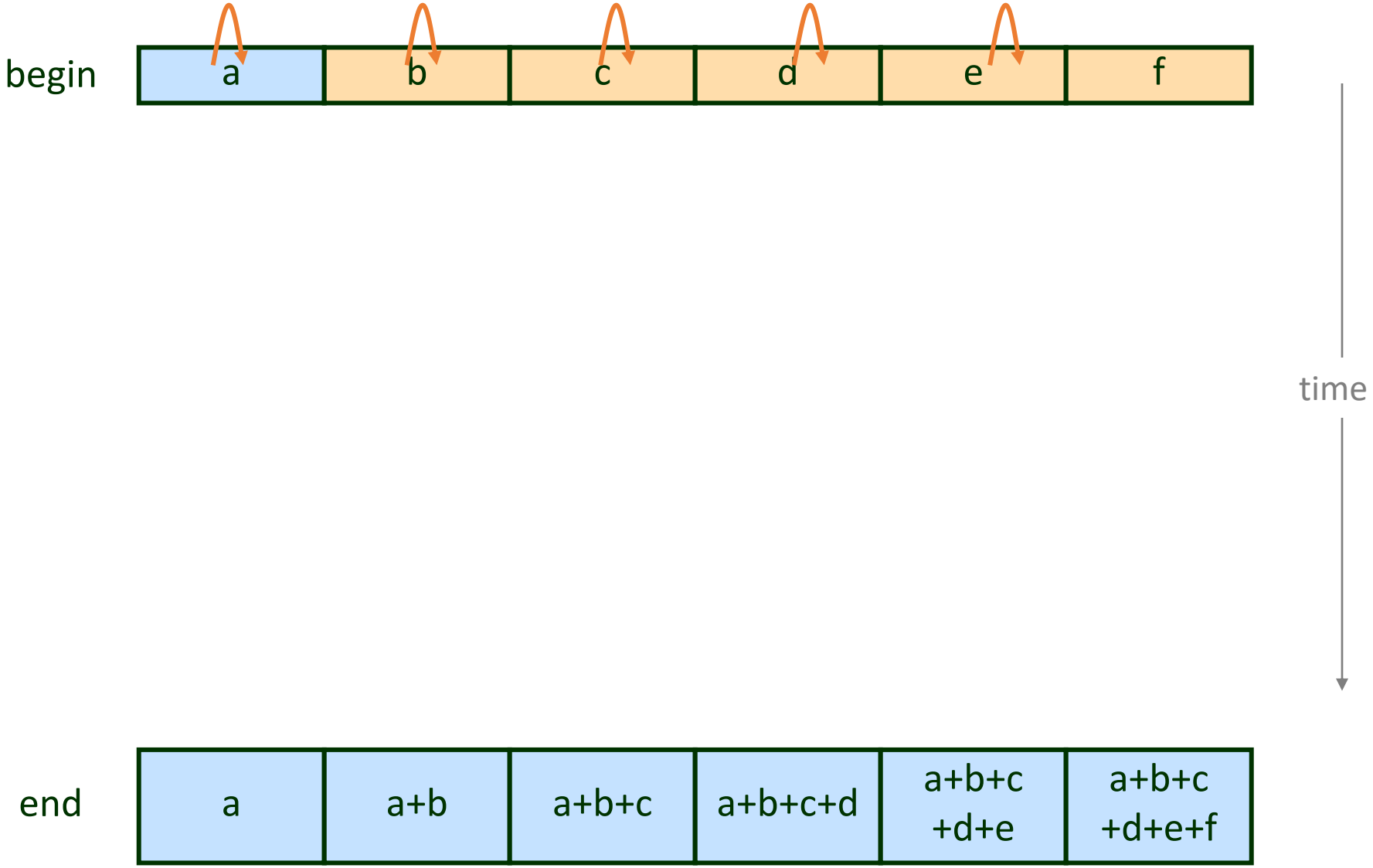
Prefix Sum


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

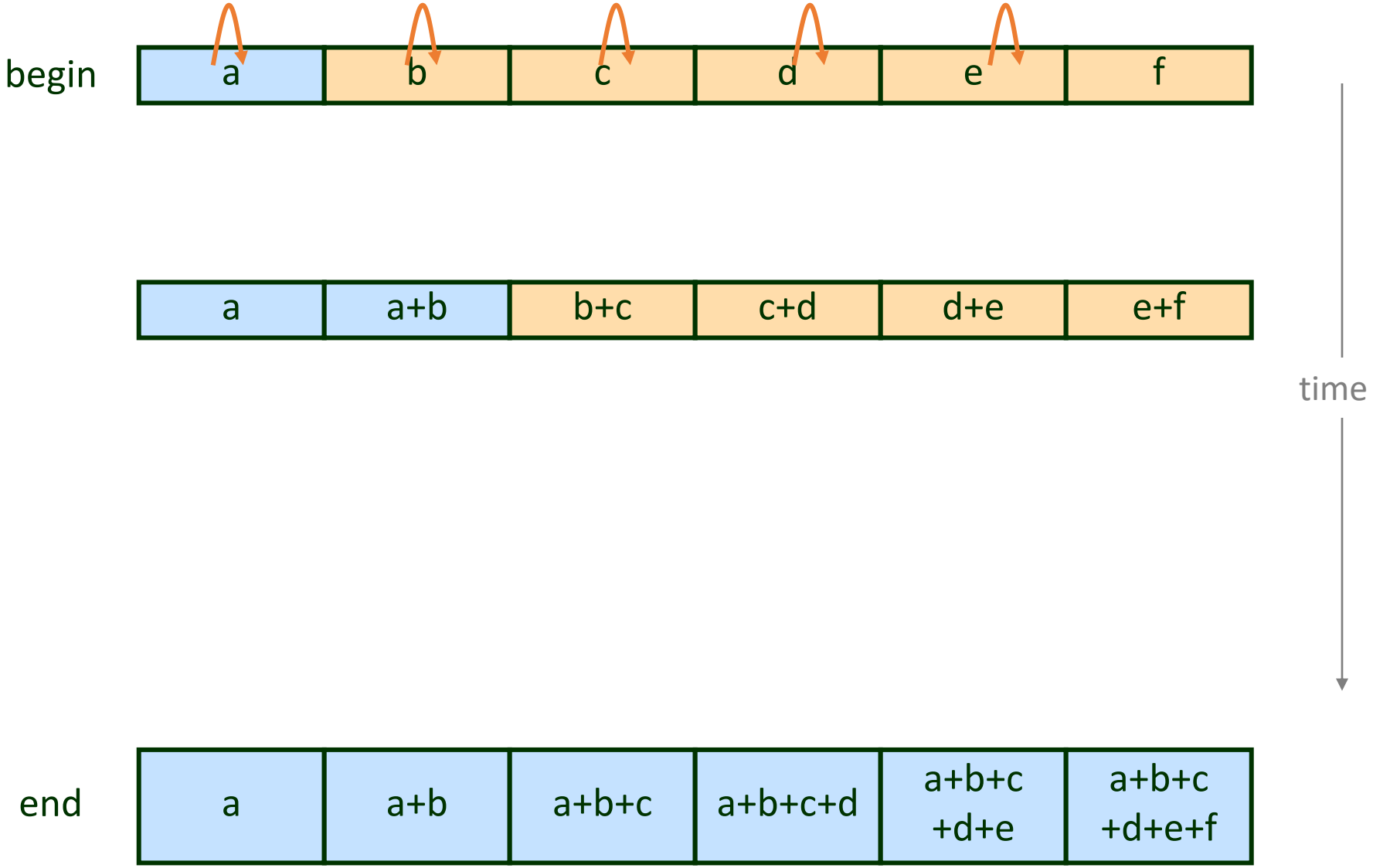
Parallel Prefix Sum



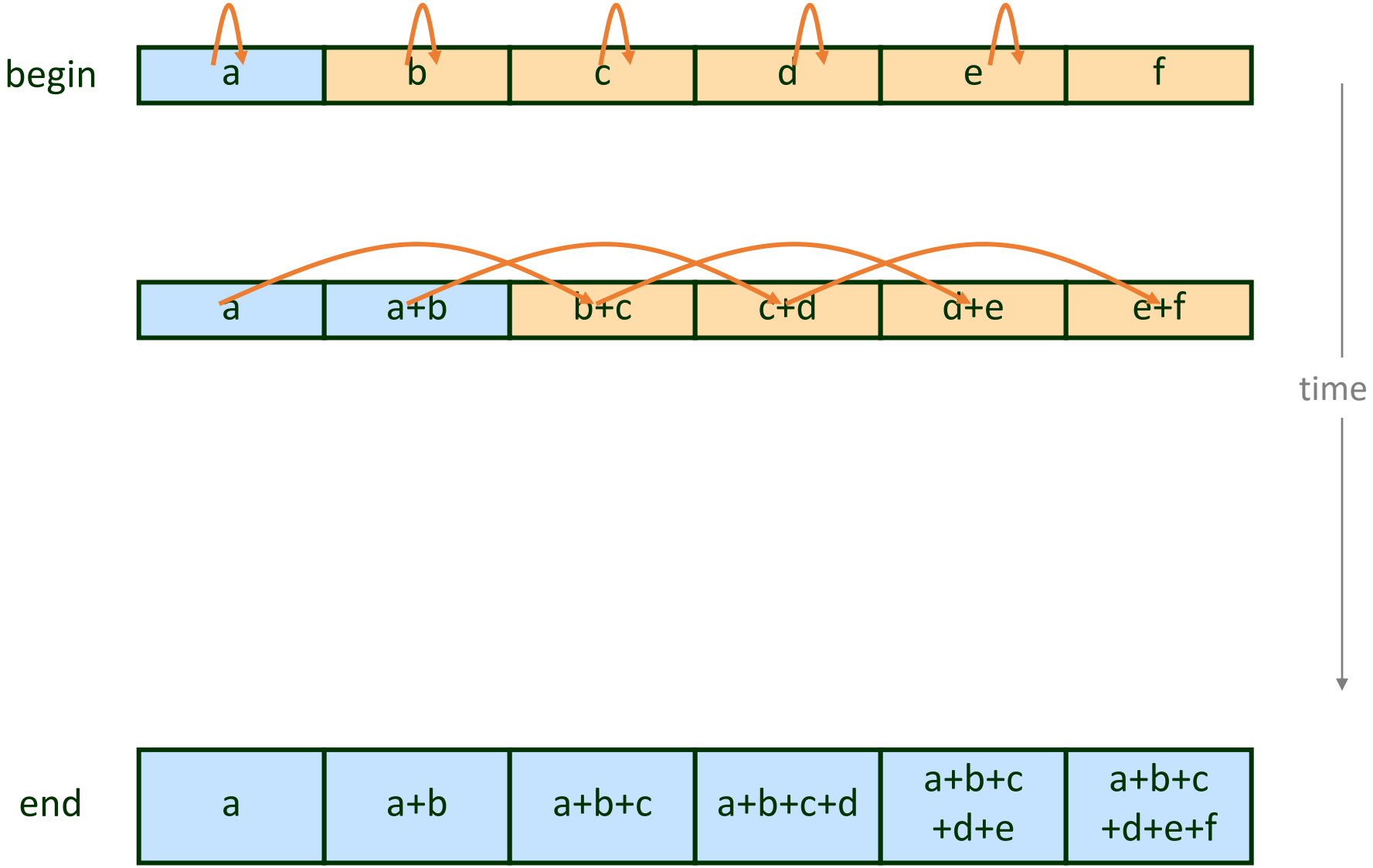
Parallel Prefix Sum



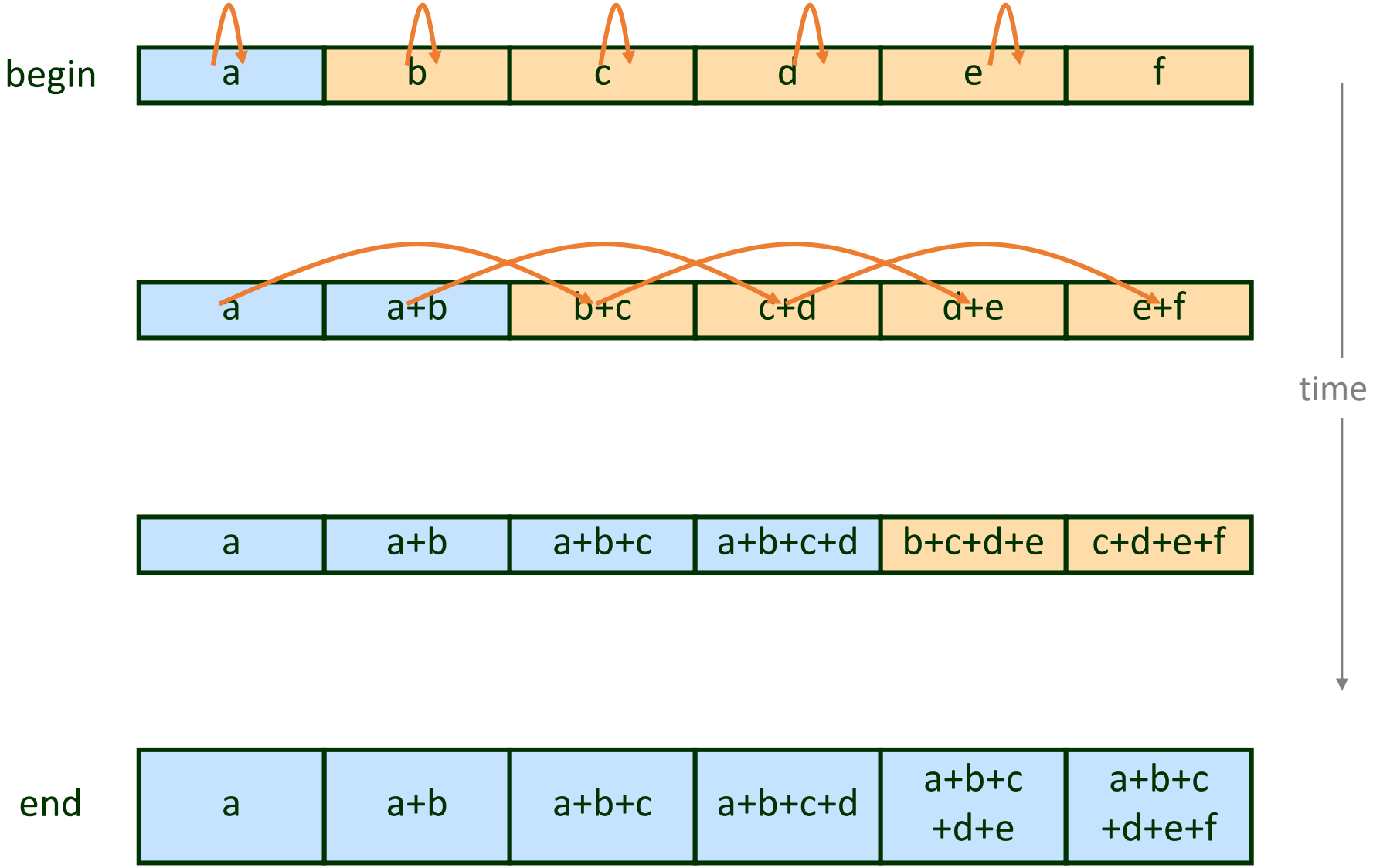
Parallel Prefix Sum



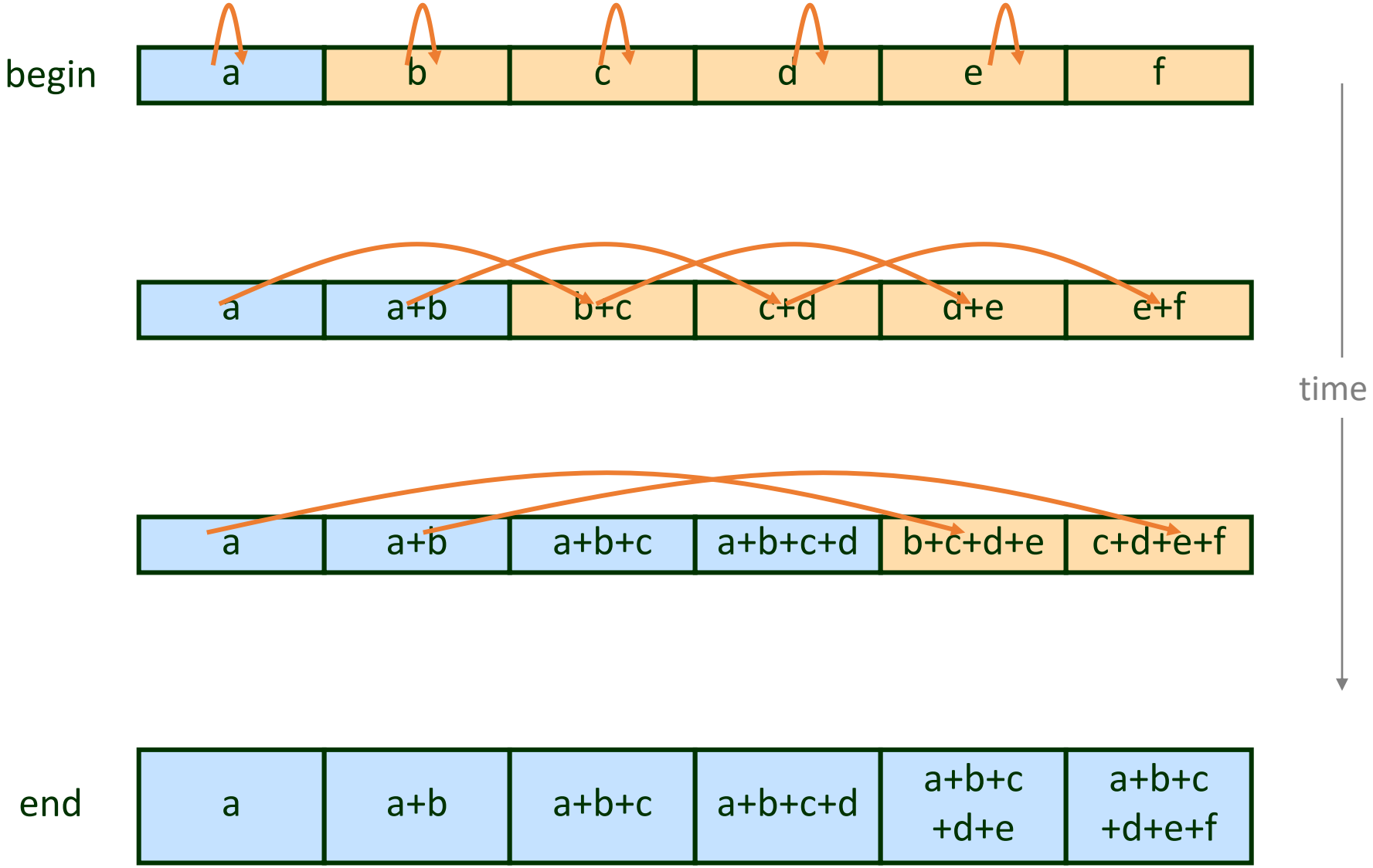
Parallel Prefix Sum



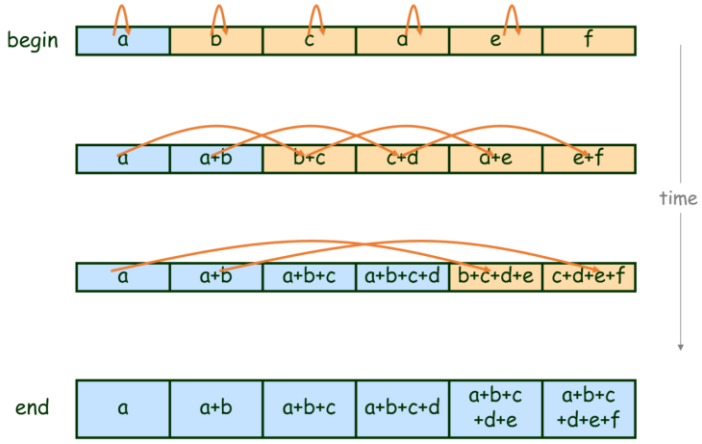
Parallel Prefix Sum



Parallel Prefix Sum

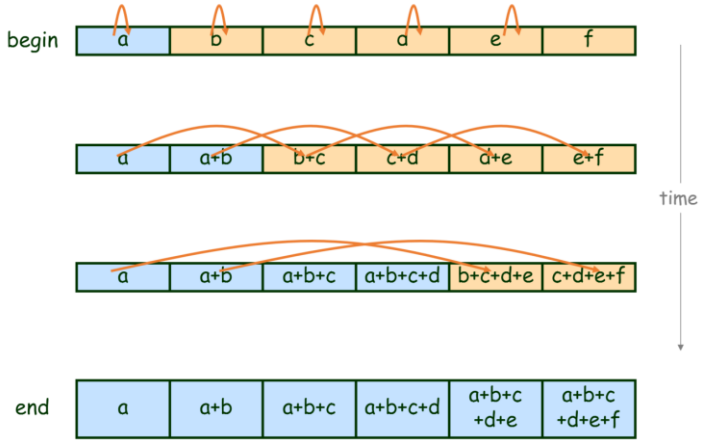


Pthreads Parallel Prefix Sum




```
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];
  }

}
```

Pthreads Parallel Prefix Sum

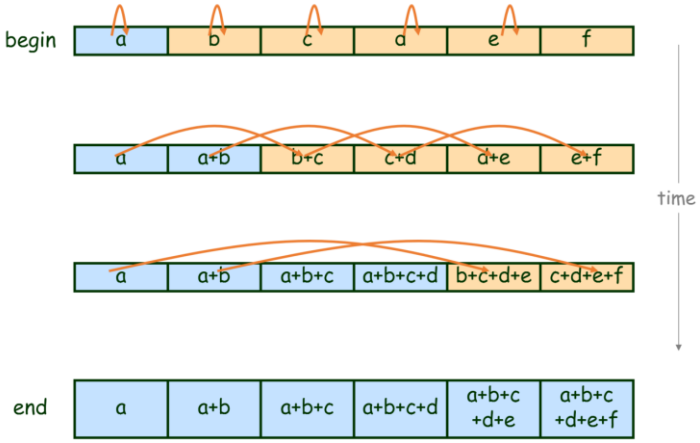



```
int g_values[N] = { a, b, c, d, e, f };

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?

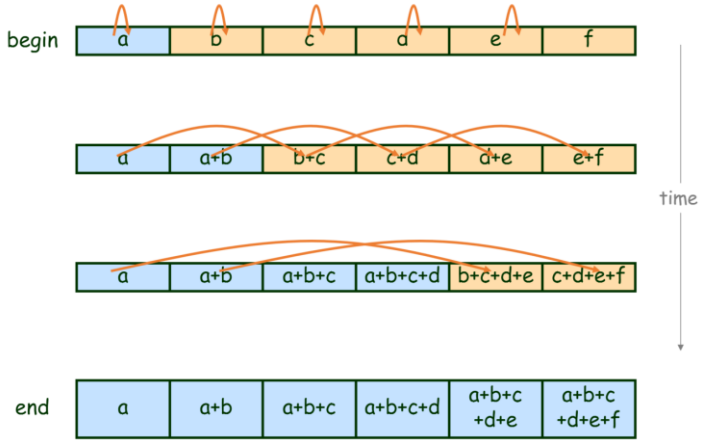
Pthreads Parallel Prefix Sum

```
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]);
  }

}
```



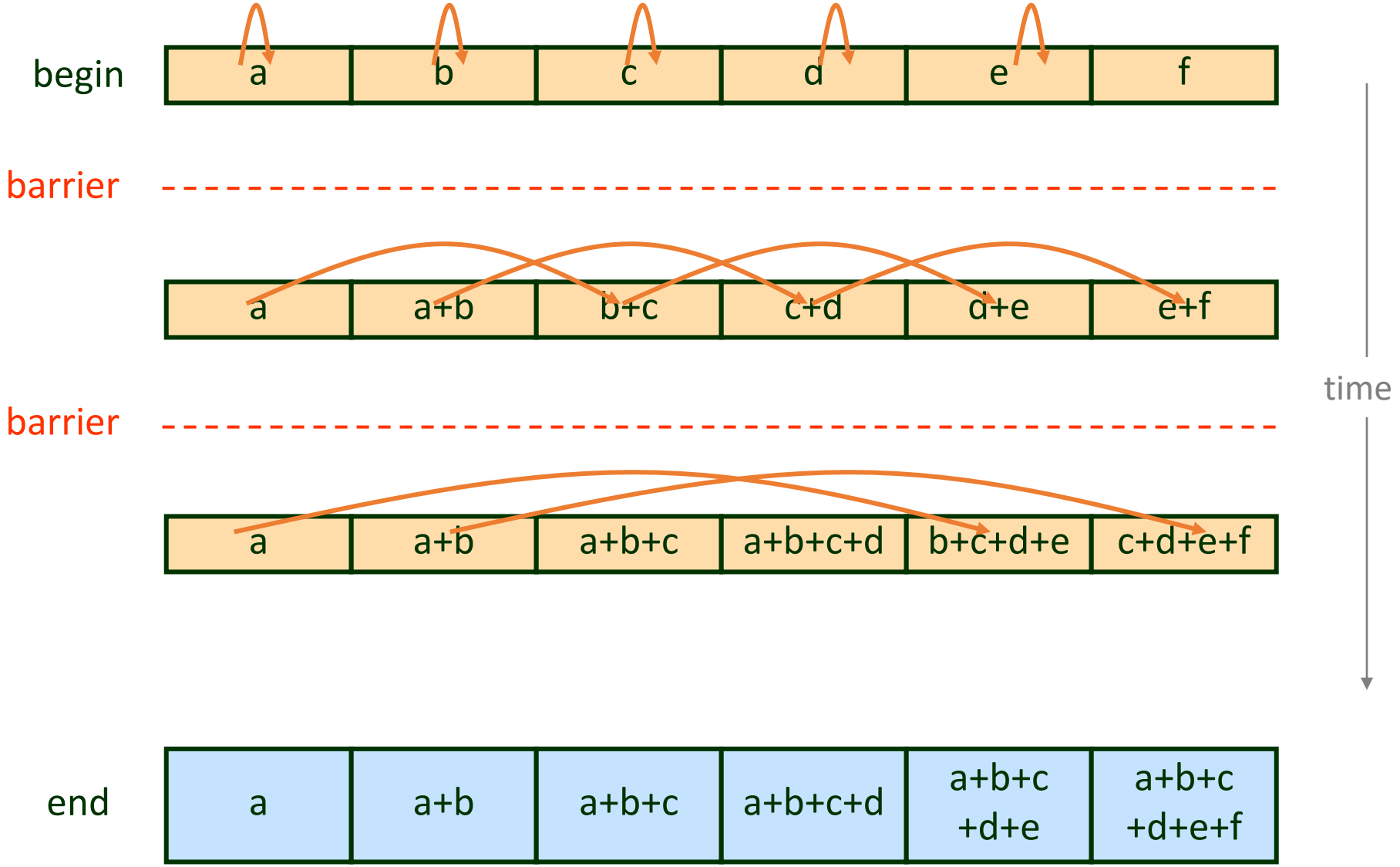
Pthreads Parallel Prefix Sum

```
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]);
  }

}
```



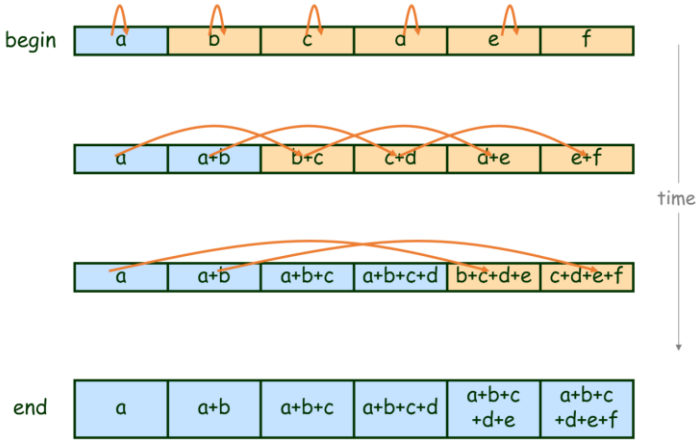
fixed?

Parallel Prefix Sum



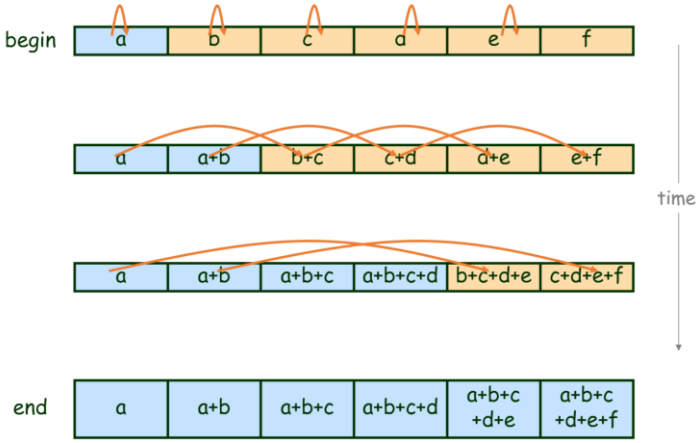
Pthreads Parallel Prefix Sum

```
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);
  }
}
```



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:

Barrier Goals

Desirable barrier properties:

- Low shared memory space complexity

Barrier Goals

Desirable barrier properties:

- Low shared memory space complexity
- Low contention on shared objects

Barrier Goals

Desirable barrier properties:

- Low shared memory space complexity
- Low contention on shared objects
- Low shared memory references per process

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

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

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

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

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

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

Barrier using Semaphores

Algorithm for N threads

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)
```

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, ...)
```

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)
```

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)
```

```
1   sem_wait(arrival);
2   if(++counter < N)
3     sem_post(arrival);
4   else
5     sem_post(departure);
6   sem_wait(departure);
7   if(--counter > 0)
8     sem_post(departure)
9   else
10    sem_post(arrival)
```

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)
```

Phase I

```
1   sem_wait(arrival);
2   if(++counter < N)
3     sem_post(arrival);
4   else
5     sem_post(departure);
```

Phase II

```
6   sem_wait(departure);
7   if(--counter > 0)
8     sem_post(departure)
9   else
10    sem_post(arrival)
```

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)
```

Phase I

```
1   sem_wait(arrival);
2   if(++counter < N)
3     sem_post(arrival);
4   else
5     sem_post(departure);
```

Phase II

```
6   sem_wait(departure);
7   if(--counter > 0)
8     sem_post(departure)
9   else
10    sem_post(arrival)
```

First N-1 threads post on arrival, wait on departure

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)
```

Phase I

Phase II

```
1   sem_wait(arrival);
2   if(++counter < N)
3     sem_post(arrival);
4   else
5     sem_post(departure);
6   sem_wait(departure);
7   if(--counter > 0)
8     sem_post(departure)
9   else
10    sem_post(arrival)
```

First N-1 threads post on arrival, wait on departure

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

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)
```

Phase I

Phase II

```
1   sem_wait(arrival);
2   if(++counter < N)
3     sem_post(arrival);
4   else
5     sem_post(departure);
6   sem_wait(departure);
7   if(--counter > 0)
8     sem_post(departure)
9   else
10    sem_post(arrival)
```

First N-1 threads post on arrival, wait on departure

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 = **1**
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

Semaphore Barrier Action Zone

N == 3

```
shared      sem_t arrival = 1
            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

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

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

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

```
em_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

Semaphore Barrier Action Zone
N == 3

```
shared      sem_t arrival = 0
            sem_t departure = 0
            atomic int counter = 1
```

1

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

```
em_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)
```

Semaphore Barrier Action Zone
N == 3

```
shared      sem_t arrival = 1
            sem_t departure = 0
            atomic int counter = 1
```

1

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

```
em_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)
```

Semaphore Barrier Action Zone
N == 3

```
shared      sem_t arrival = 0
            sem_t departure = 0
            atomic int counter = 1
```

1

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)
```

Semaphore Barrier Action Zone
N == 3

```
shared      sem_t arrival = 0
            sem_t departure = 0
            atomic int counter = 2
```

1

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)
```

Semaphore Barrier Action Zone
N == 3

```
shared      sem_t arrival = 1
            sem_t departure = 0
            atomic int counter = 2
```

1

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)
```

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 0
          atomic int counter = 2
```

1

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)
```

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 0
          atomic int counter = 3
```

1

CPU 0

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

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 1
          atomic int counter = 3
```

1

CPU 0

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

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 0
          atomic int counter = 3
```

1

CPU 0

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

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 0
          atomic int counter = 2
```

1

CPU 0

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

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 1
          atomic int counter = 2
```

1

CPU 0

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

Semaphore Barrier Action Zone
N == 3

```
shared      sem_t arrival = 0
            sem_t departure = 0
            atomic int counter = 2
```

1

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)
```

Semaphore Barrier Action Zone
N == 3

```
shared      sem_t arrival = 0
            sem_t departure = 0
            atomic int counter = 1
```

1

CPU 0

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

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 1
          atomic int counter = 1
```

1

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)
```

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 0
          atomic int counter = 1
```

1

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)
```

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 0
          sem_t departure = 0
          atomic int counter = 0
```

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 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

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 1
          sem_t departure = 0
          atomic int counter = 0
```

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 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

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 1
          sem_t departure = 0
          atomic int counter = 0
```

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 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

Still correct if counter is not atomic?

Semaphore Barrier Action Zone
N == 3

```
shared    sem_t arrival = 1
          sem_t departure = 0
          atomic int counter = 0
```

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 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

Do we need two phases?

Still correct if counter is not atomic?

Barrier using Semaphores
Properties

- Pros:

- Cons:

Barrier using Semaphores
Properties

- Pros:
  - Very Simple
  - Space complexity O(1)
  - Symmetric
- Cons:

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

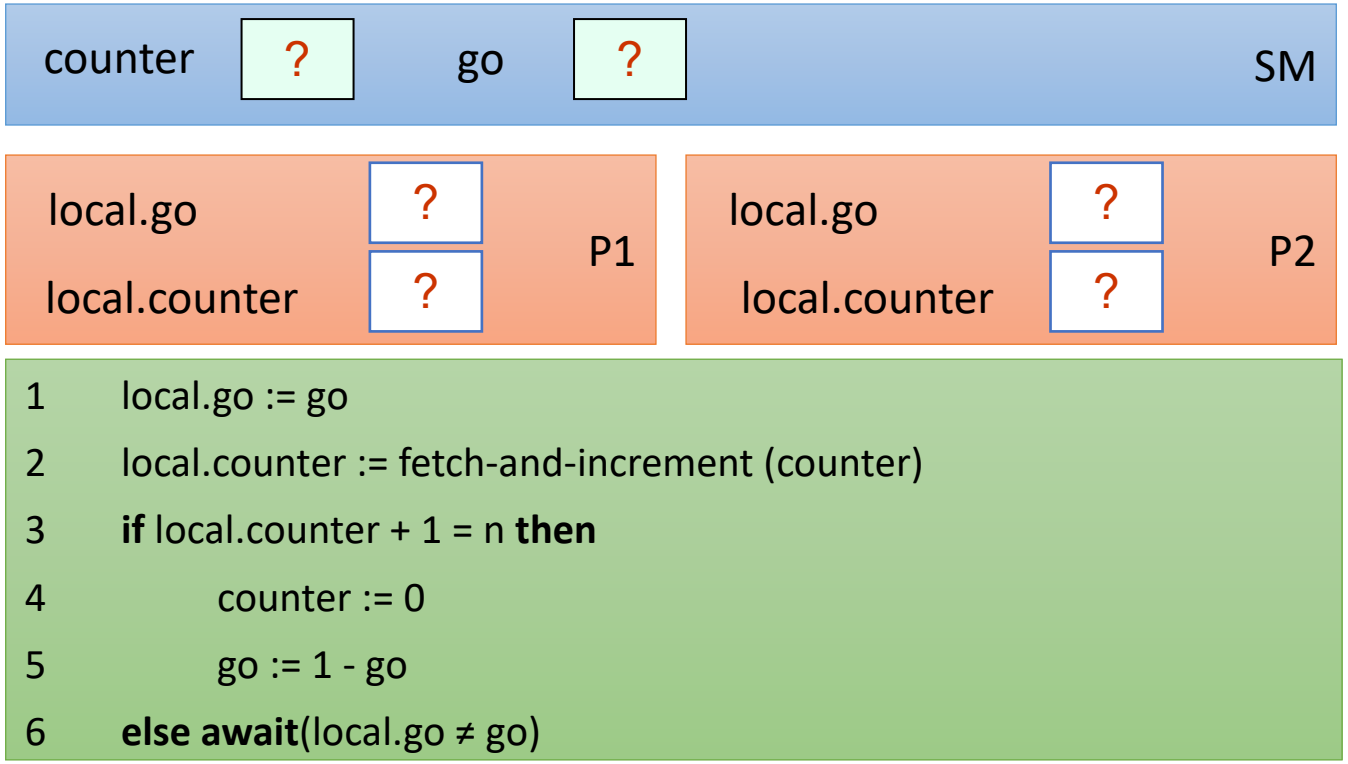
Simple Barrier Using an Atomic Counter

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

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

Simple Barrier Using an Atomic Counter

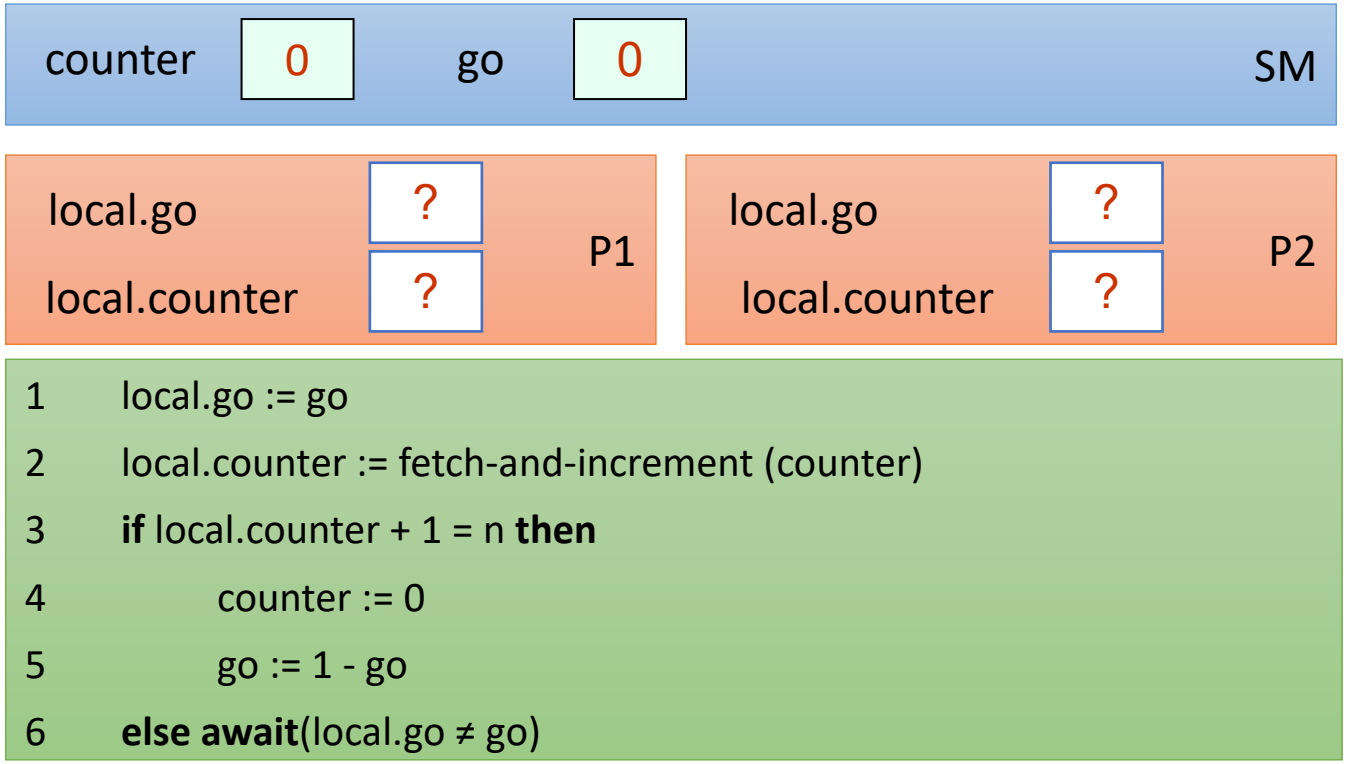
Run for n=2 Threads




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

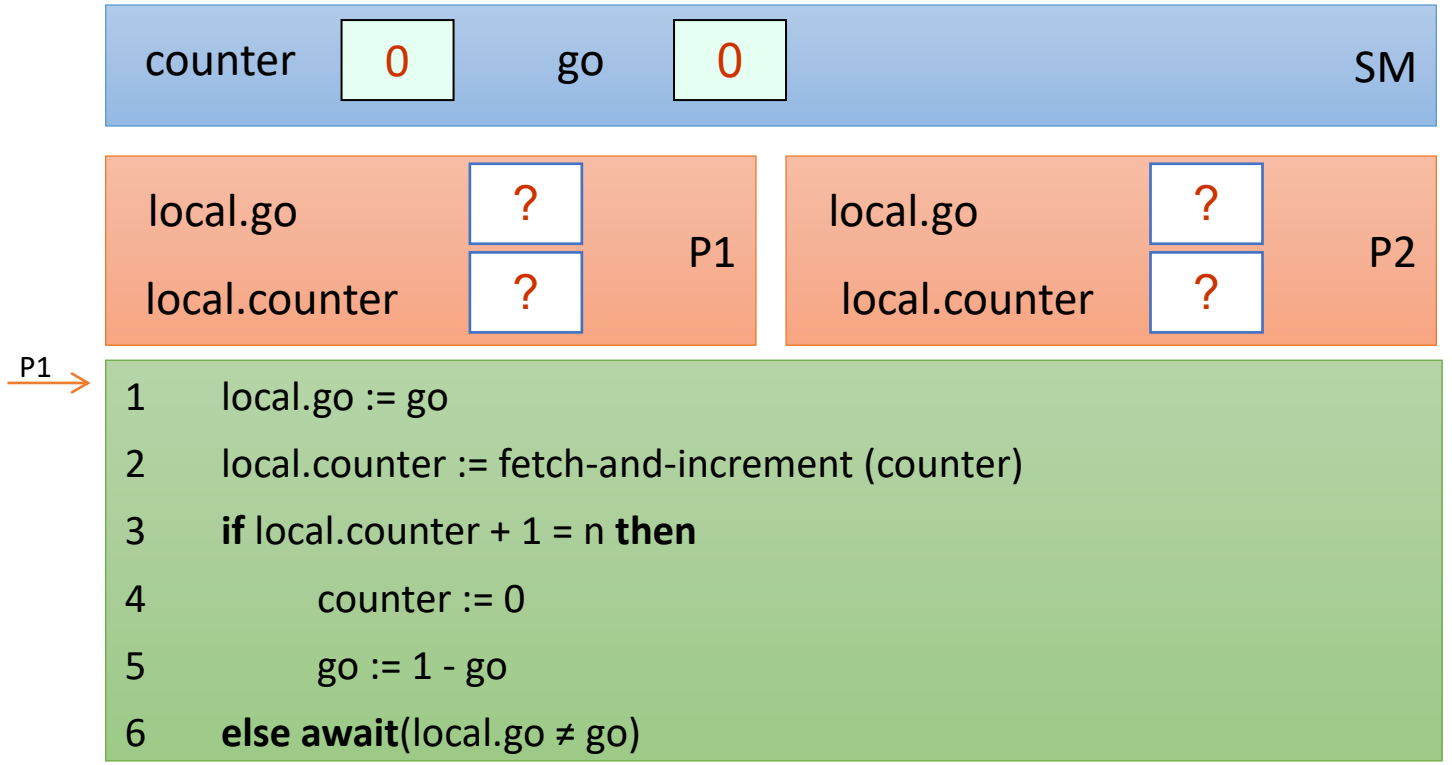
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



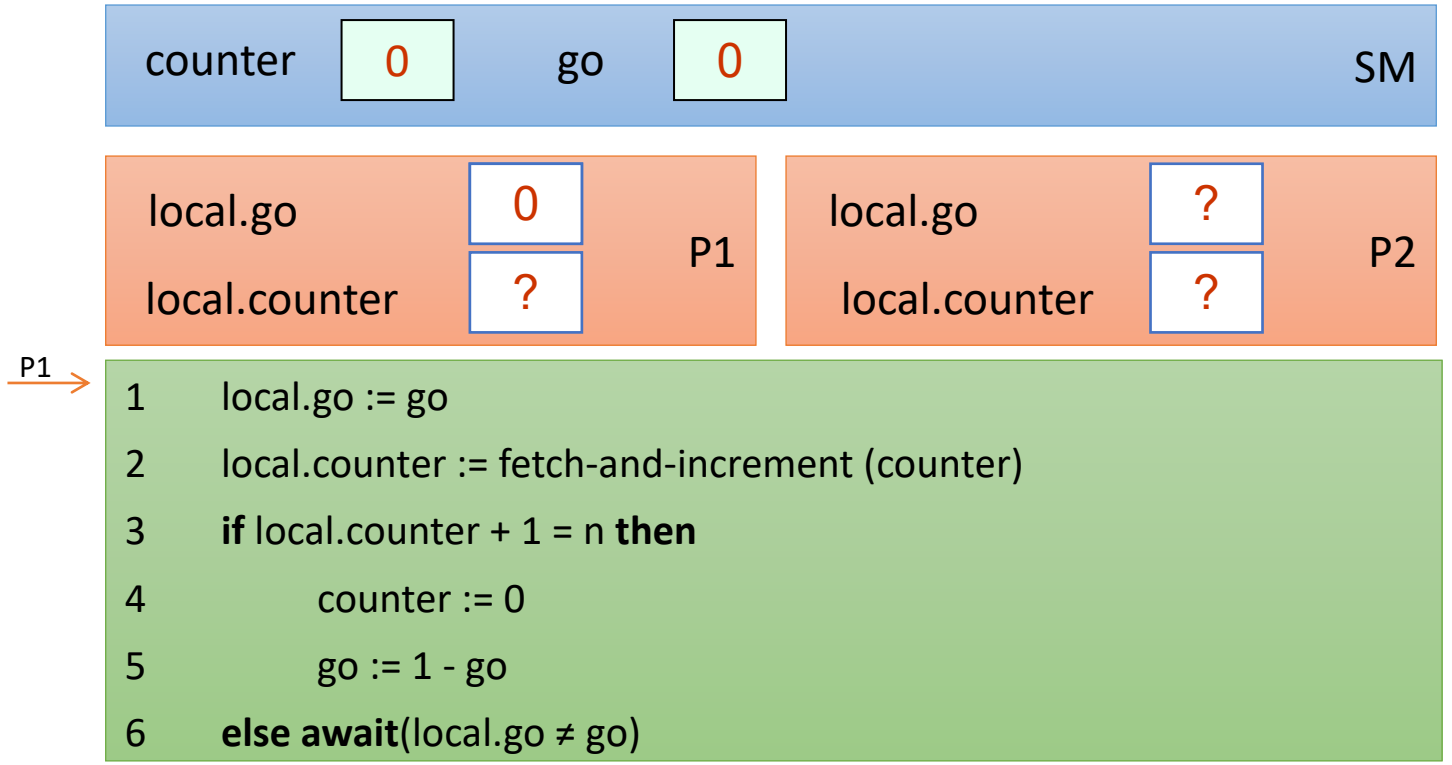
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



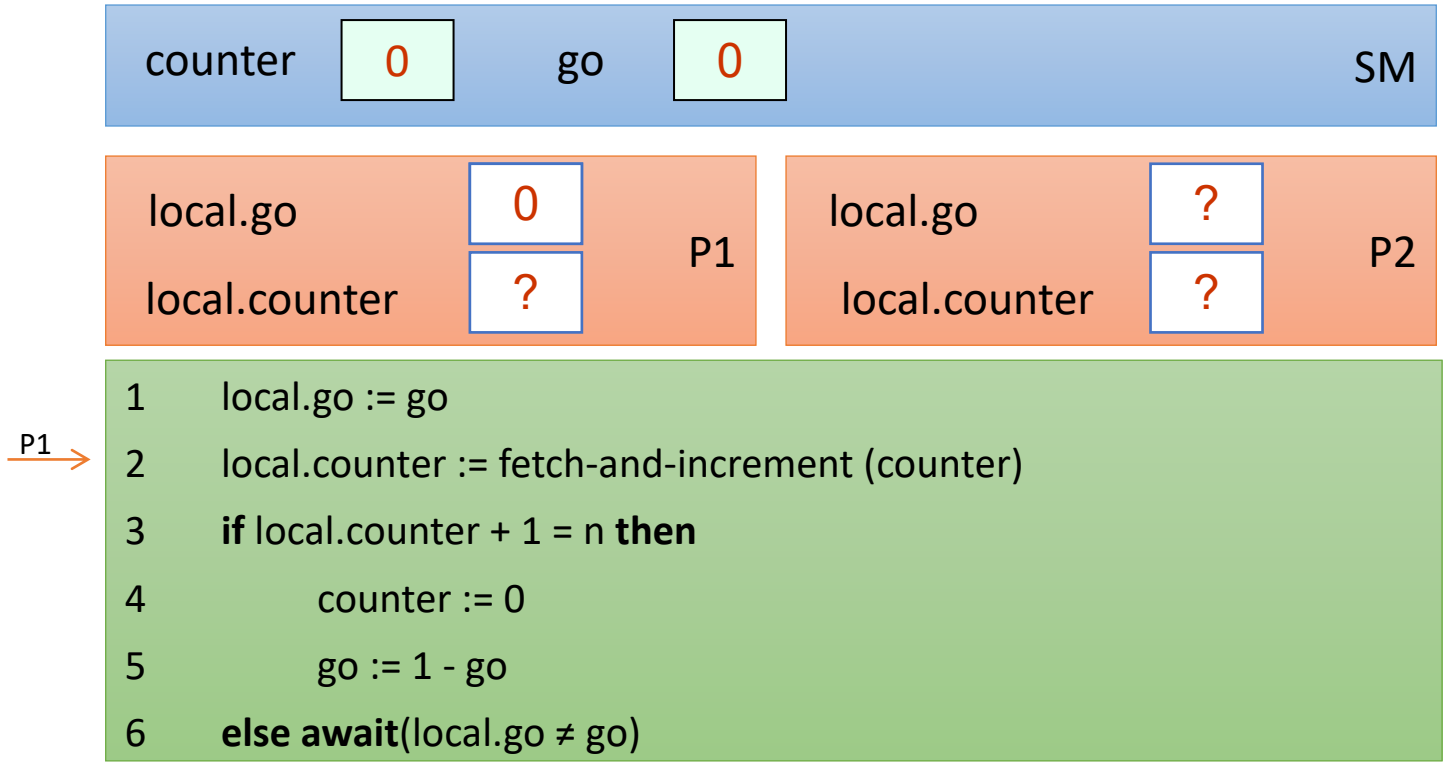
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



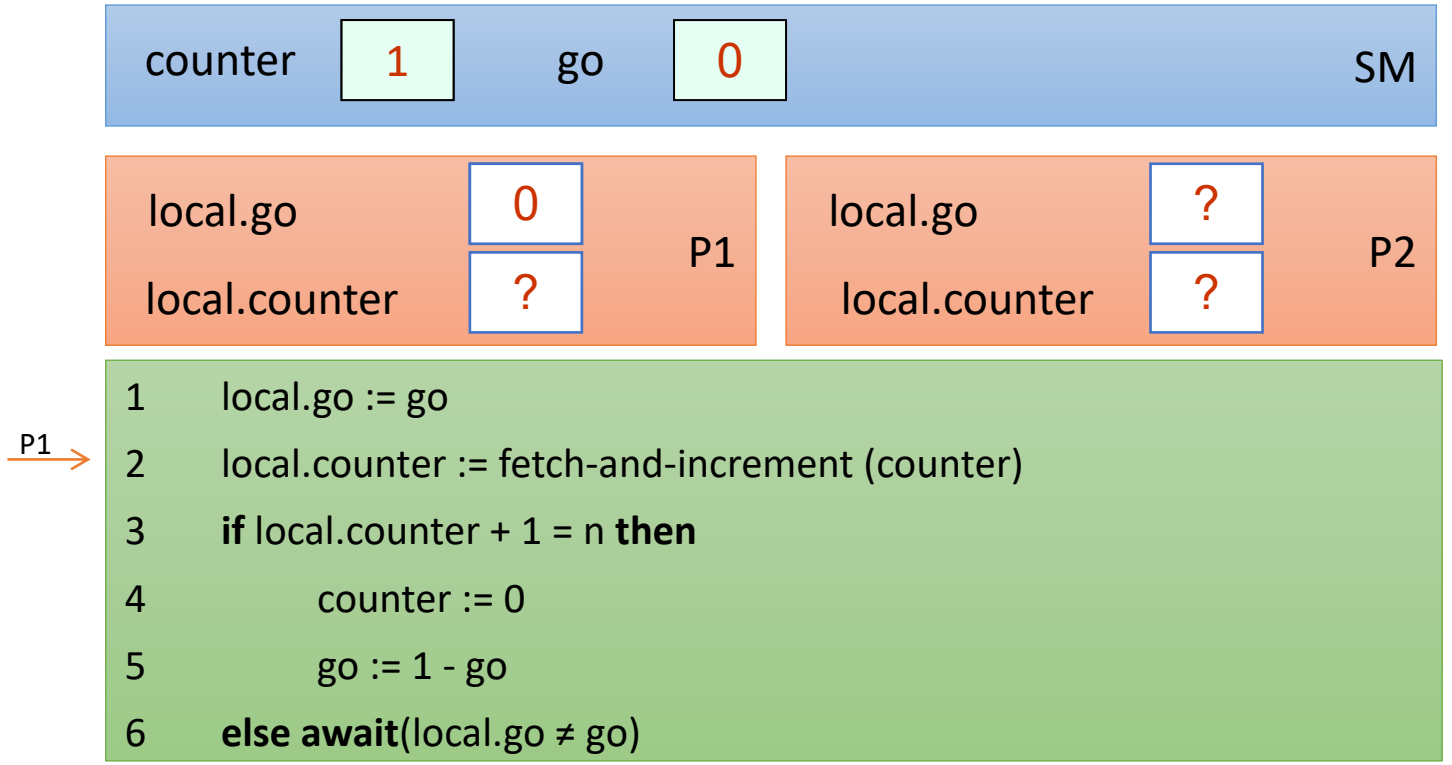
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



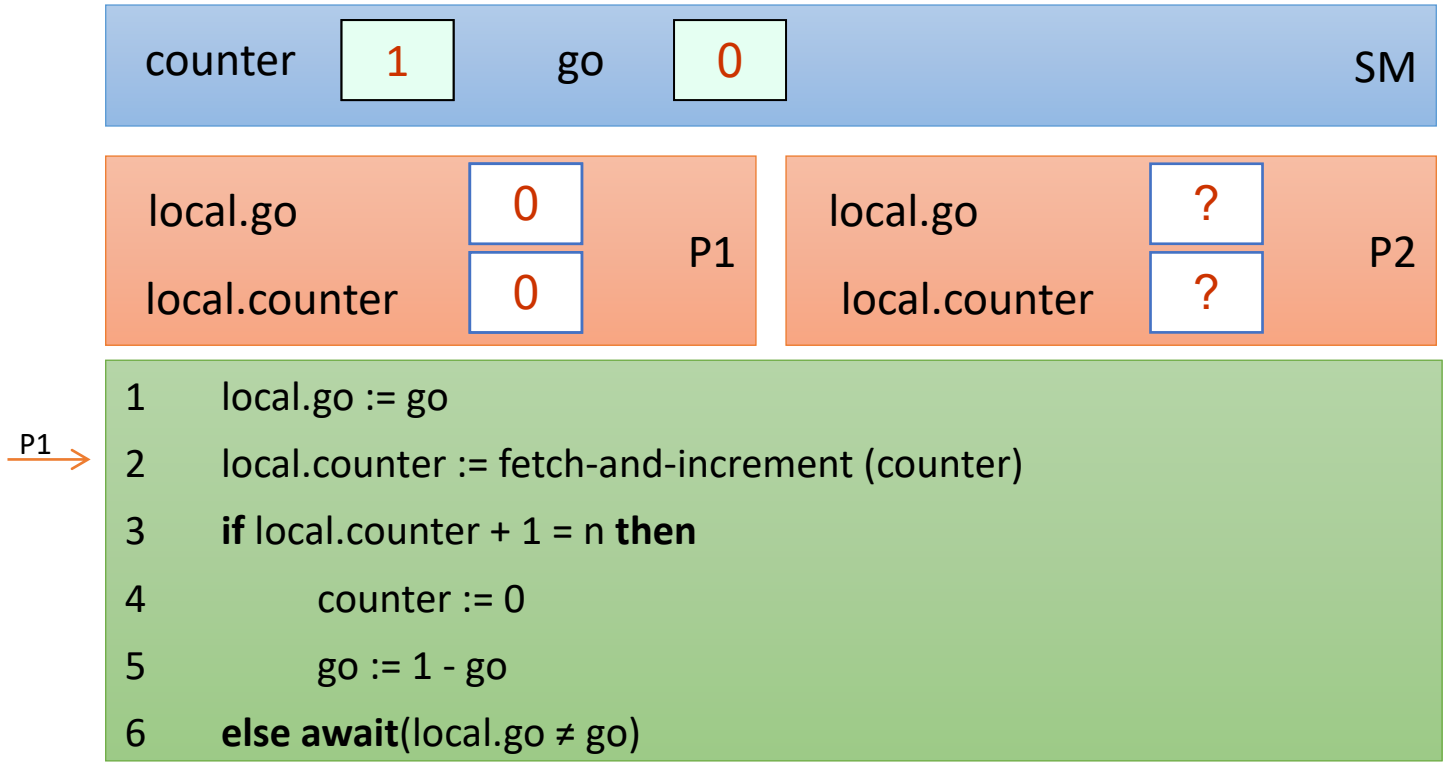
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



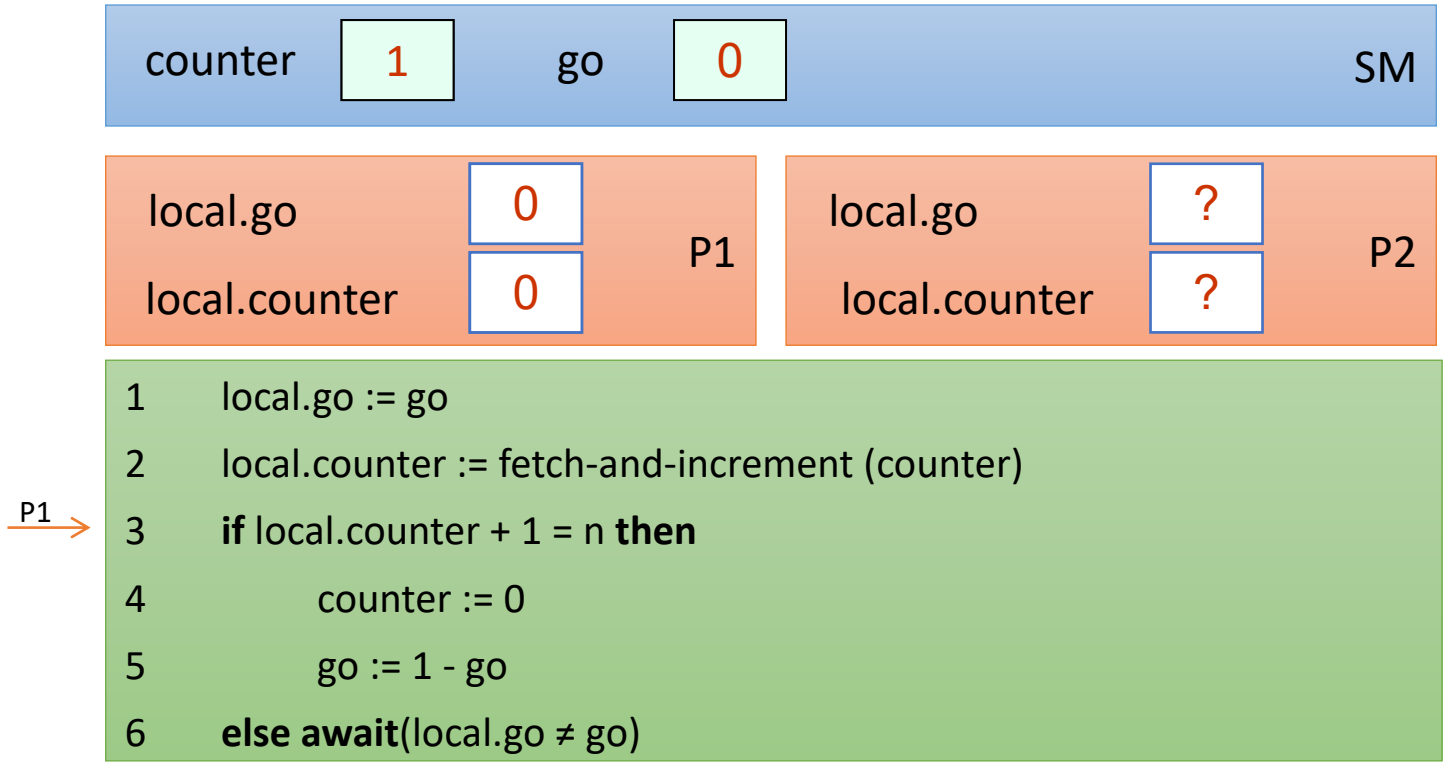
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



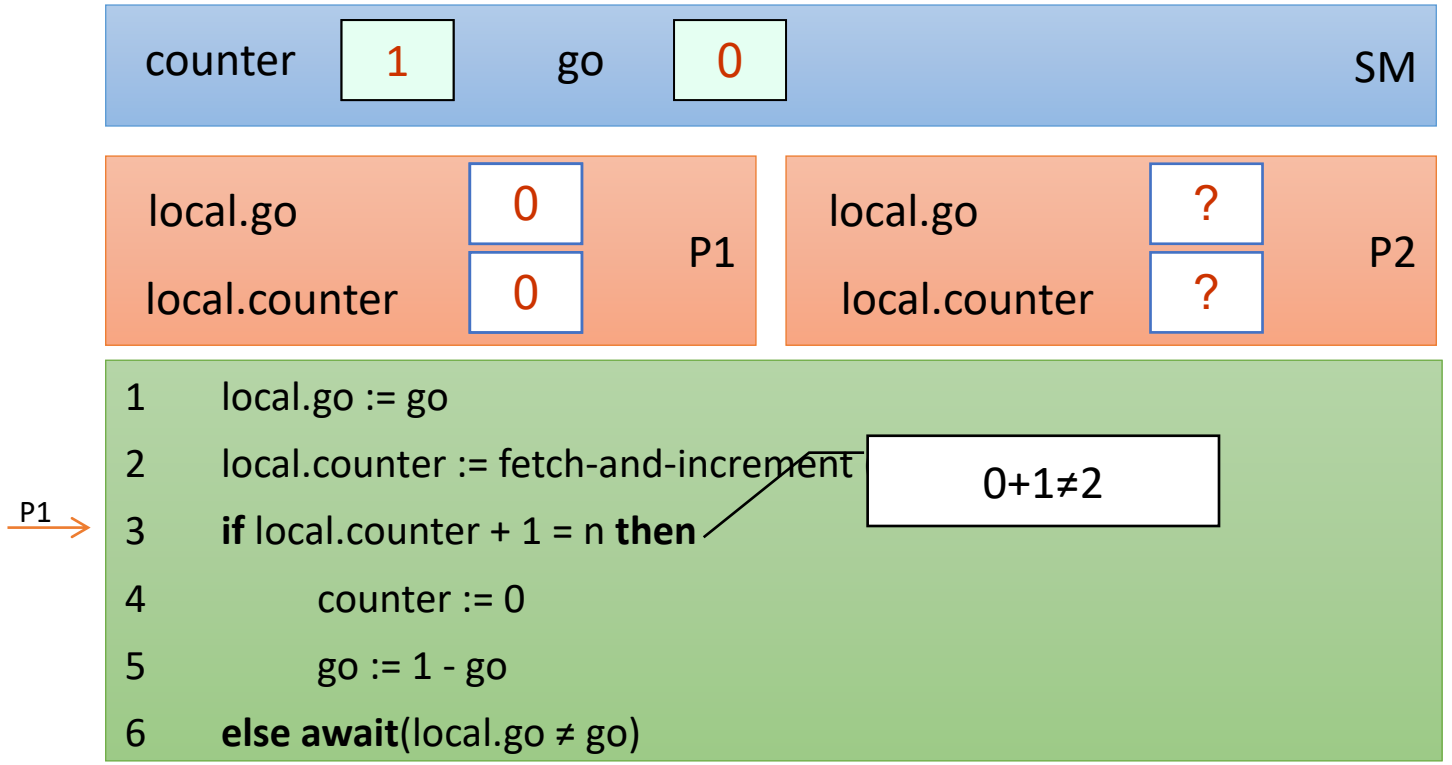
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



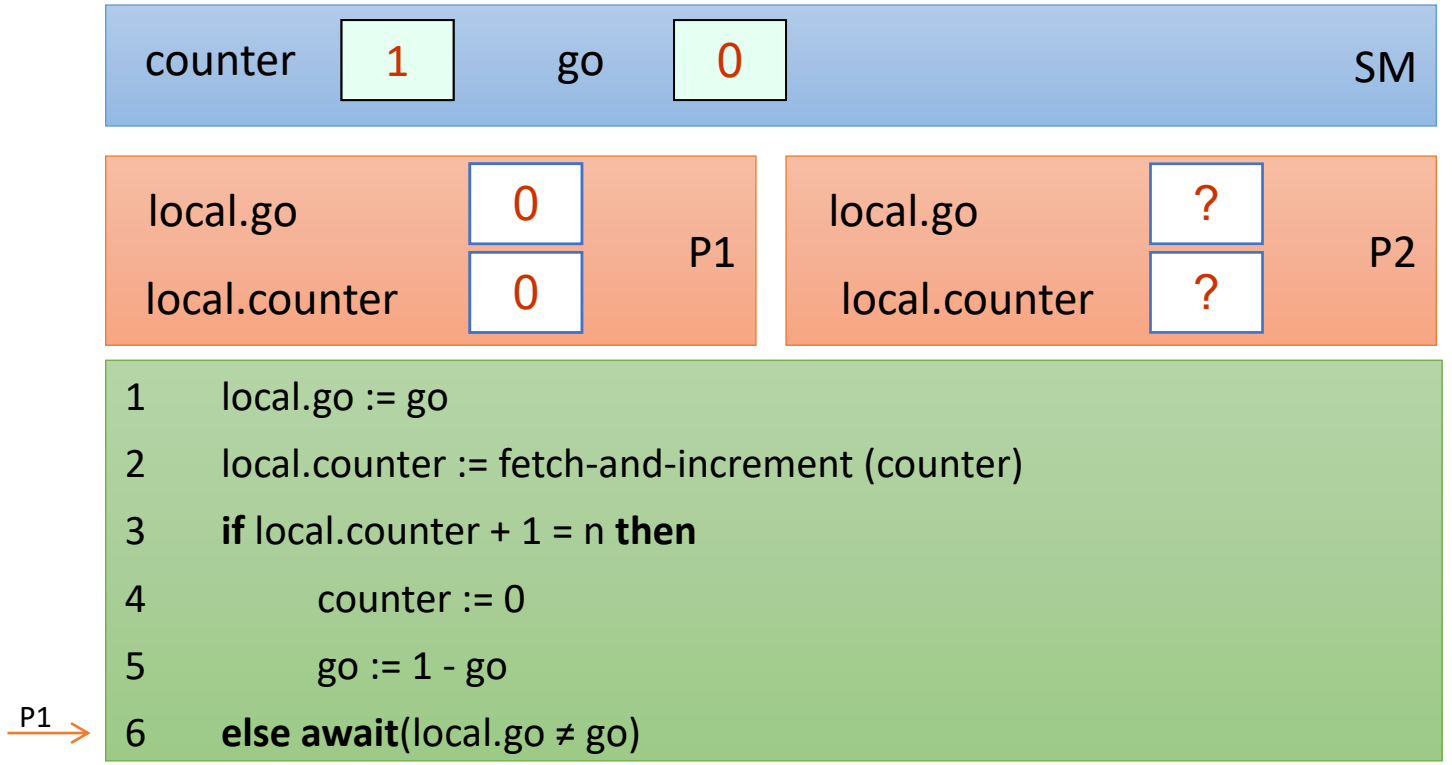
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



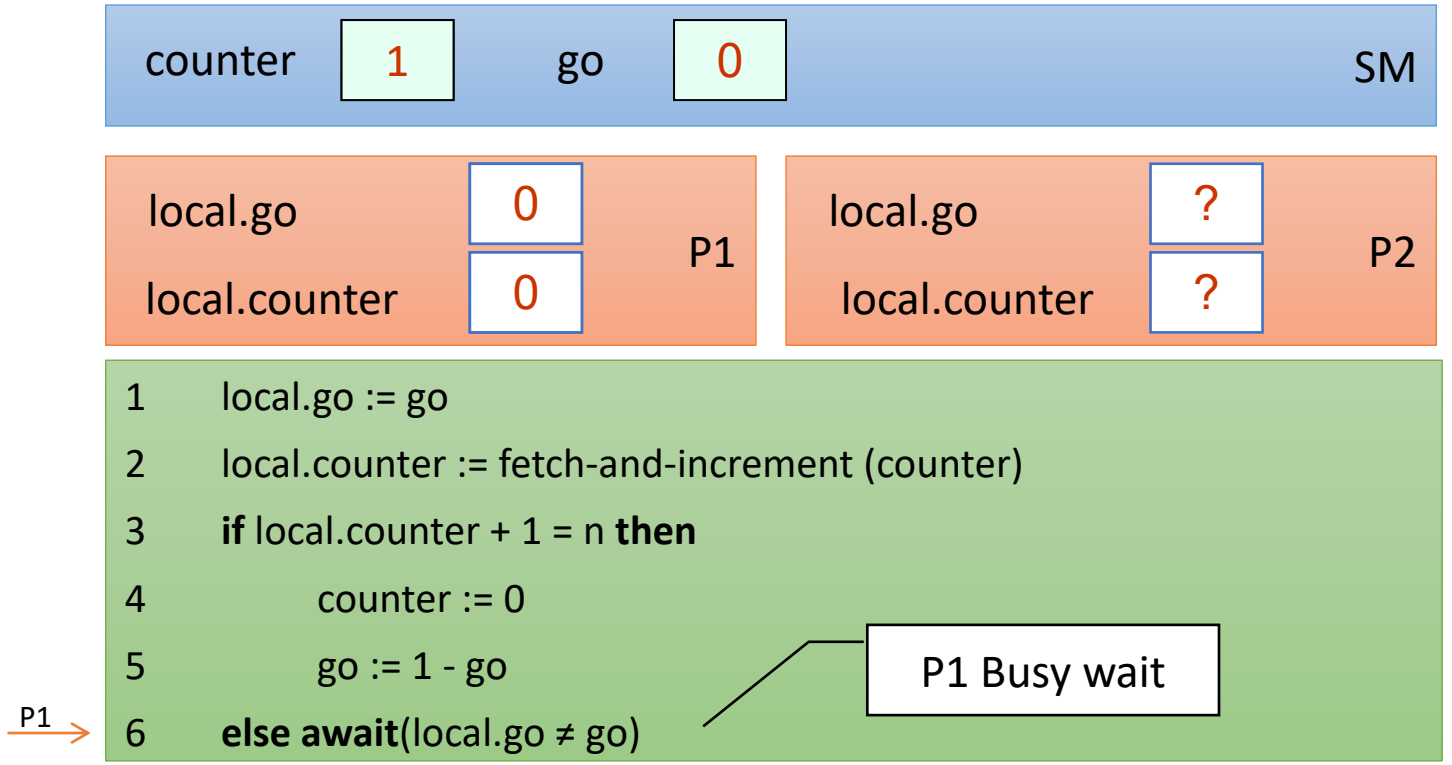
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



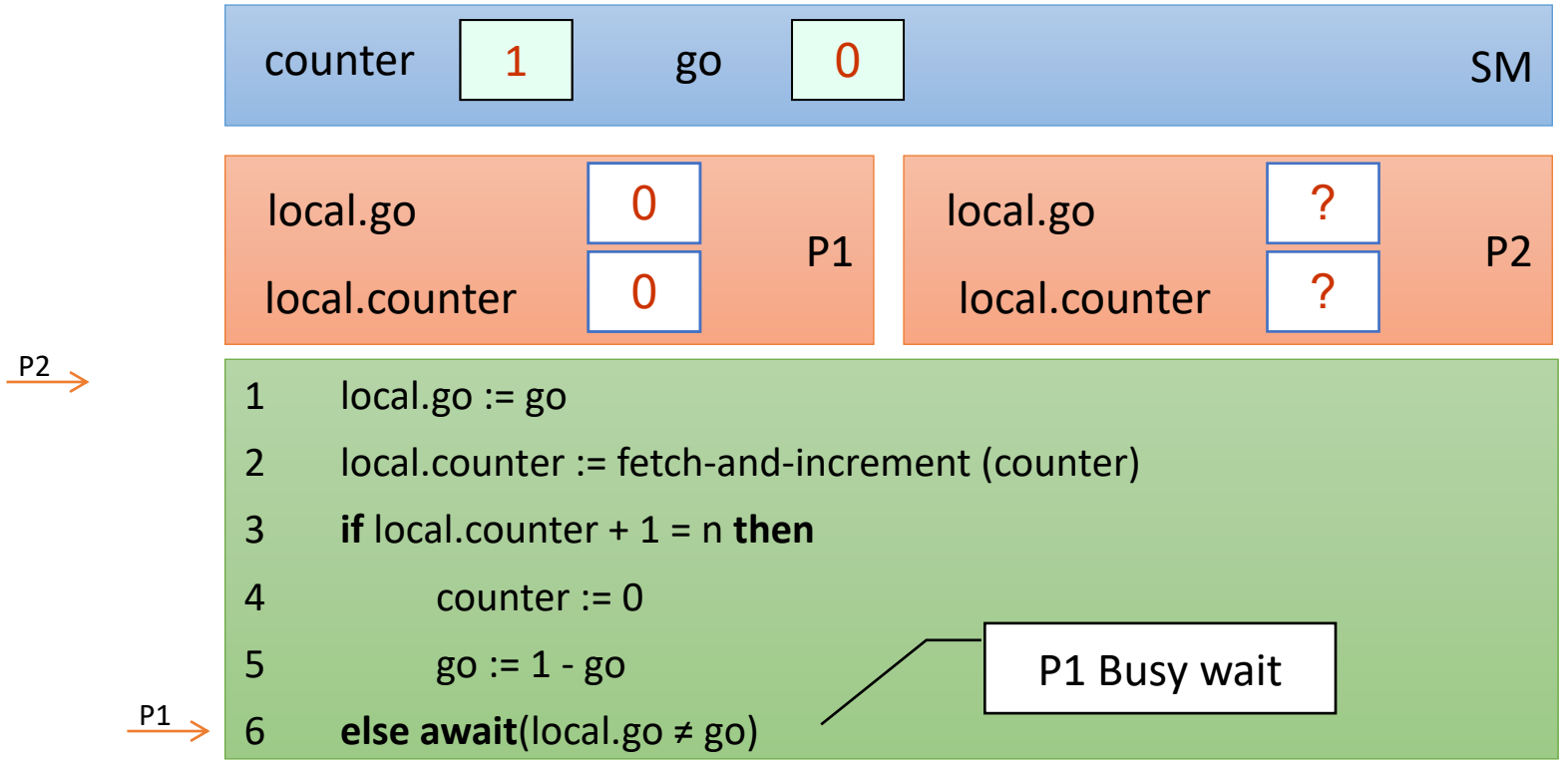
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



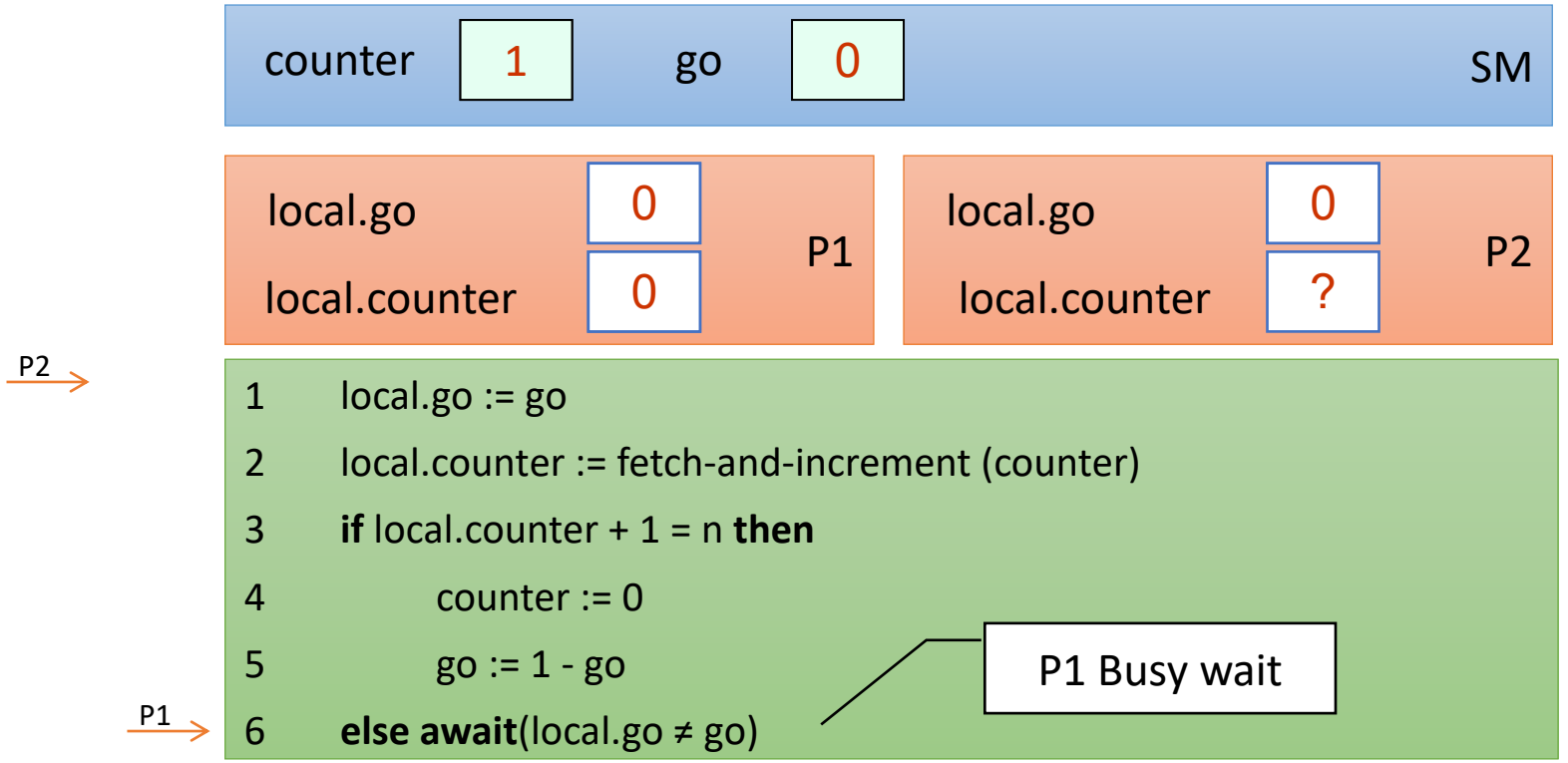
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



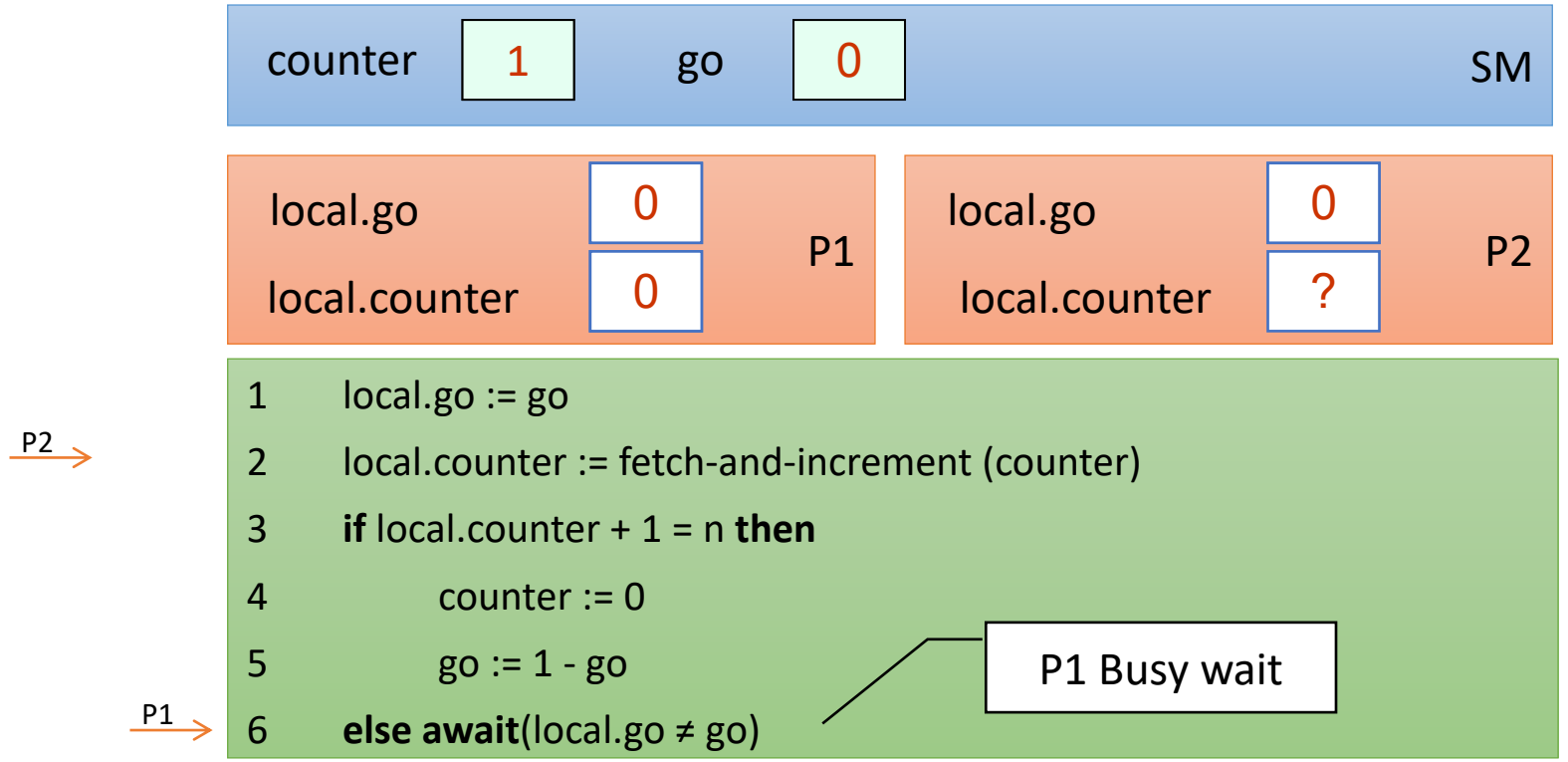
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



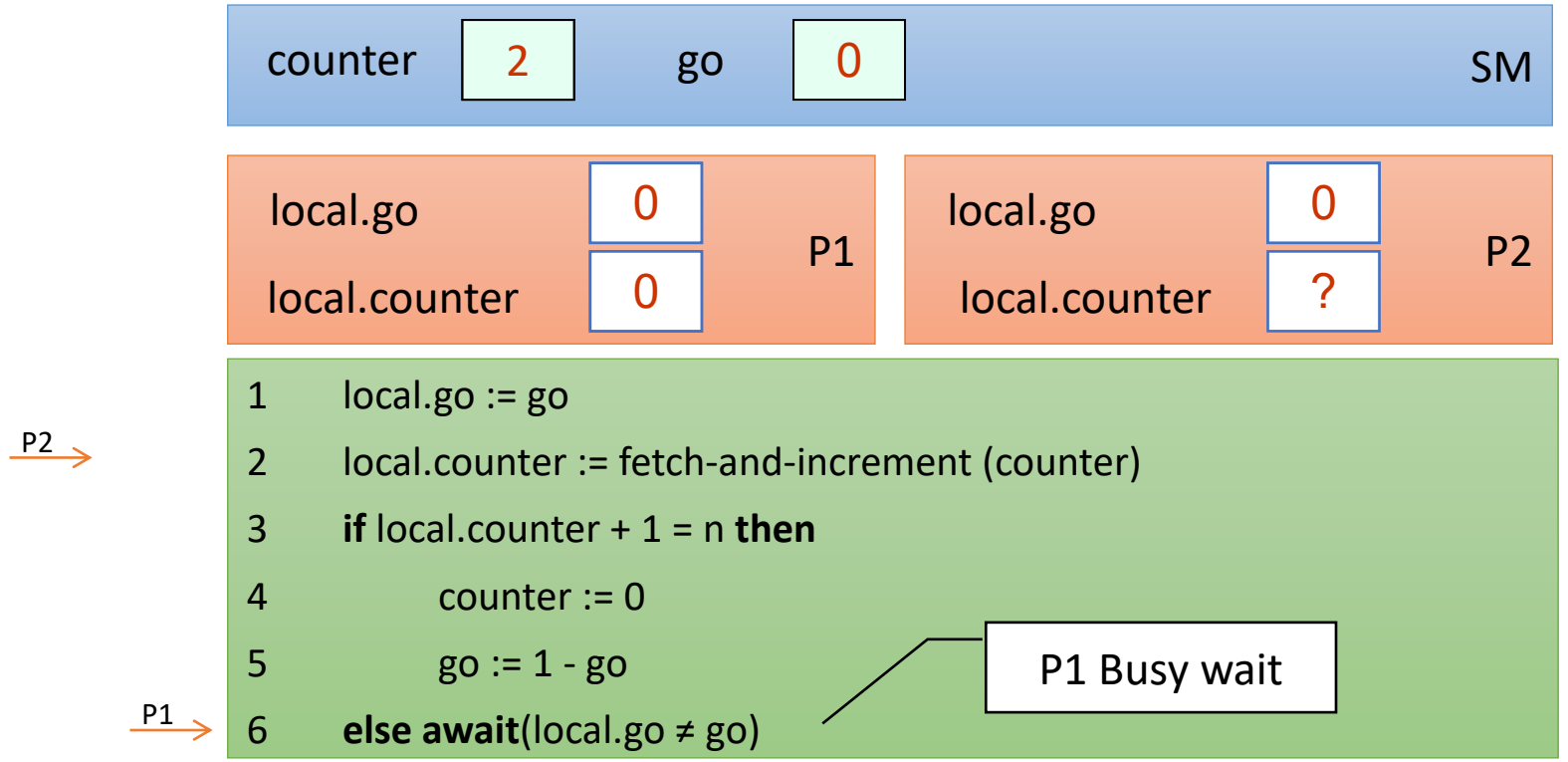
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



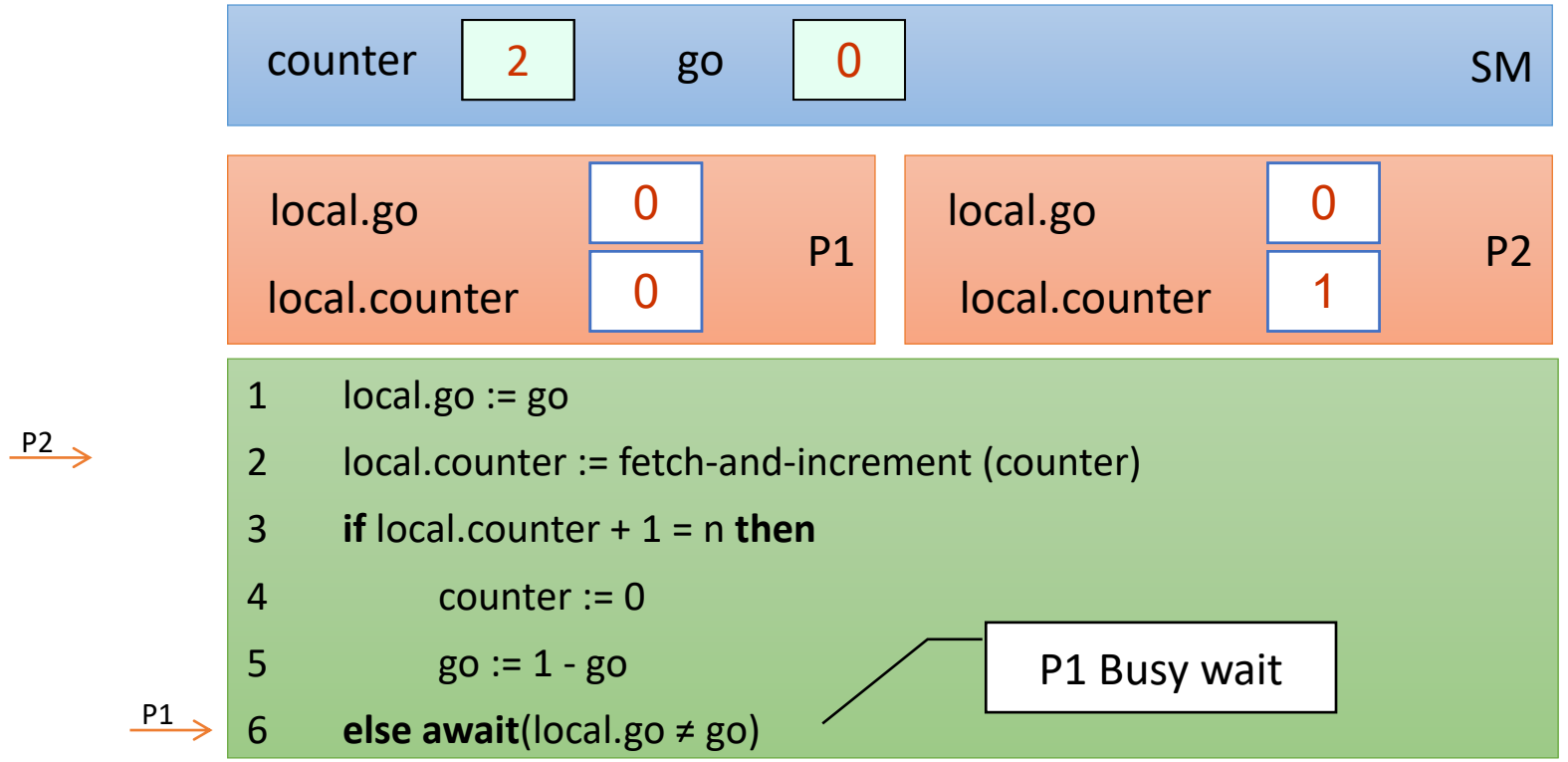
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



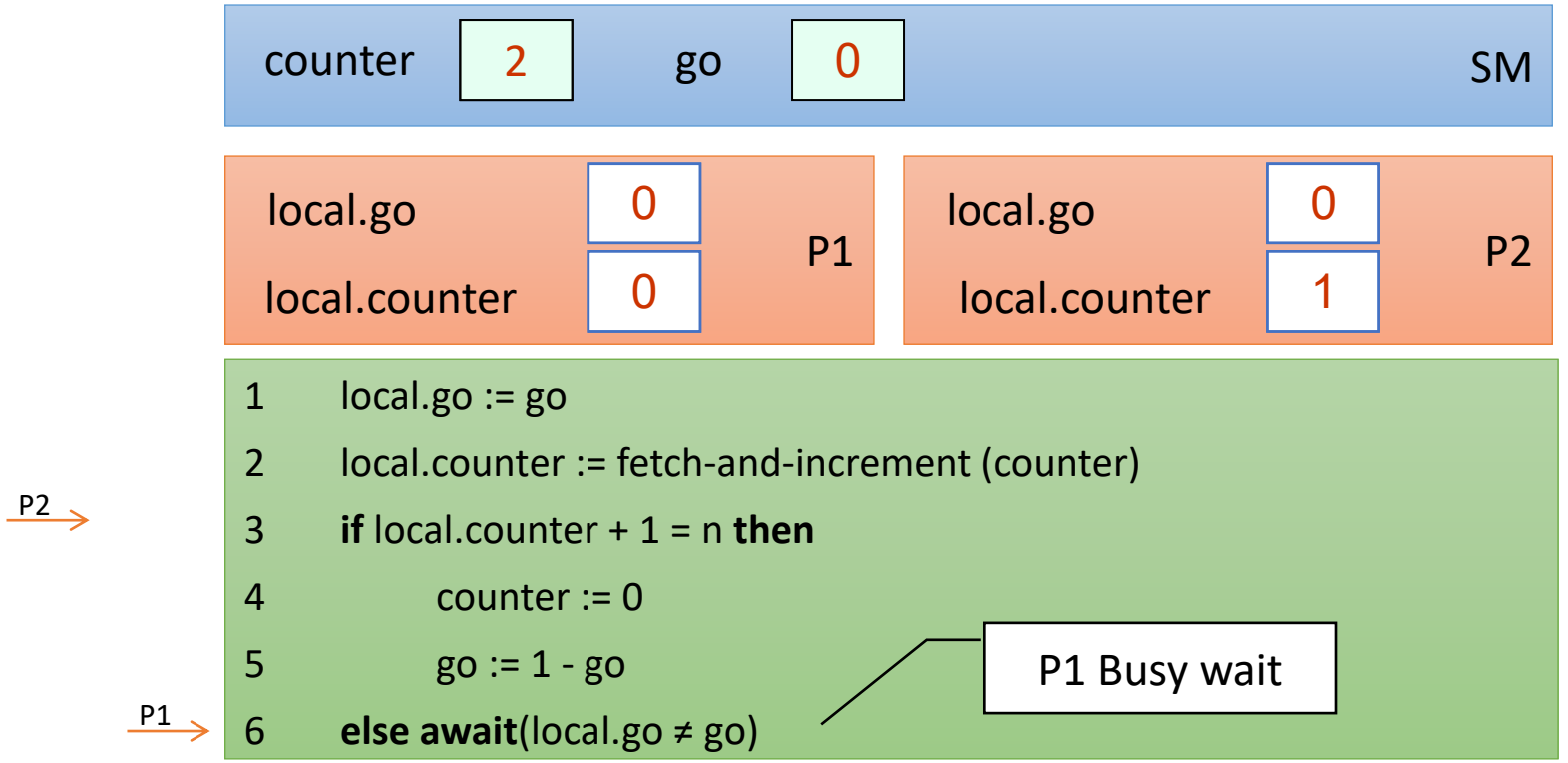
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



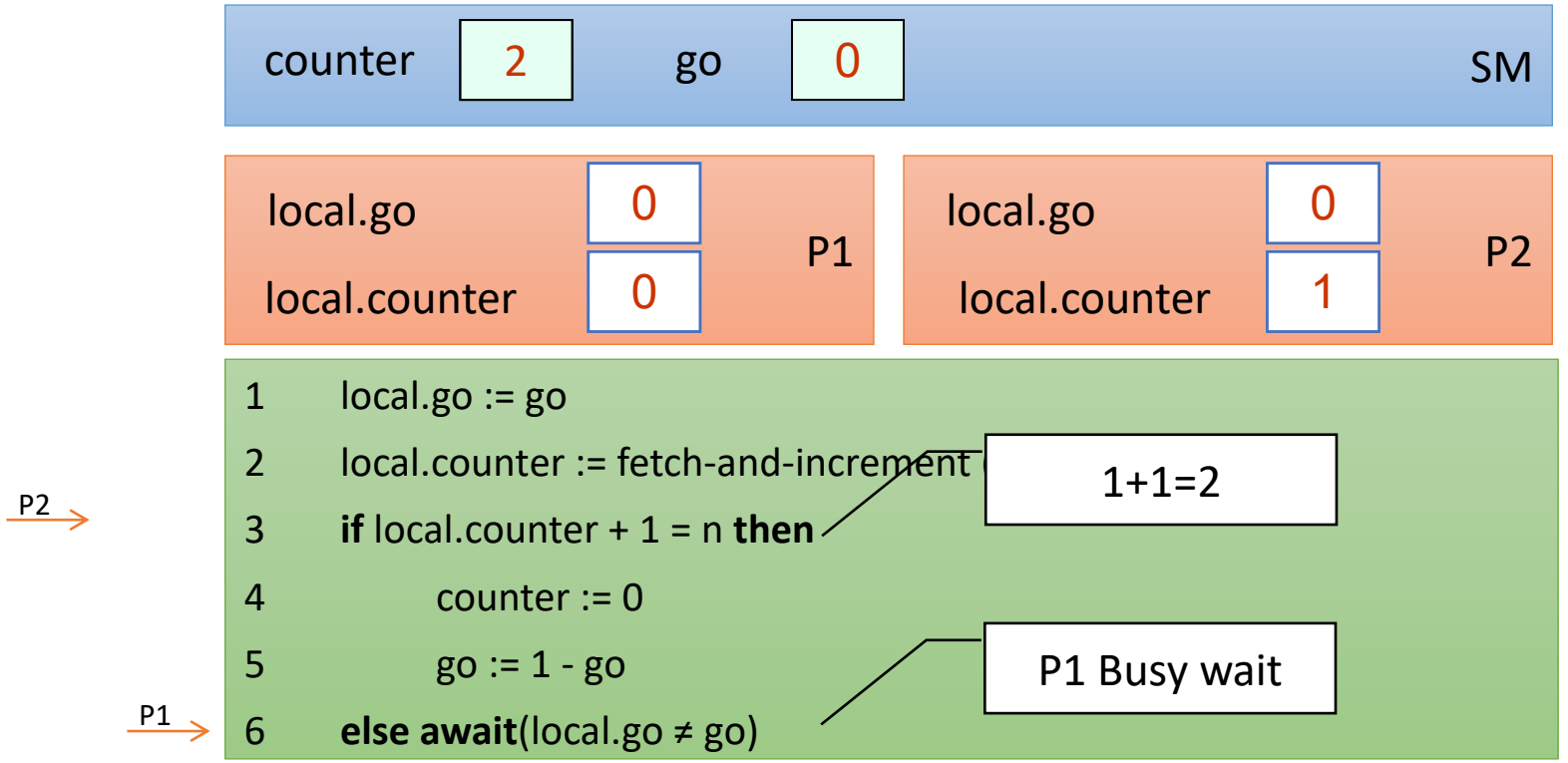
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



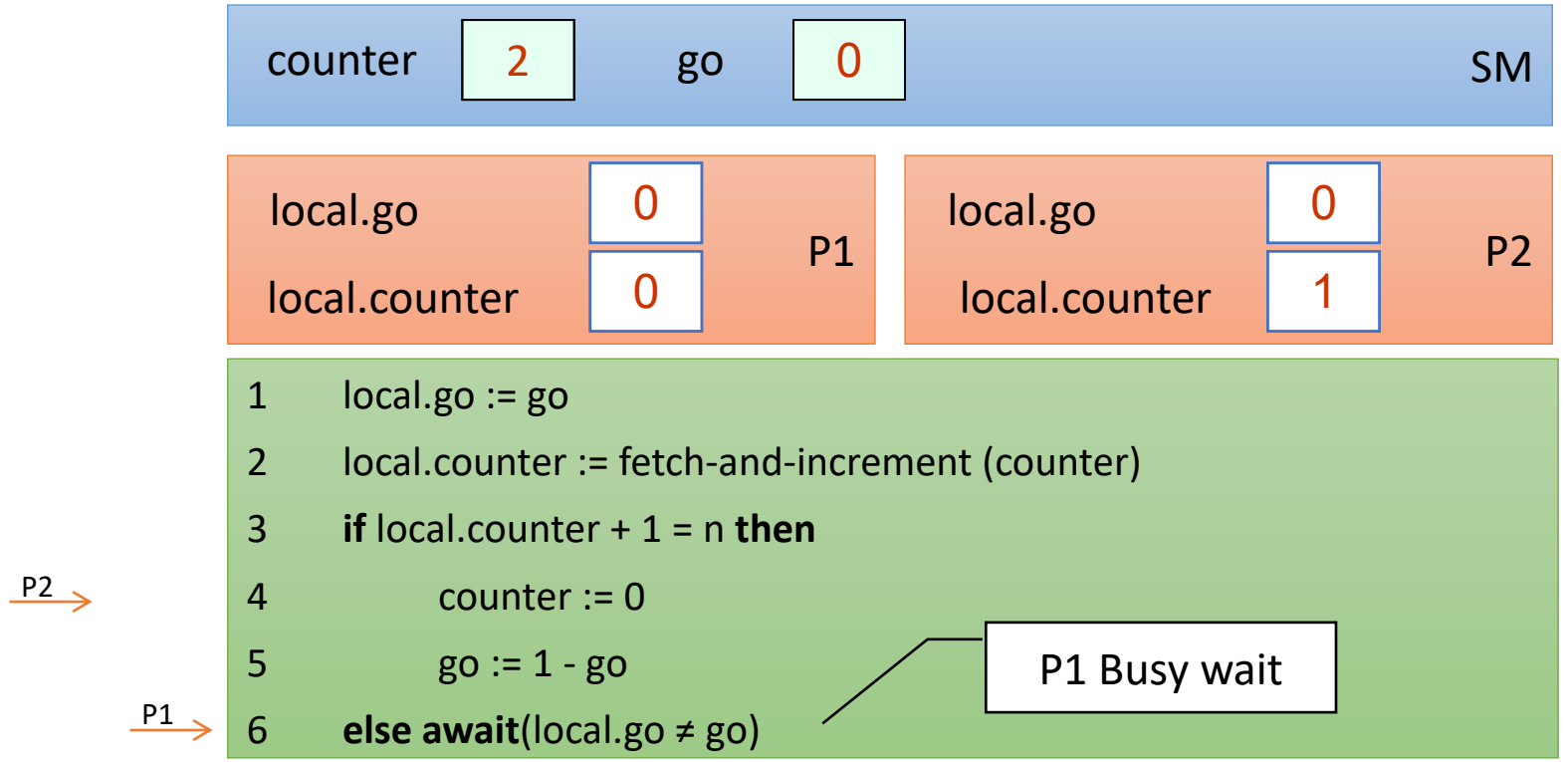
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



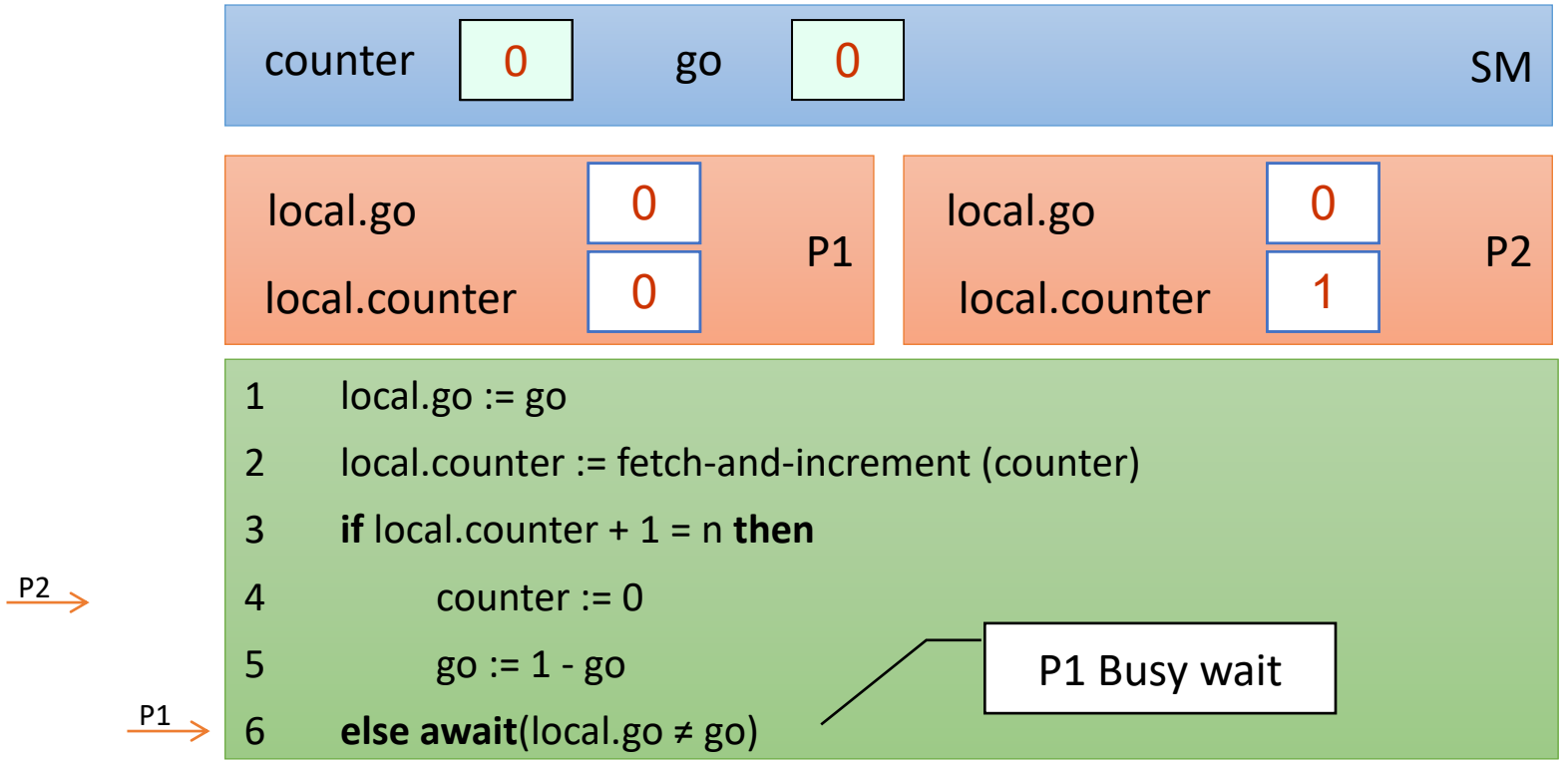
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



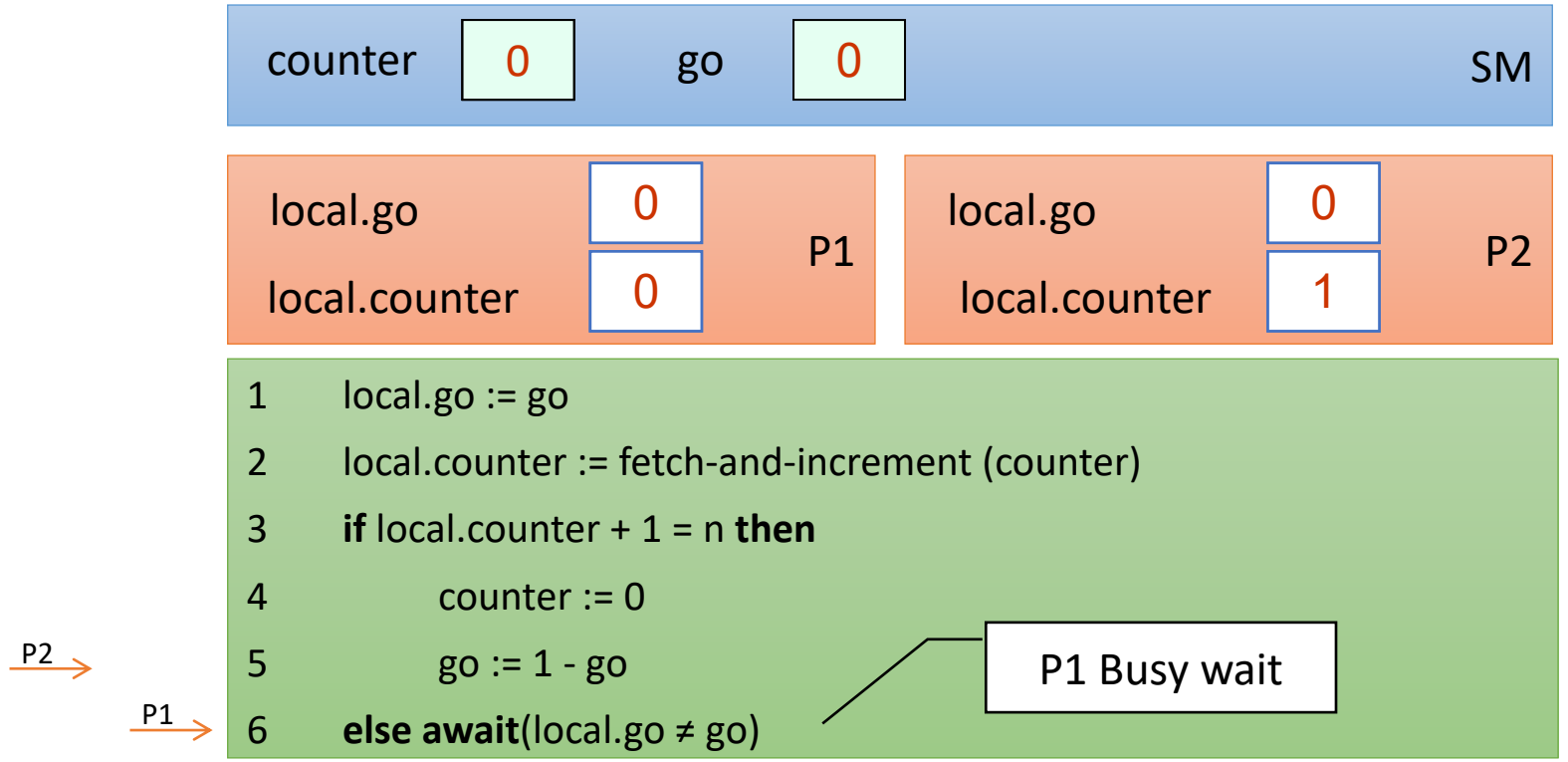
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



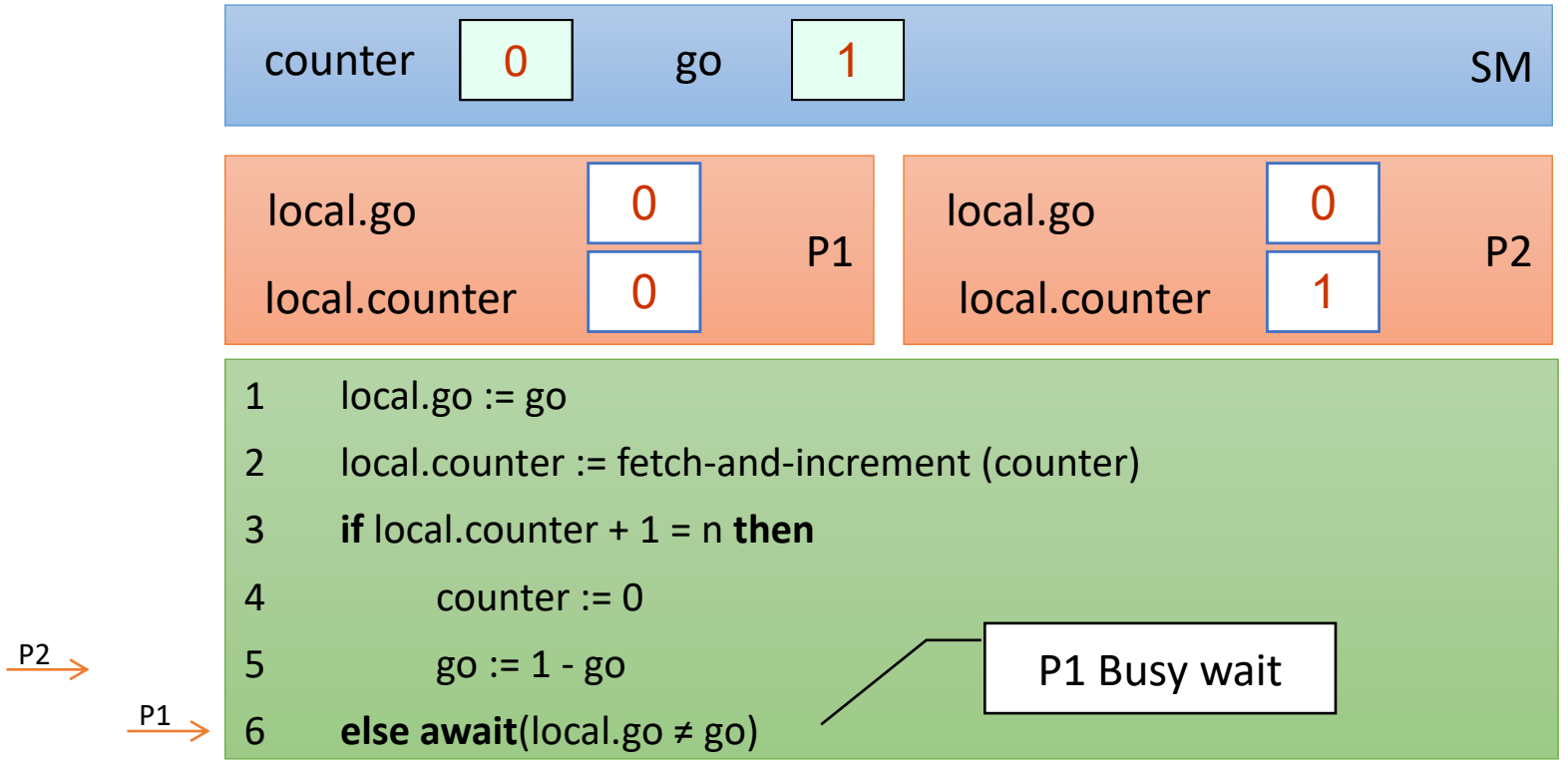
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



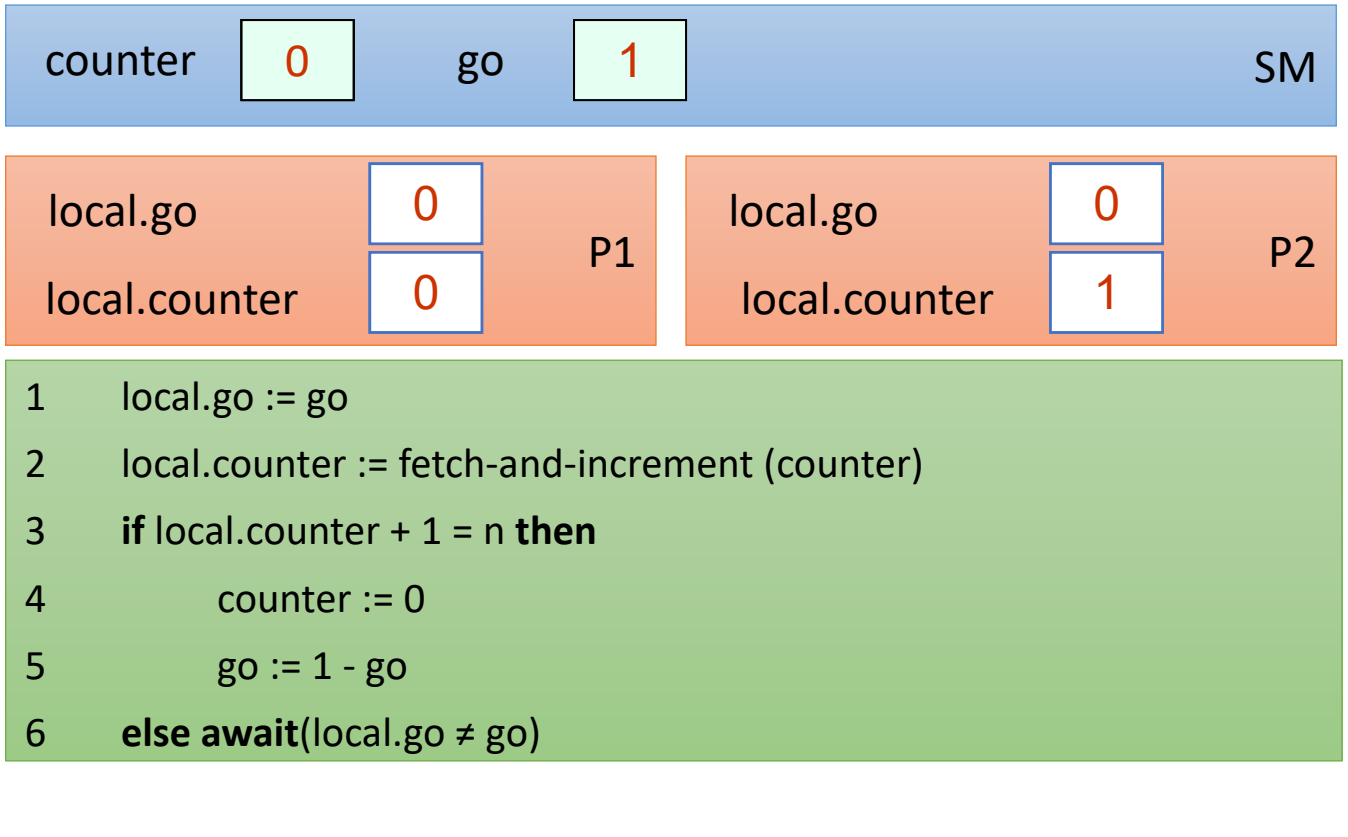
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



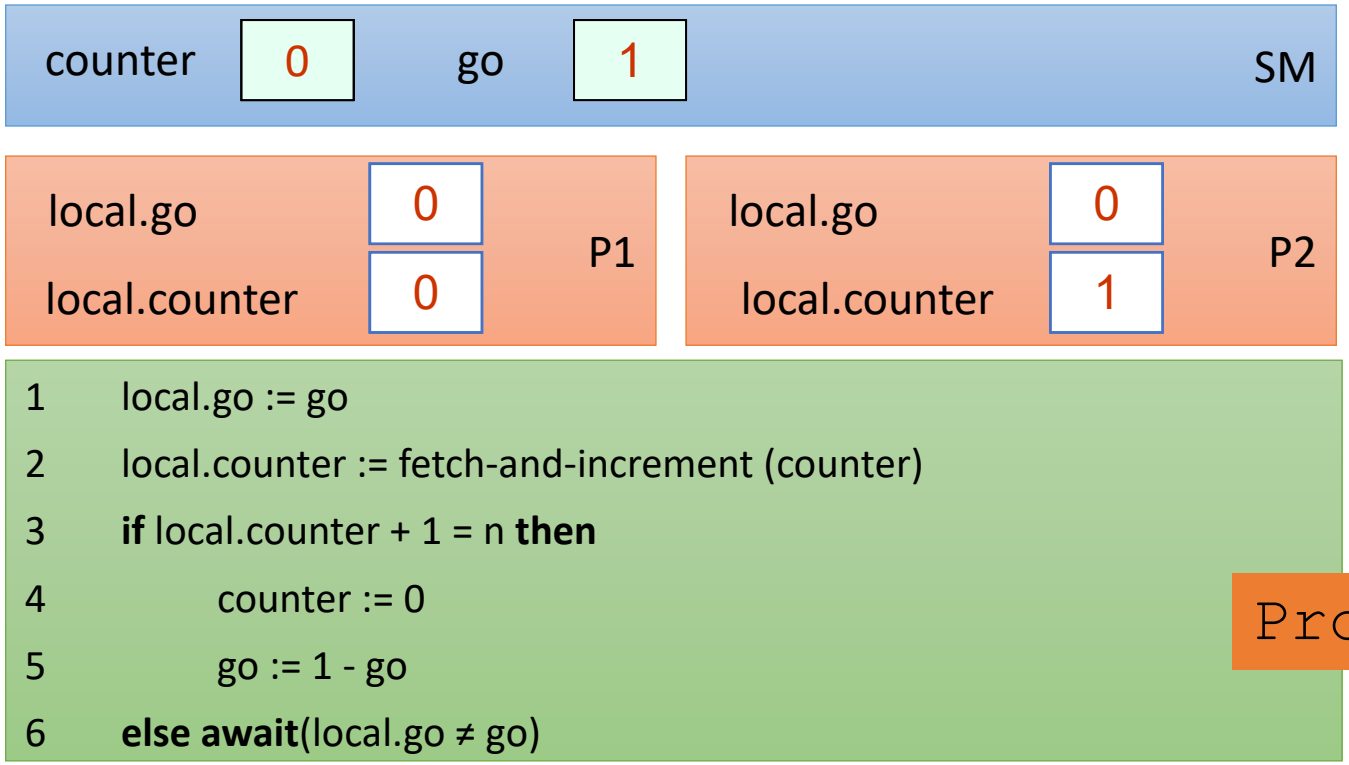
Simple Barrier Using an Atomic Counter

Run for n=2 Threads



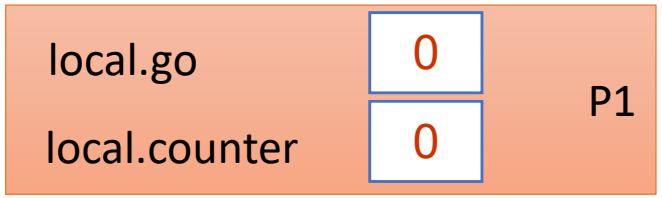
Simple Barrier Using an Atomic Counter

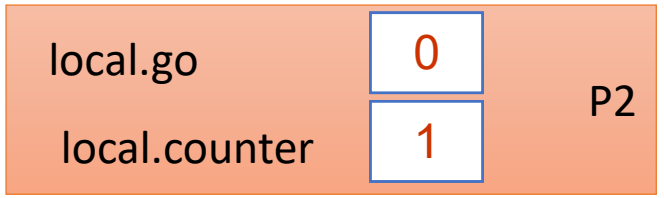
Run for n=2 Threads



Simple Barrier Using an Atomic Counter

Run for n=2 Threads



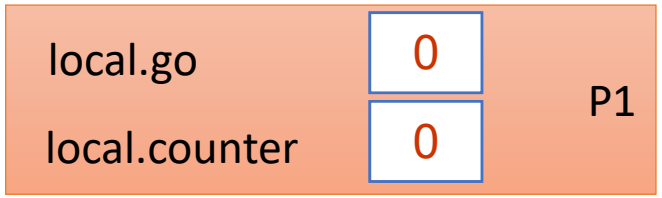


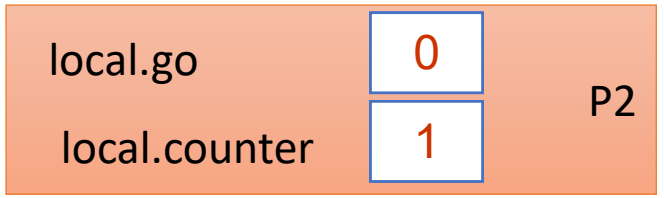

```
1    local.go := go
2    local.counter := fetch-and-increment (counter)
3    if local.counter + 1 = n then
4            counter := 0
5            go := 1 - go
6    else await(local.go ≠ go)
```

Pros/Cons?

Simple Barrier Using an Atomic Counter

Run for n=2 Threads





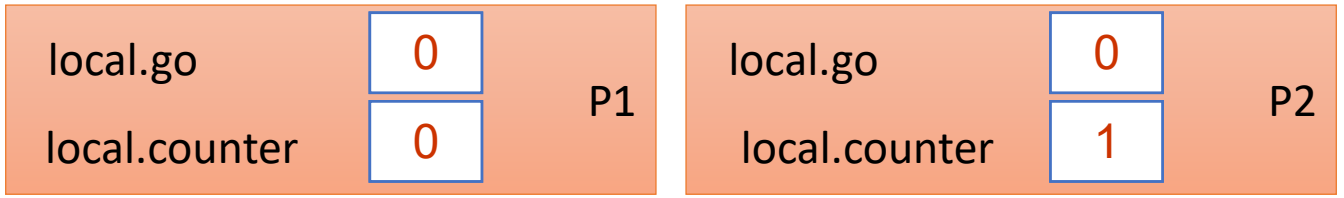

```
1    local.go := go
2    local.counter := fetch-and-increment (counter)
3    if local.counter + 1 = n then
4            counter := 0
5            go := 1 - go
6    else await(local.go ≠ go)
```

Pros/Cons?

- There is high memory contention on *go* bit

Simple Barrier Using an Atomic Counter

Run for n=2 Threads




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

P2 → P1 →

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*]

Questions?