departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrency and Parallelism
(Concorrência e Paralelismo – CP 11158)

Lecture 2
— Lets talk about synchronization —

Slides based in material from:
Gadi Taubenfeld (http://www.faculty.idc.ac.il/gadi/book.htm)

Synchronization

- Type of synchronization
  - Contention
  - Coordination
- Why synchronization is Difficult?
  - Easy between humans
  - Communication between computers is restricted
    - reading and writing of notes
    - sending and receiving messages

The Too-Much-Milk Problem

Time	Alice	Bob
5:00	Arrive Home	
5:05	Look in fridge; no milk	
5:10	Leave for grocery	
5:15		Arrive home
5:20	Arrive at grocery	Look in fridge; no milk
5:25	Buy milk	Leave for grocery
5:30	Arrive home; put milk in fridge	
3:40		Buy milk
3:45		Arrive home; too much milk!

Solving the Too-Much-Milk Problem

Correctness properties

- Mutual Exclusion:
  - Only one person buys milk "at a time"
- Progress:
  - Someone always buys milk if it is needed

Communication primitives

- Leave a note
  - set a flag
- Remove a note
  - reset a flag
- Read a note
  - test the flag

Important Distinction

- **Safety Property**
- Nothing bad happens ever
  - Example: mutual exclusion

- **Liveness Property**
- Something good happens eventually
  - Example: progress

Solution 1

Does it work?

Alice	Bob

if (no note) {
  if (no milk) {
  leave Note
  buy milk
  remove note
  }
}
	if (no note) {
	  if (no milk) {
	  leave Note
	  buy milk
	  remove note
	  }
	}

No, may end up with too much milk.

x mutual exclusion
✓progress

Solution 2: Using labeled notes

Does it work?

Alice	Bob

leave note A
if (no note B) {
  if (no milk) {buy milk}
}
remove note A
	leave note B
	if (no note A) {
	  if (no milk) {buy milk}
	}
	remove note B

No, may end up with no milk.

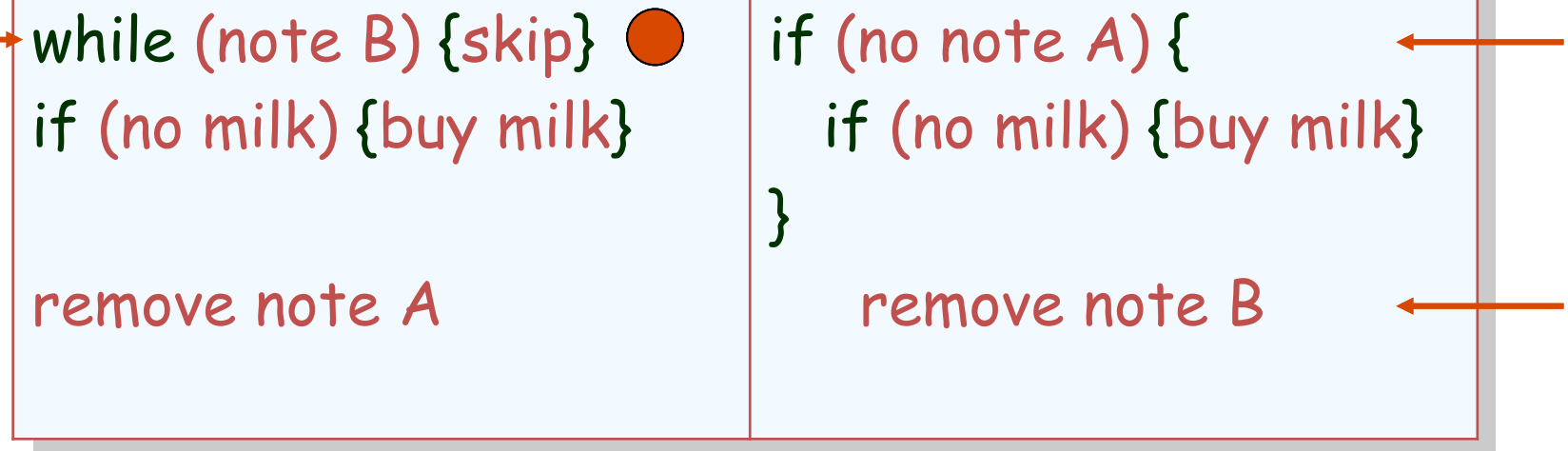
✓mutual exclusion
X progress

Solution 3

Bob's code is as before

Does it work?

Alice	Bob
leave note A	leave note B
while (note B) {skip}	if (no note A) {
if (no milk) {buy milk}	if (no milk) {buy milk}
	}
remove note A	remove note B



No ! a timing assumption is needed

✓mutual exclusion
X progress

Is solution #3 a good solution? No!

- A timing assumption is needed!
- Unfair to one of the processes
- It is asymmetric and complicated
- Alice is busy waiting — consuming CPU cycles without accomplishing useful work

Solution 3+

Does it work?

Alice code is as before

BUT, Bob may forever get stuck in the repeat-until loop !!!

Alice	Bob
	until (note B) if (no milk) {buy milk} remove note B

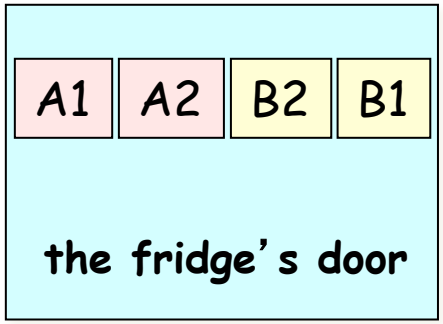
Additional requirement [informal definition]

Thirst-freedom: Nobody ever gets stuck forever.
(Both Alice and Bob eventually complete their programs.)

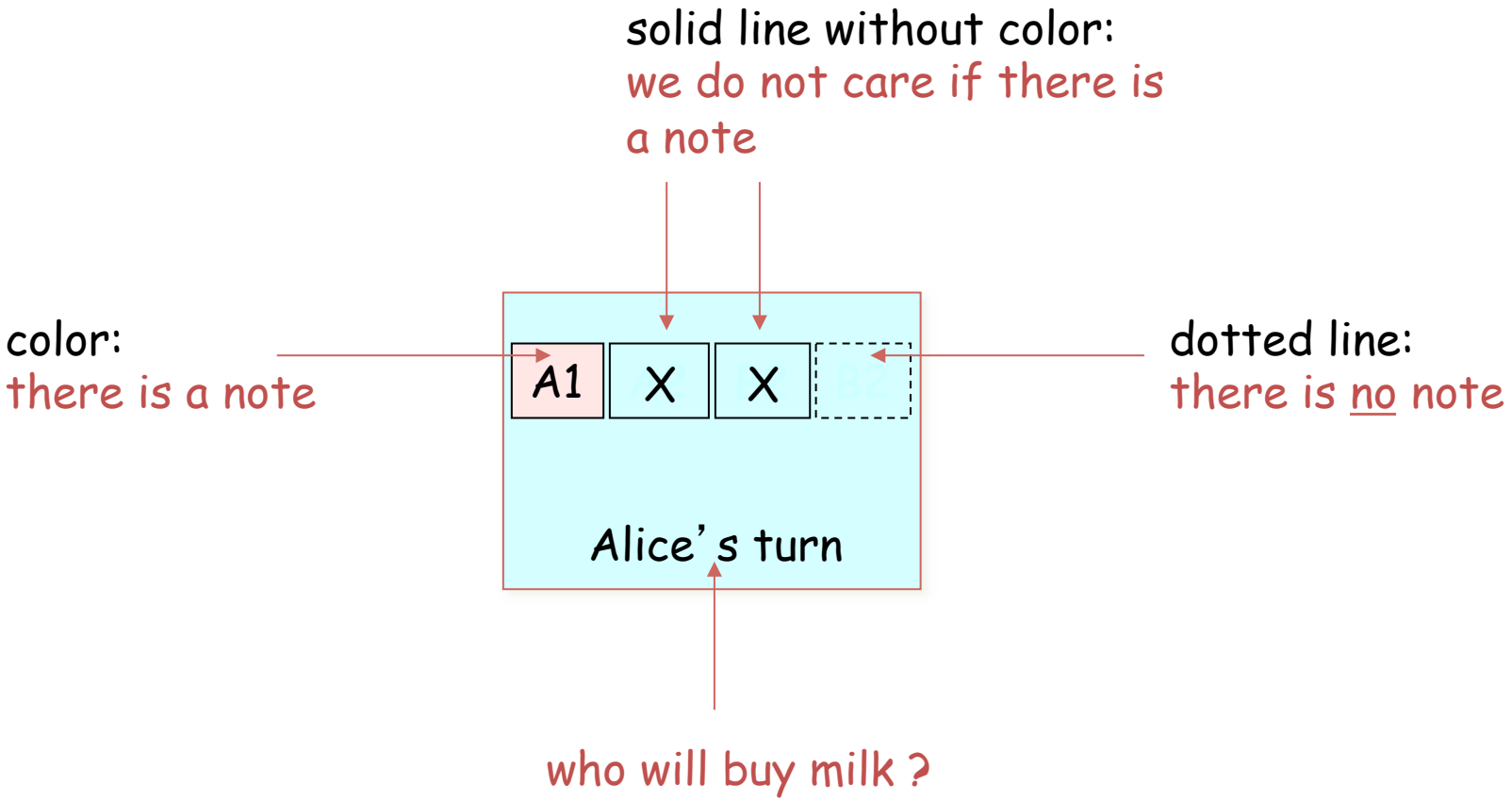
Yes, BUT ...

✓mutual exclusion
✓progress

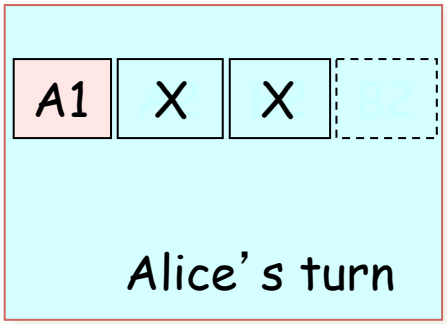
Solution 4: Using 4 labeled notes

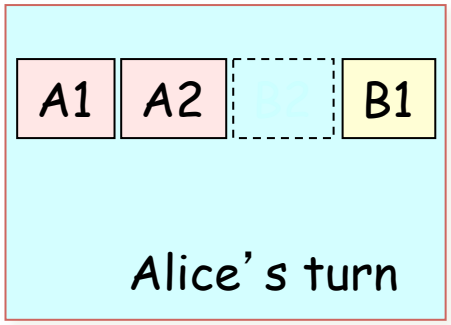


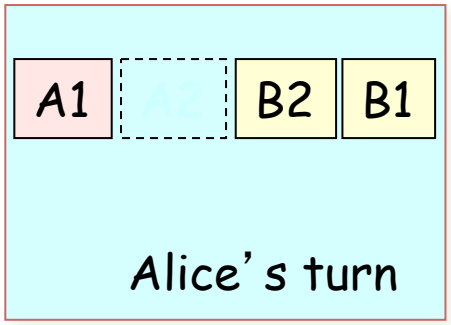
Solution 4: Notations

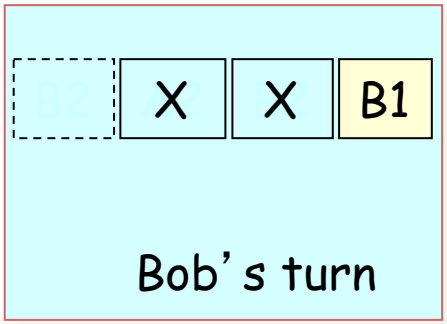


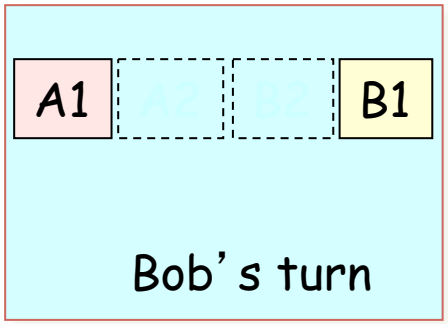
Solution 4

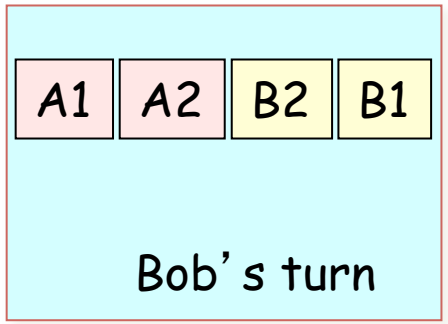

A1
X
X
Alice's turn


A1
A2
B1
Alice's turn


A1
B2
B1
Alice's turn


X
X
B1
Bob's turn


A1
B1
Bob's turn


A1
A2
B2
B1
Bob's turn

Solution 4

A correct, fair, symmetric algorithm!

- ✓ mutual exclusion
- ✓ progress
- ✓ both Alice and Bob eventually complete their programs

Alice

```
leave A1
if B2 {leave A2} else {remove A2}
while B1 and ((A2 and B2) or
              (no A2 and no B2))
      {skip}
if (no milk) {buy milk}
remove A1
```

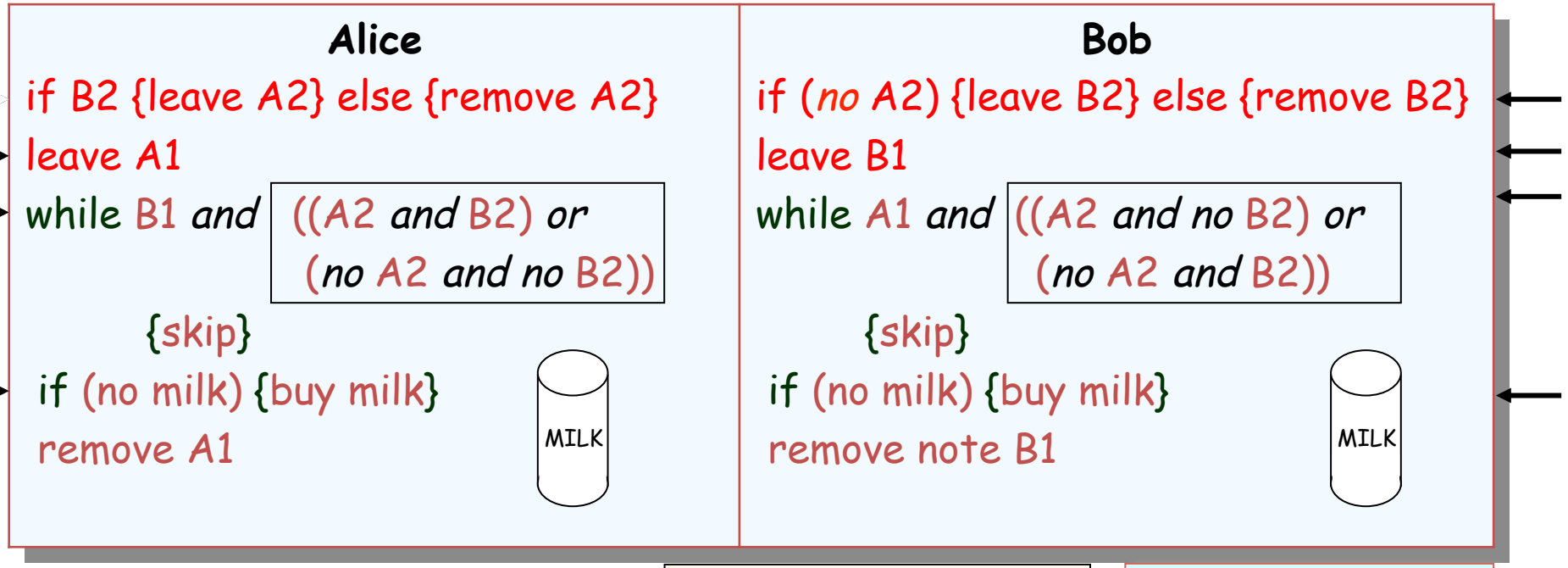
Bob

```
leave B1
if (no A2) {leave B2} else {remove B2}
while A1 and ((A2 and no B2) or
              (no A2 and B2))
      {skip}
if (no milk) {buy milk}
remove note B1
```

A variant of Solution 4

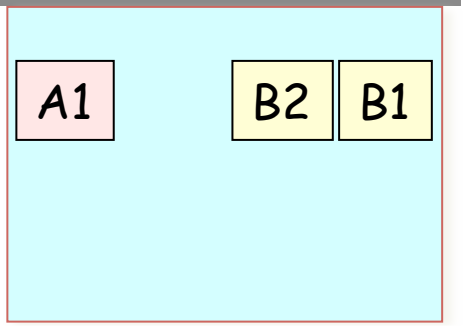
The order of the first two statements is replaced

- Is it correct ?



No, may end up with two milk.

X mutual exclusion
✓progress



Question

- Alice and Bob have a daughter C.
- Make it work for all the three of them! ☺

It is possible to solve a generalization of the too-much-milk problem, called the mutual exclusion problem, for any number of participants (not just two or three).

Question

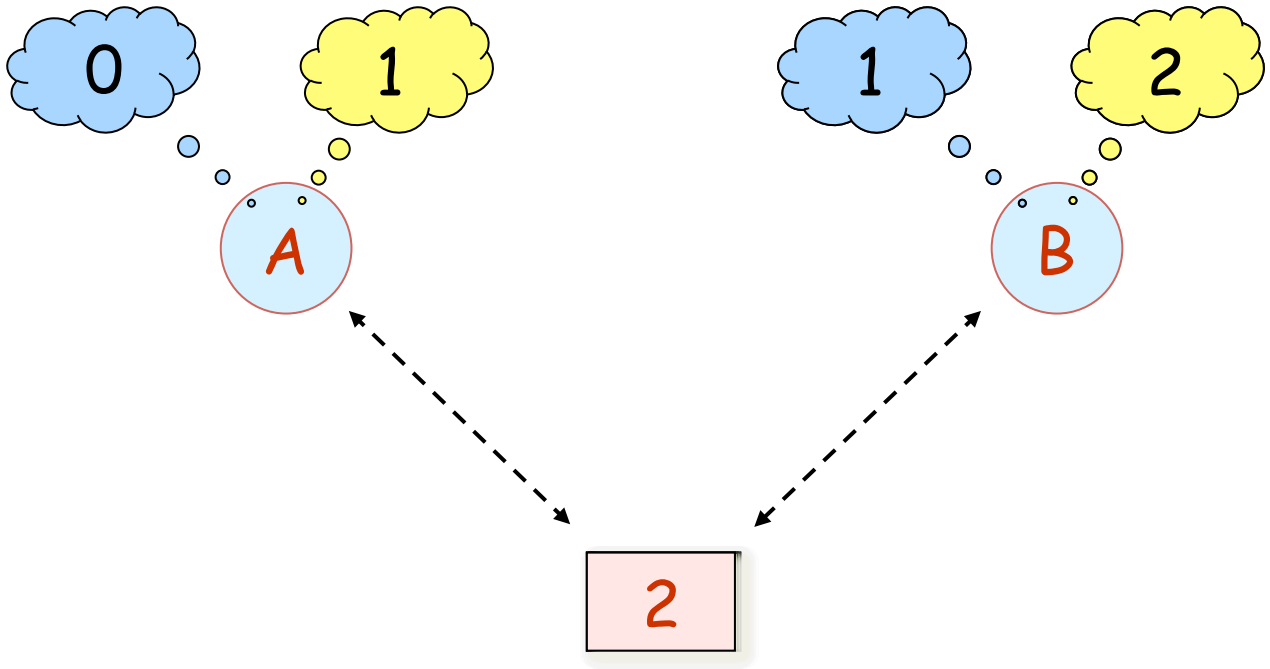
x is an atomic register, initially 0 | 0

Q: What are all the possible values for x after both processes terminate?

Process A	Process B
for i = 1 to 5 { x:=x+1 }	for i = 1 to 5 { x:=x+1 }

A: 2 through 10 inclusive.

The smallest value is 2



Question

x is an atomic register, initially 0 | 0

Q: What are all the possible values for x after the three processes terminate?

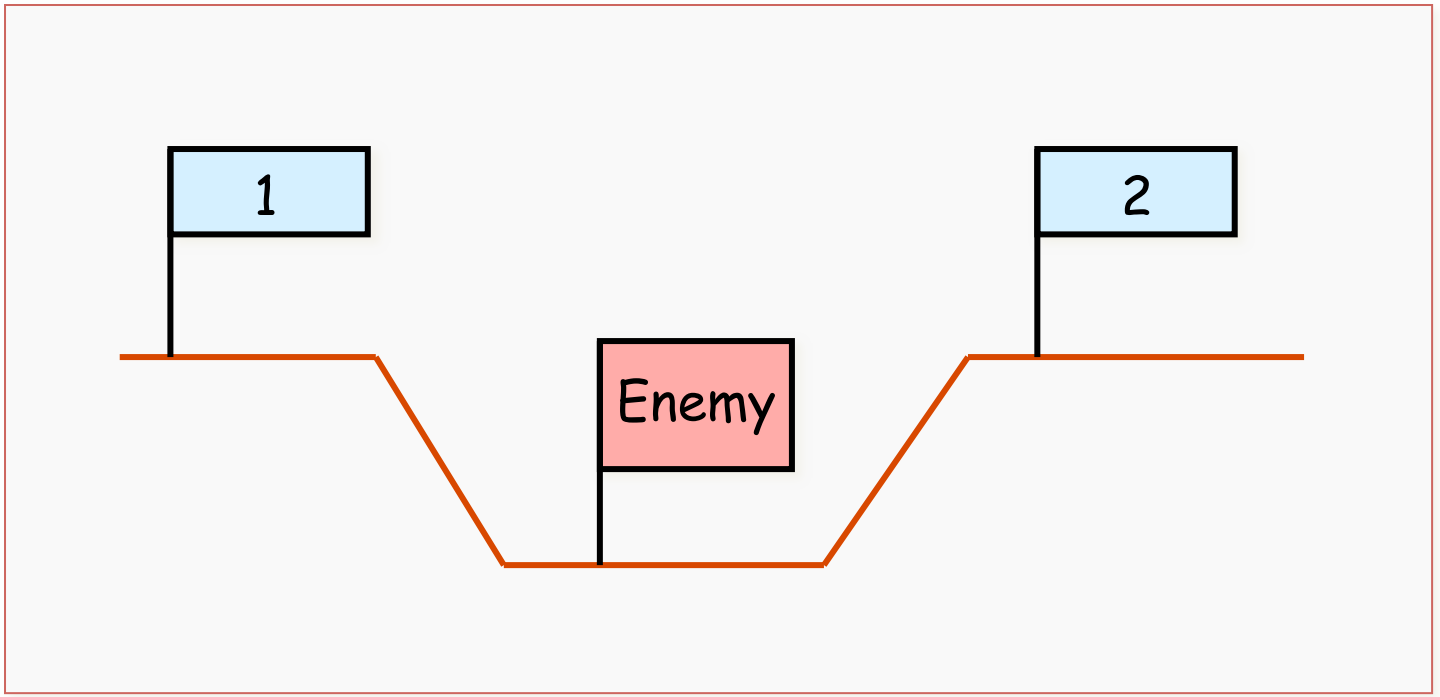
Process A	Process B	Process C
x:=x+1;	x:=x+1;	x:=10
x:=x+1	x:=x+1	

A: 2, 3, 4, 10, 11, 12, 13, 14.

Homework: find interlevings for each of the possible results!

The coordinated attack problem

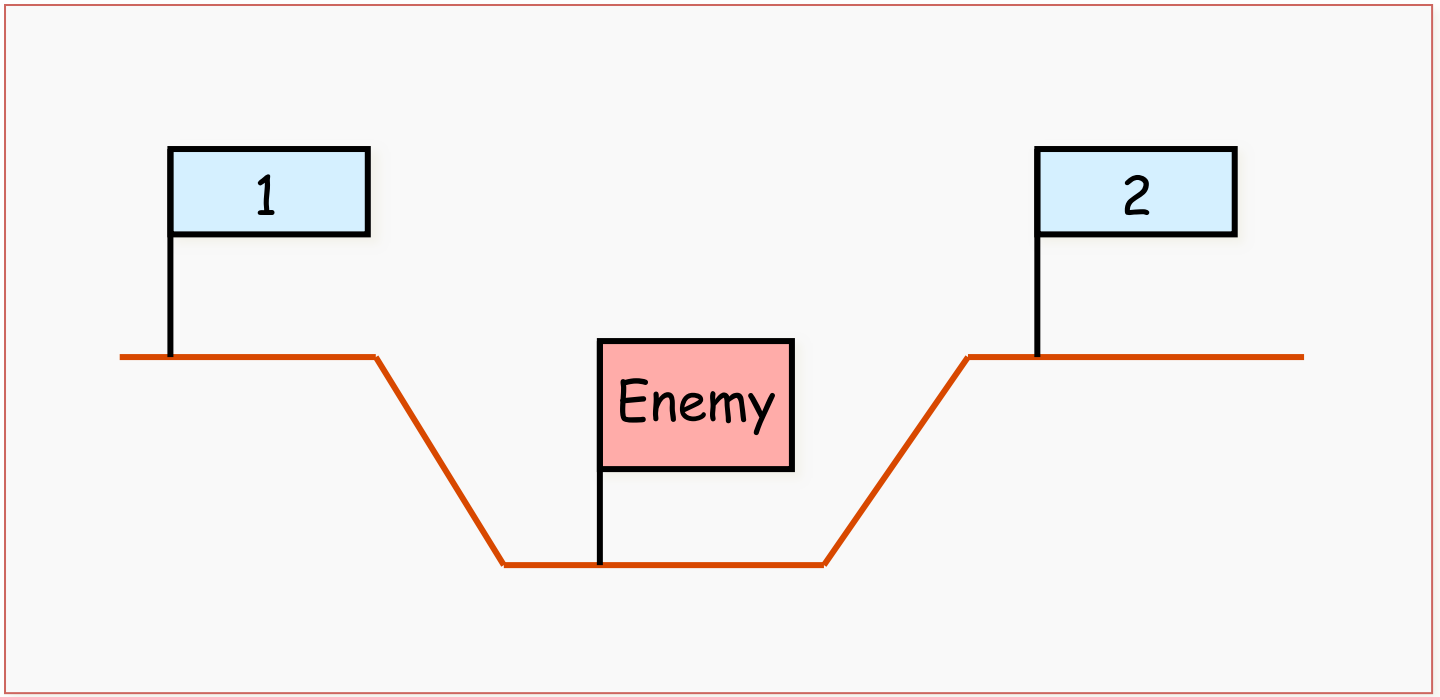
- Blue army wins if both blue camps attack simultaneously
- Design an algorithm to ensure that both blue camps attack simultaneously



The problem is due to Jim Gray (1977)

The coordinated attack problem

- Communication is done by sending messengers across the valley
- Messengers may be lost or captured by the enemy ☹



The problem is due to Jim Gray (1977)

The coordinated attack problem

- Theorem: There is no algorithm that solves the problem !
- Proof:
  - Assume to the contrary that such an algorithm exits.
  - Let P be an algorithm that solves with fewest # of
  - messages, when no message is lost.
  - P should work even when the last messenger is captured.
  - So P should work even if the last messenger is never sent.
  - But this is a new algorithm with one less message.
  - A contradiction. ■

→ The (asynchronous) model is too weak.

Synchronization

Contention

- *Too-much-milk*
- Mutual exclusion
- Concurrent Data Structures
- Multiple resources: dinning philosophers
- Readers & writer
- Group-exclusions
- …

Coordination

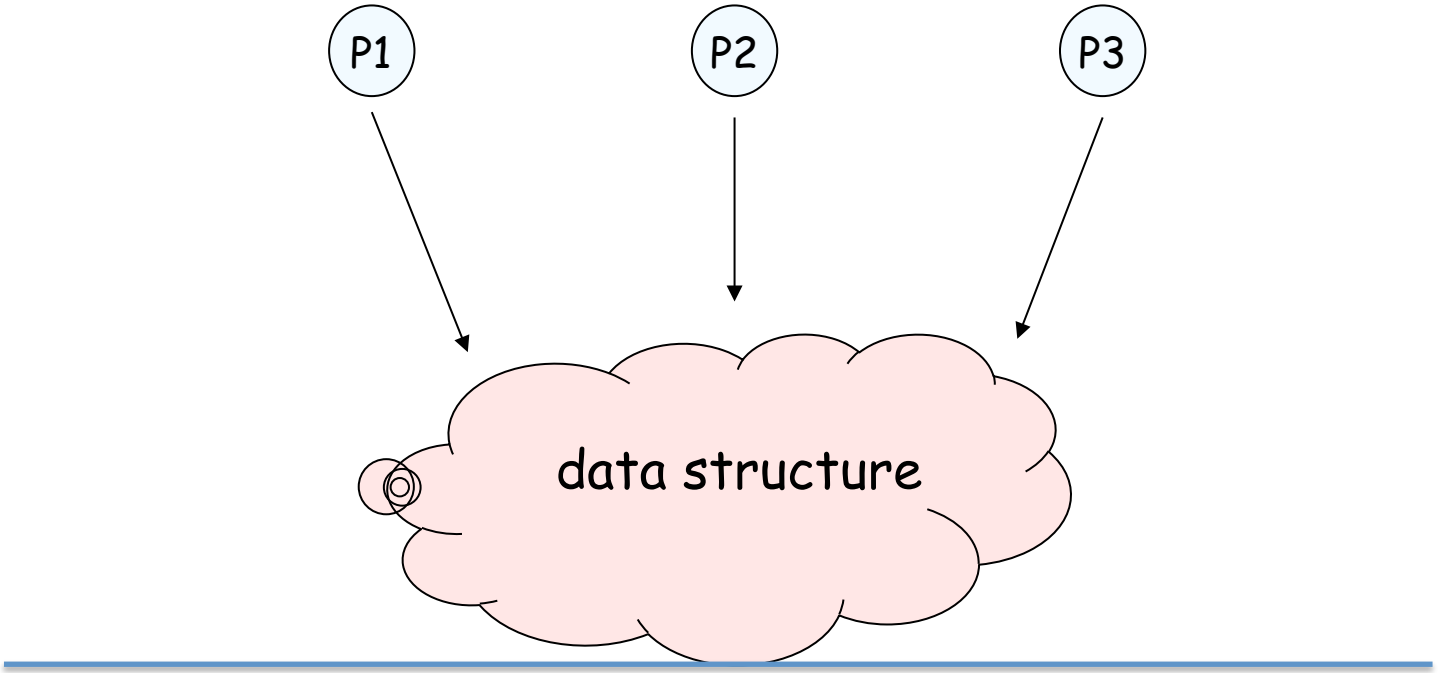
- The coordinated attack
- Barrier synchronization
- Concurrent Data Structures
- Producer-consumer
- Choice Coordination
- Consensus (data coordination)
- ….

The mutual exclusion problem

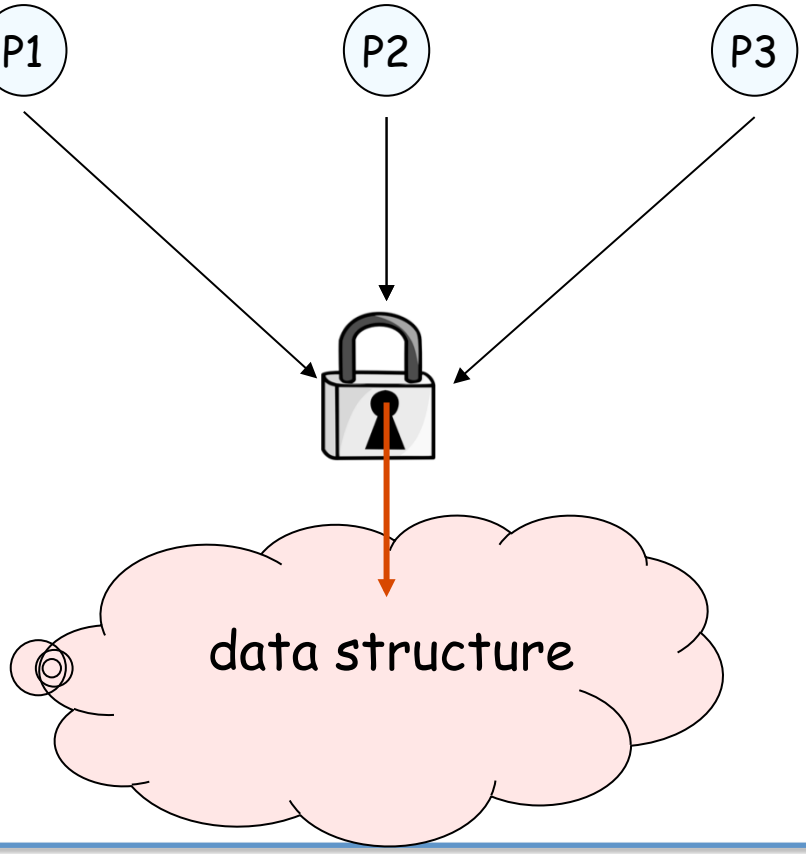
- The first problem we cover in detail.
- A generalization of the too-much-milk problem.

Blocking and Non-blocking Synchronization
Concurrent data structures

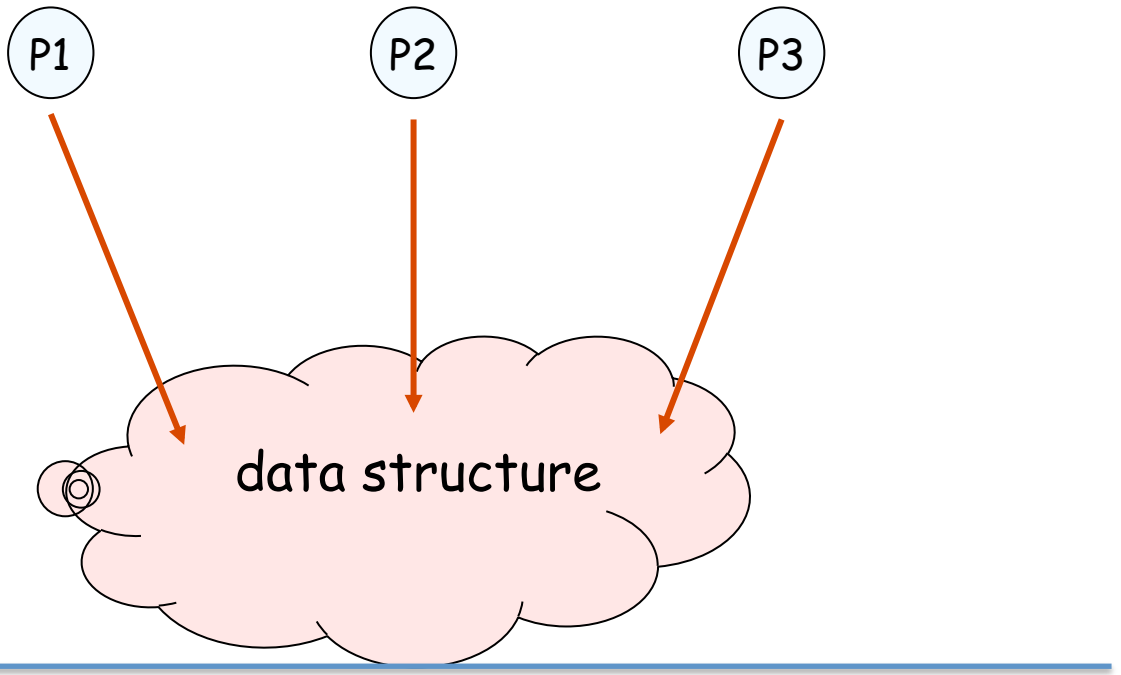
- counter, stack, queue, link list



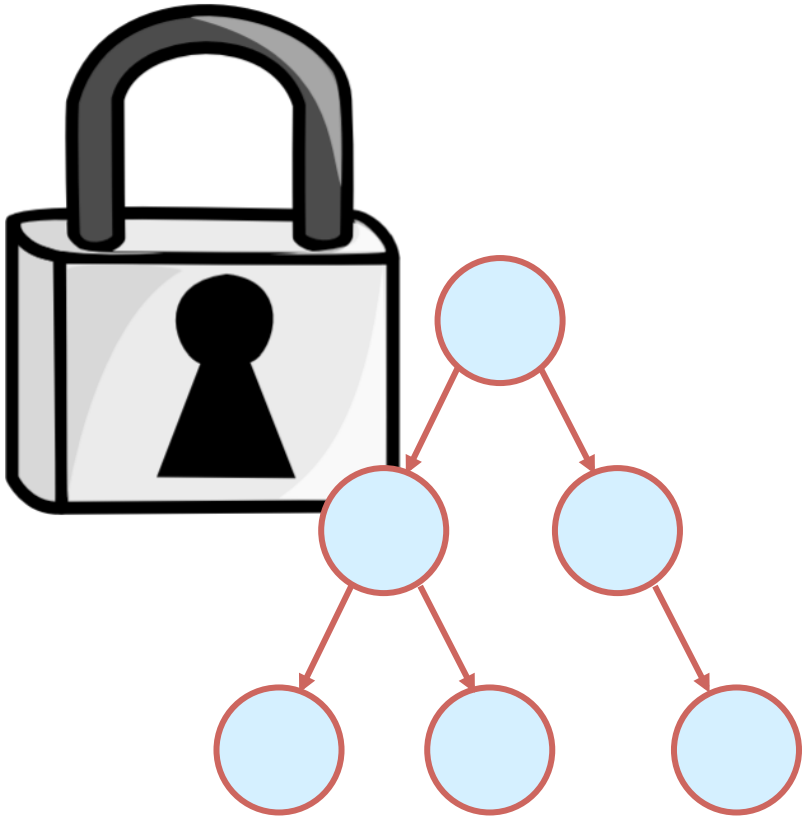
Blocking



Non-blocking

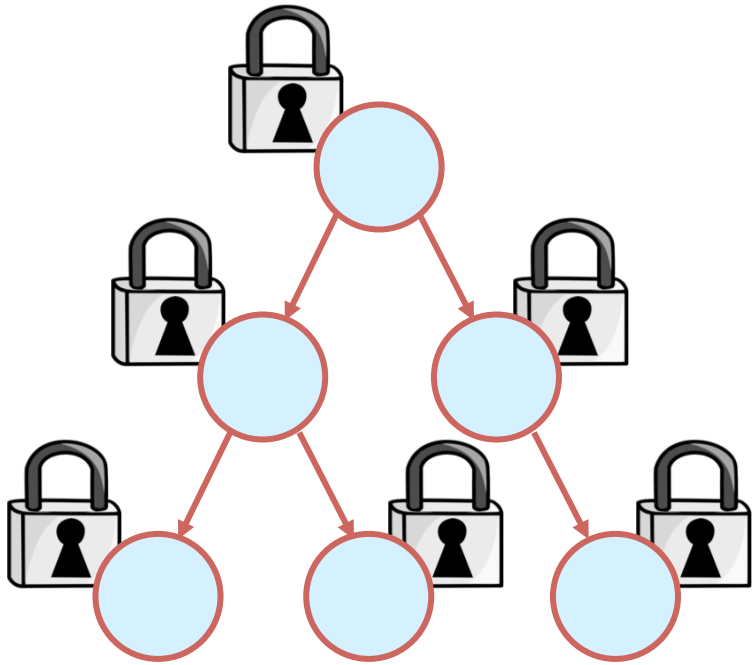


Coarse-Grained Locking



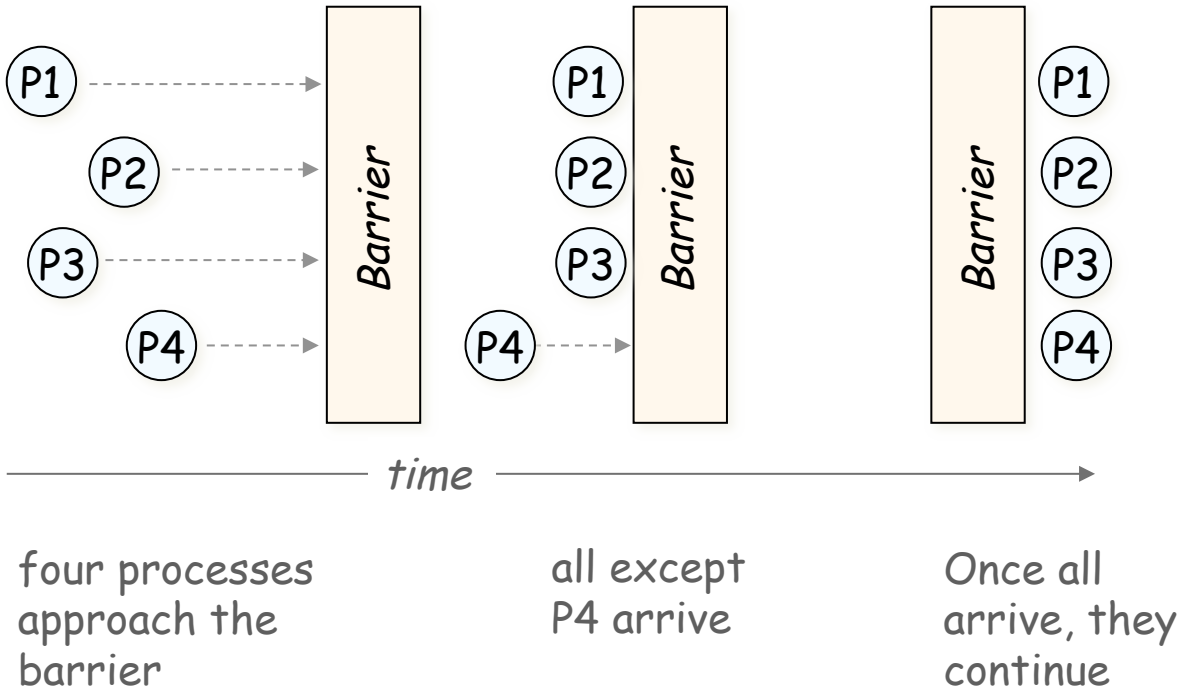
Easier to program, but not scalable.

Fine-Grained Locking

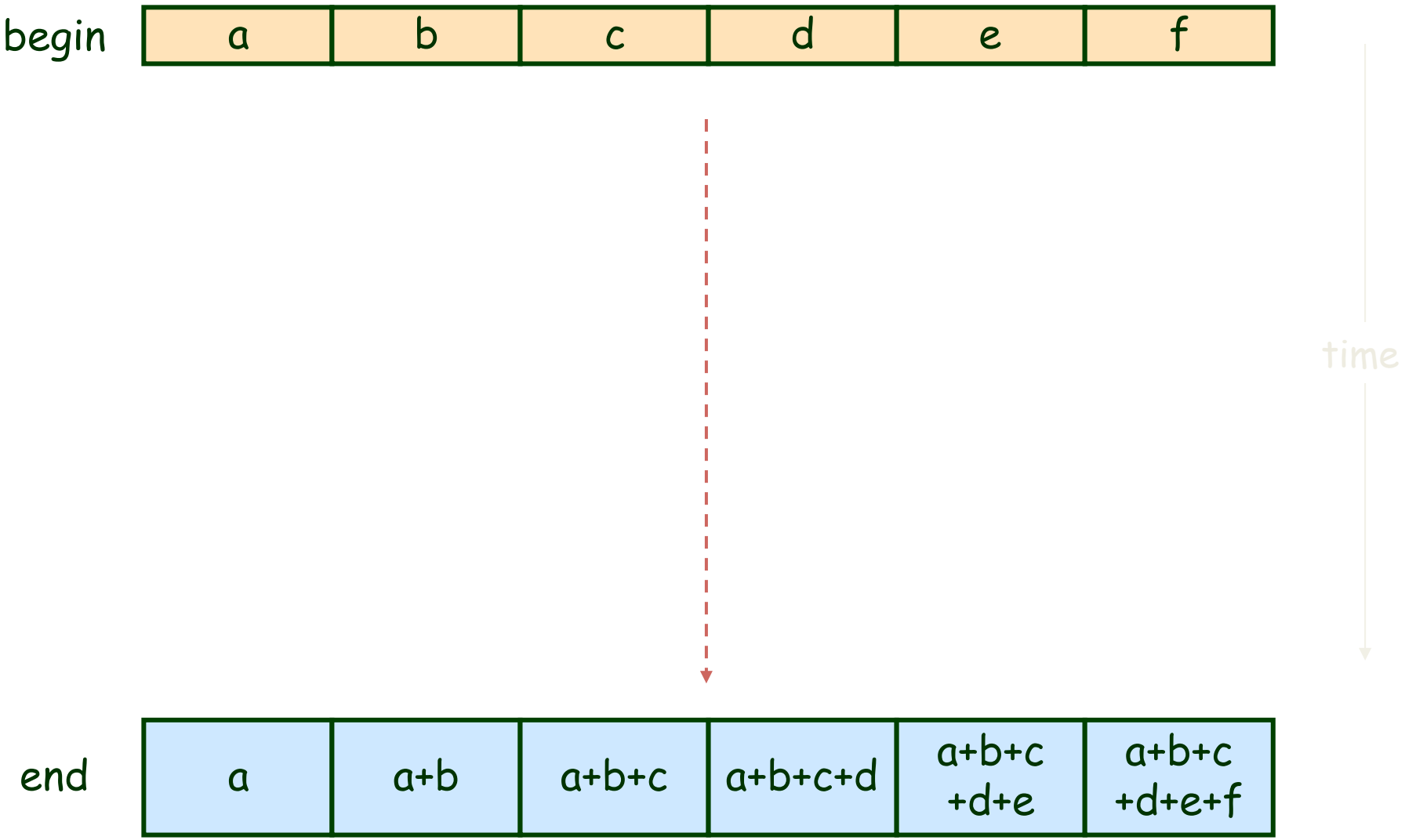


Efficient and scalable, but
too complicated (deadlocks, etc.).

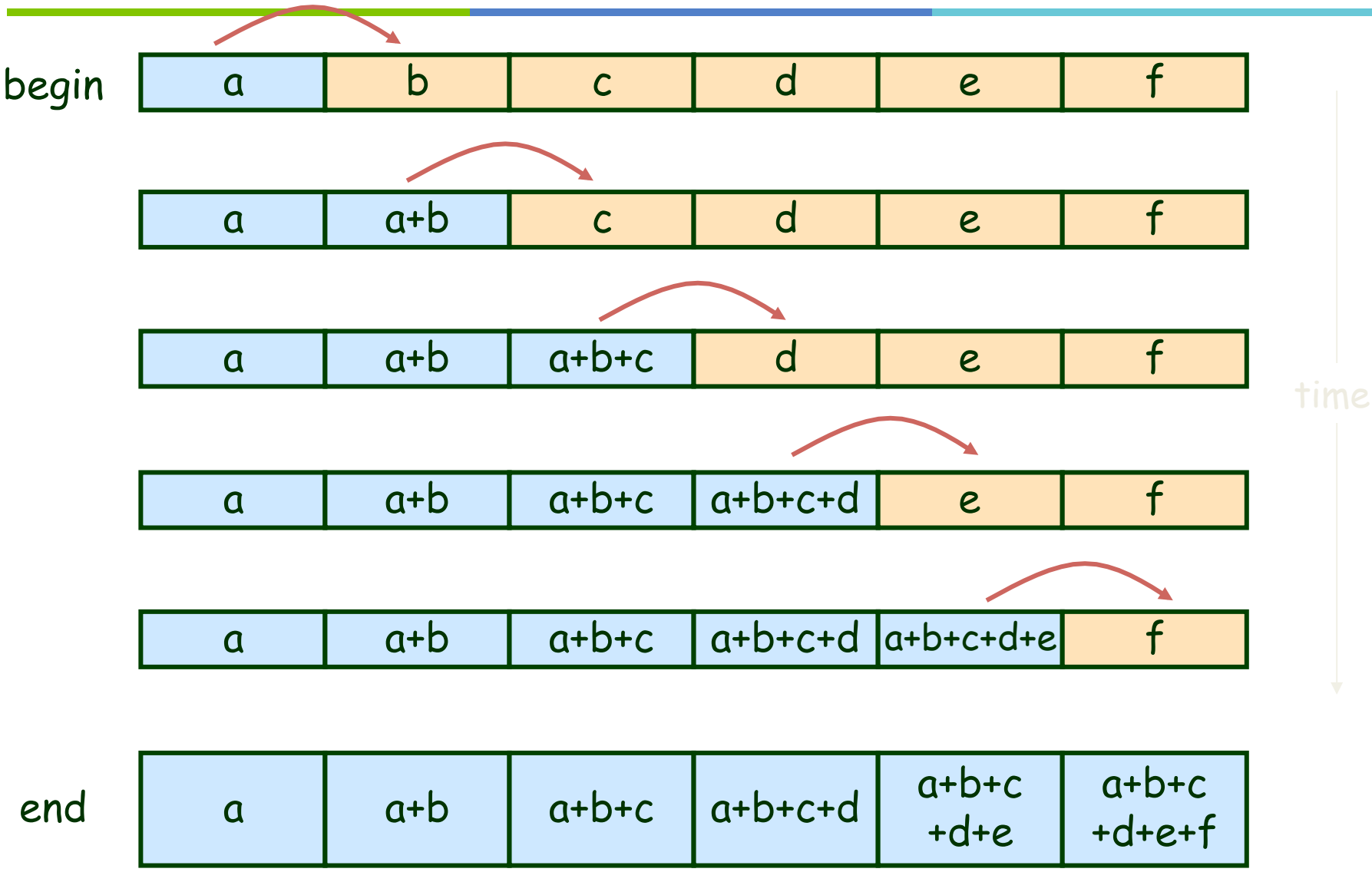
Barrier Synchronization



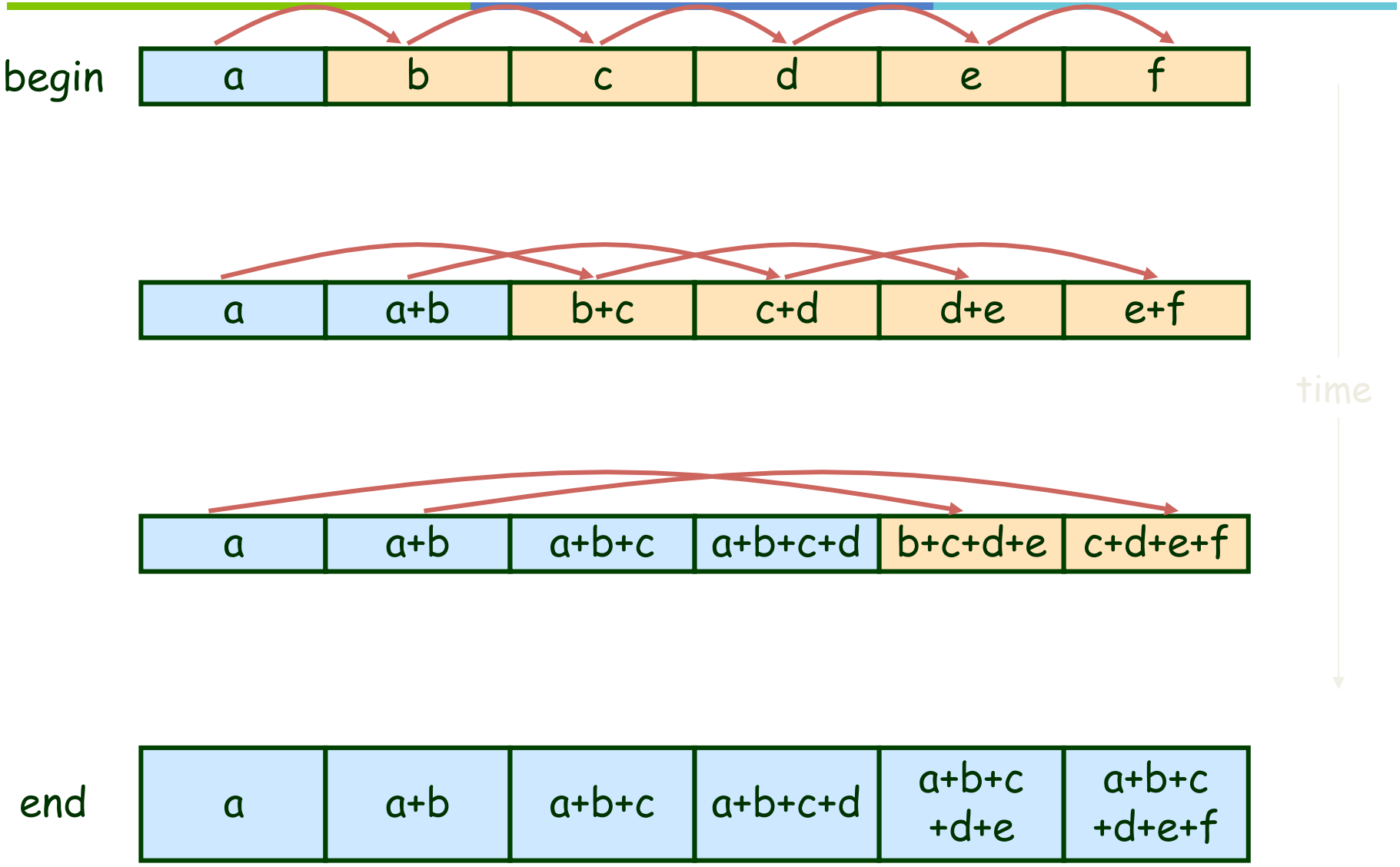
Example: Prefix Sum



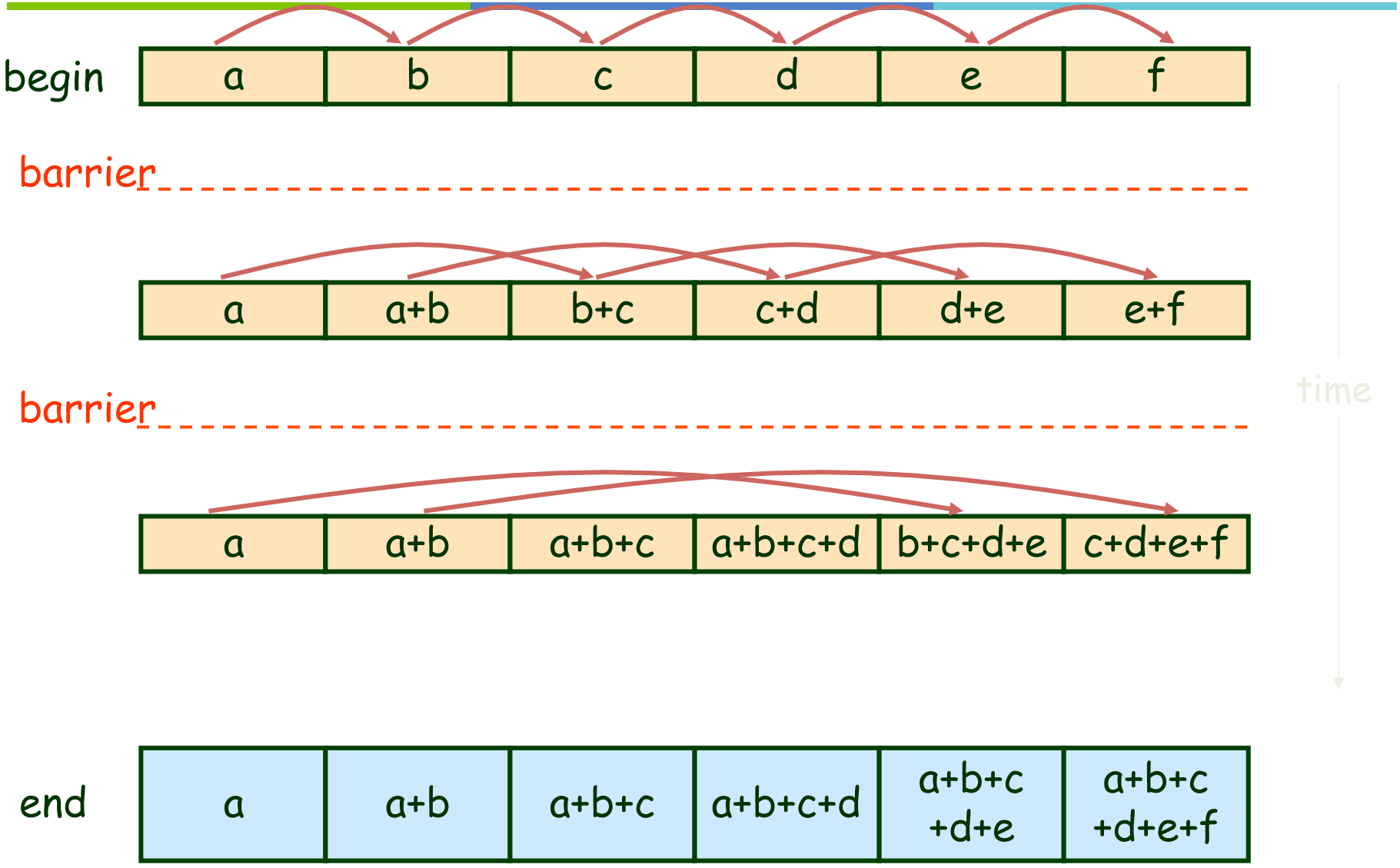
Example: Sequential Prefix Sum



Example: Parallel Prefix Sum



Example: Parallel Prefix Sum



departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Communication
models of computation

Concurrent programming ➔ Shared memory

- safe registers
- atomic registers (read/write)
- test-and-set
- swap
- compare-and-swap
- load-link / store-conditional
- read-modify-write
- semaphore
- monitor

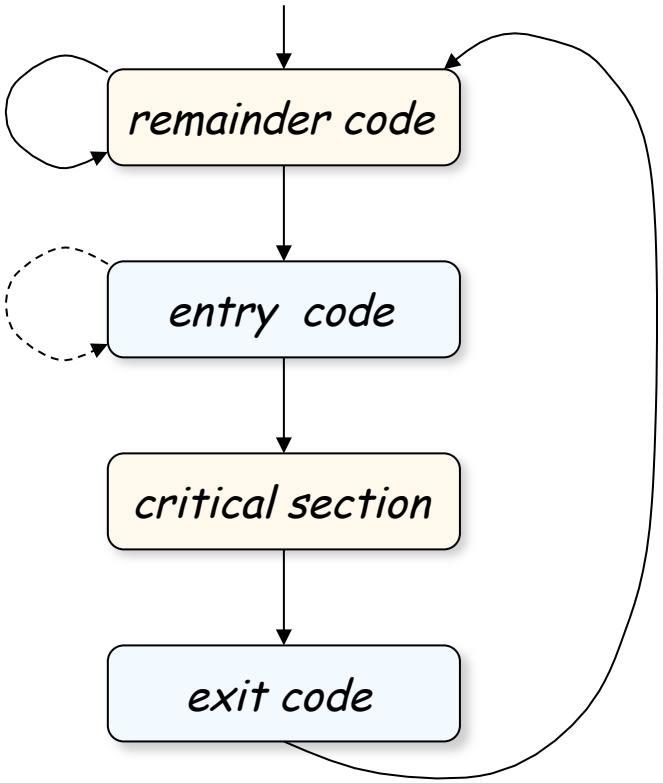
What is the relative power of these communication primitives?

Communication
models of computation

- Message passing
  - send/receive
  - multi-cast
  - broadcast
  - network topology

Distributed message passing systems are not covered in this course

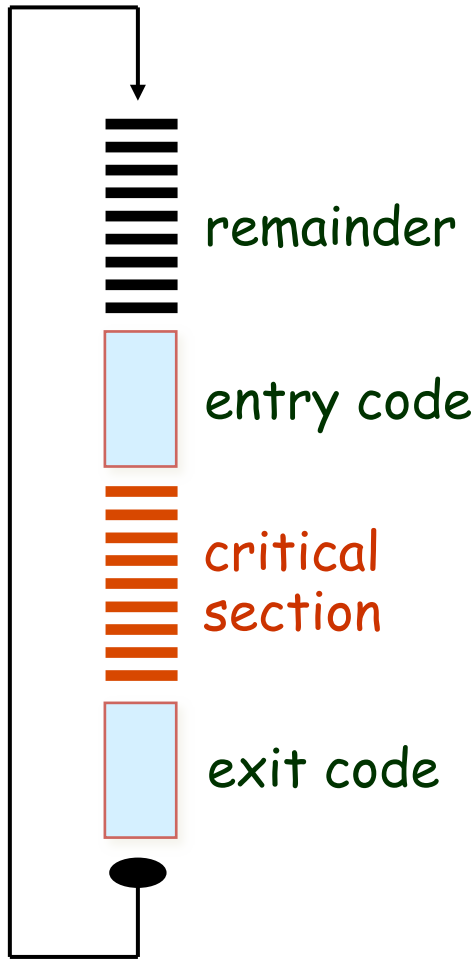
The mutual exclusion problem



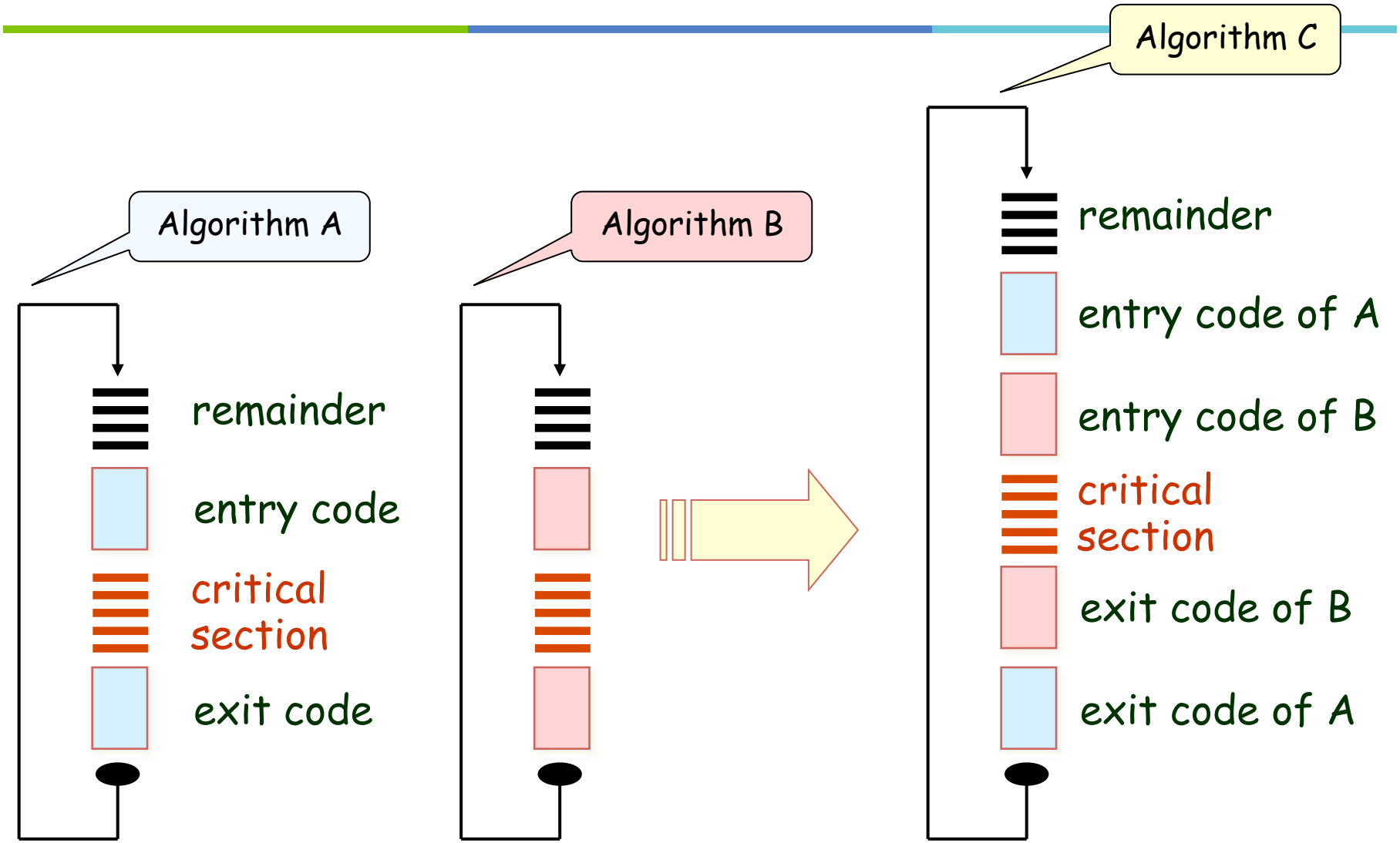
The problem is to design the entry and exit code in a way that guarantees that the mutual exclusion and deadlock-freedom properties are satisfied.

The mutual exclusion problem

- **Mutual Exclusion:** No two processes are in their critical section at the same time.
- **Deadlock-freedom:** If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- **Starvation-freedom:** If a process is trying to enter its critical section, then this process must eventually enter its critical section.

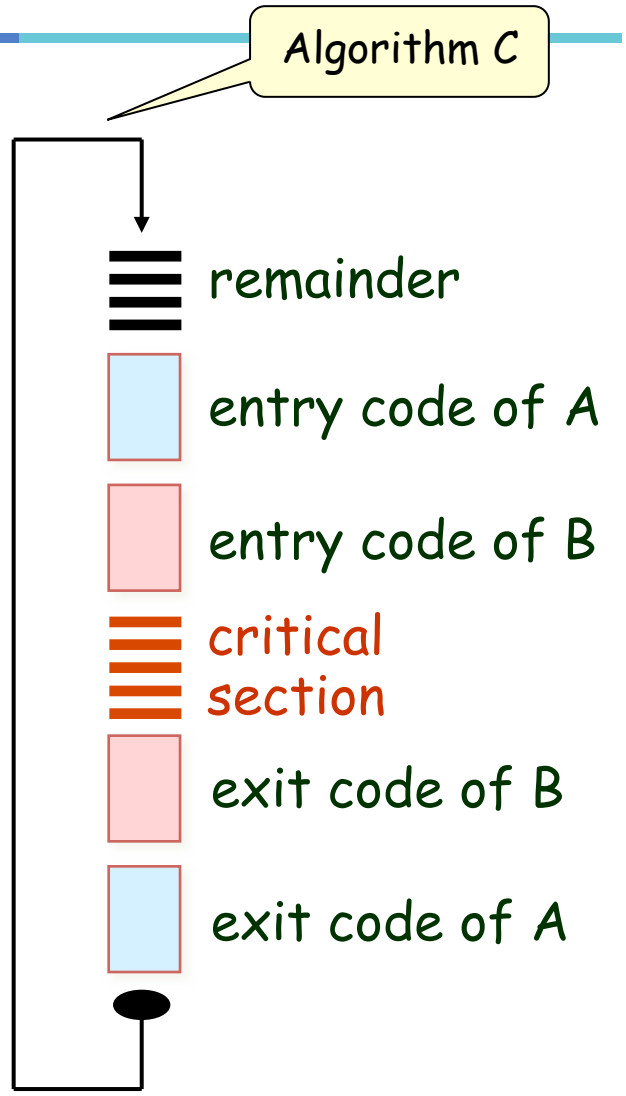


Question: true or false ?



Question: true or false ?

- A and B are deadlock-free \ C is deadlock-free.
- A and B are starvation-free \ C is starvation-free.
- A or B satisfies mutual exclusion \ C satisfies mutual exclusion.
- A is deadlock-free and B is starvation-free \ C is starvation-free.
- A is starvation-free and B is deadlock-free \ C is starvation-free.



Properties & complexity

- Time complexity
  - Fast
  - Adaptive
- Fairness
  - FIFO, …
- Fault-tolerance
- Local-spinning
- Space complexity
- Communication primitives

Complexity Measures

- Counting steps
- Process step complexity
- Process time complexity
- Contention-free time complexity
- Counting time units
- System response time
- Process response time
- Counting communications
- Distributed shared memory
- Coherent caching
- Space complexity

History of the mutex problem

- 1965
  - Defined by Dijkstra
  - First solution by Dekker
  - General solution by Dijkstra
- Fischer, Knuth, Lynch, Rabin, Rivest, ...
- 1974 Lamport's bakery algorithm
- 1981 Peterson's solution
- There are hundreds of published solutions
- Lots of related problems

E. W. Dijkstra
www.cs.utexas.edu/users/EWD/
(Photo by *Hamilton Richards* 2002)

The END

Detected errors: Misuses of the POSIX pthreads API

- unlocking an invalid mutex
- unlocking a not-locked mutex
- unlocking a mutex held by a different thread
- destroying an invalid or a locked mutex
- recursively locking a non-recursive mutex
- deallocation of memory that contains a locked mutex
- passing mutex arguments to functions expecting reader-writer lock arguments, and vice versa
- when a POSIX pthread function fails with an error code that must be handled
- when a thread exits whilst still holding locked locks
- calling pthread_cond_wait with a not-locked mutex, an invalid mutex, or one locked by a different thread
- inconsistent bindings between condition variables and their associated mutexes
- invalid or duplicate initialisation of a pthread barrier
- initialisation of a pthread barrier on which threads are still waiting
- destruction of a pthread barrier object which was never initialised, or on which threads are still waiting
- waiting on an uninitialised pthread barrier

for all of the pthreads functions that Helgrind intercepts, an error is reported, along with a stack trace, if the system threading library routine returns an error code, even if Helgrind itself detected no error

```
#include <pthread.h>
int counter = 0; // shared variable

void * func (void * params) { counter++;  return NULL; }

int main ()
{
      pthread_t thread1, thread2;
      pthread_create (&thread1, NULL, func, NULL);
      pthread_create (&thread2, NULL, func, NULL);
      pthread_join (thread1, NULL);       pthread_join(thread2, NULL);
      return 0;
}
```

- valgrind --tool=helgrind myapplication