Whatever can go wrong
will go wrong.

attributed to Edward A. Murphy

Murphy was an optimist.

authors of lock-free programs

LOCK FREE RUNTIME SYSTEM

Literature

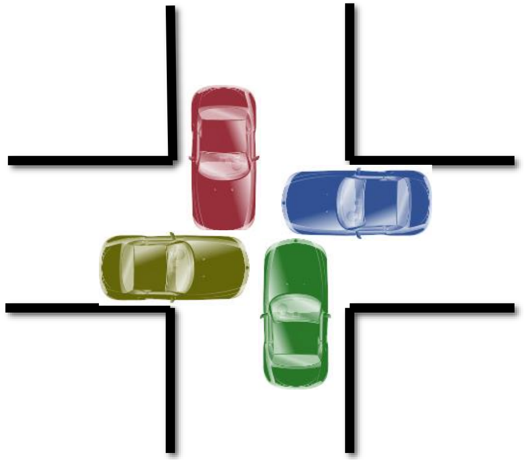
Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

Florian Negele. *Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System.* ETH-Zürich, 2014. http://dx.doi.org/10.3929/ethz-a-010335528

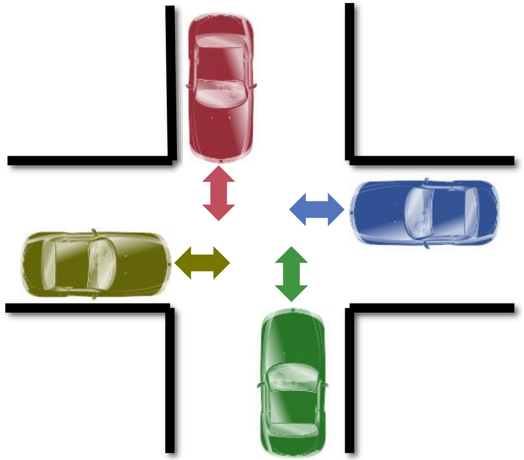
A substantial part of the following material is based on Florian Negele's Thesis.

Florian Negele, Felix Friedrich, Suwon Oh and Bernhard Egger, *On the Design and Implementation of an Efficient Lock-Free Scheduler,* 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2015.

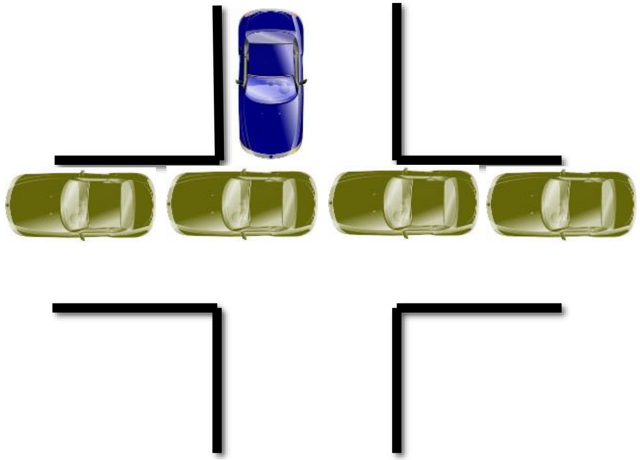
Problems with Locks



Deadlock



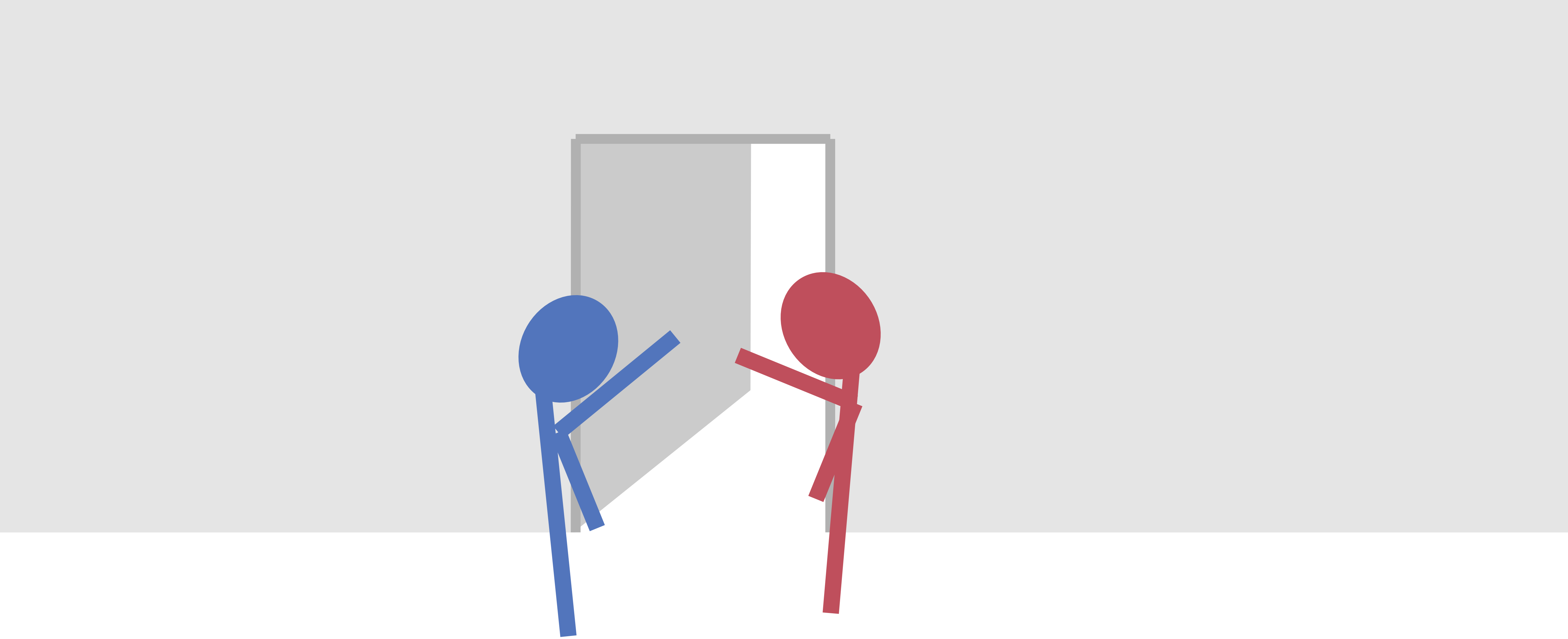
Livelock



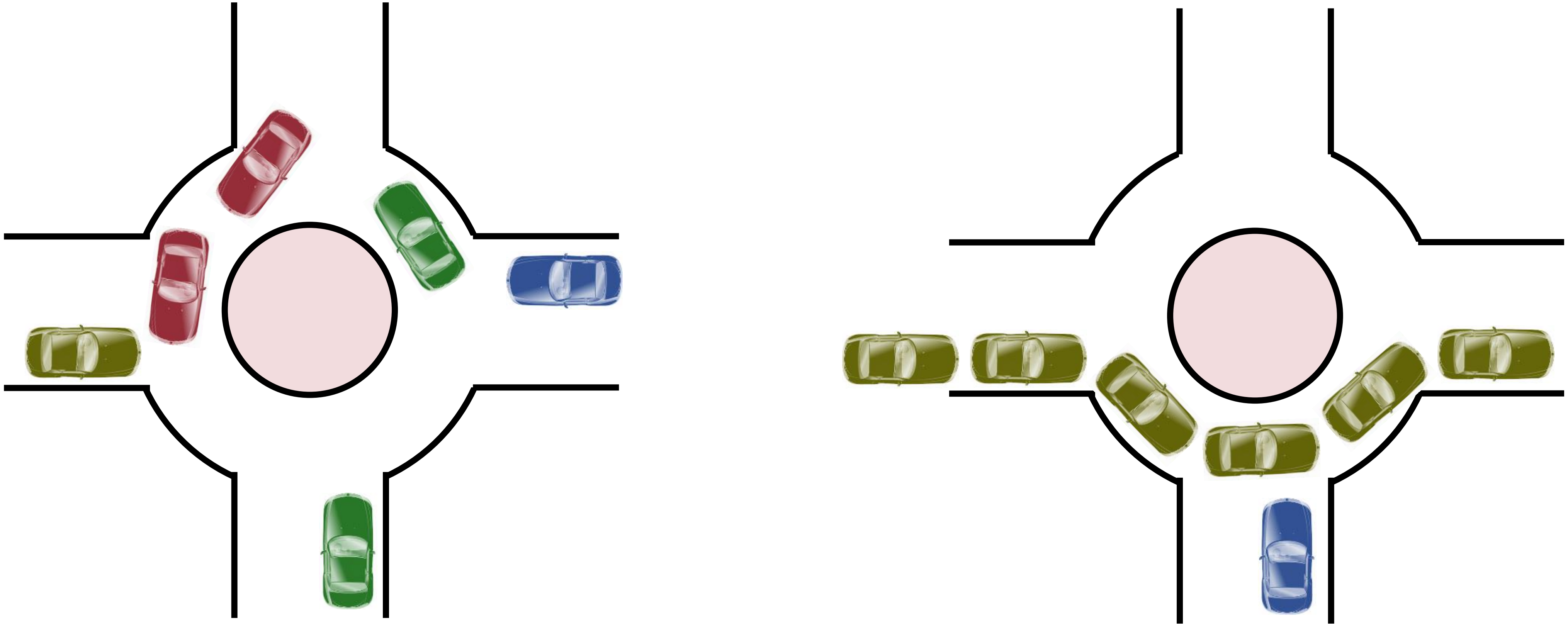
Starvation

Parallelism? Progress Guarantees? Reentrancy? Granularity? Fault Tolerance?

Politelock

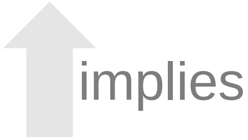


Lock-Free



Definitions

Lock-freedom: at least one algorithm makes progress even if other algorithms run concurrently, fail or get suspended.
Implies system-wide progress but not freedom from starvation.



Wait-freedom: all algorithms eventually make progress.
Implies freedom from starvation.

Progress Conditions

	Blocking	Non-Blocking
Someone make progress	Deadlock-free	**Lock-free**
Everyone makes progress	Starvation-free	Wait-free

Goals

Lock Freedom

- Progress Guarantees
- Reentrant Algorithms

Portability

- Hardware Independence
- Simplicity, Maintenance

Guiding principles

1. Keep things **simple**
2. Exclusively employ **non-blocking** algorithms in the system

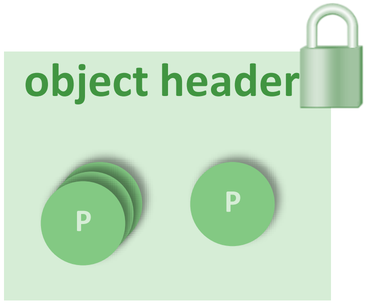
→ Use **implicit cooperative multitasking**

→ no virtual memory

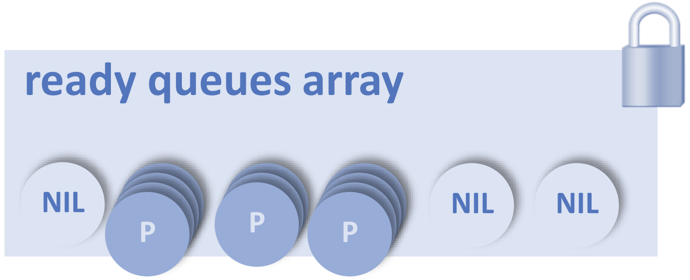
→ limits in optimization

Where are the Locks in the Kernel?

Scheduling Queues / Heaps



Memory Management



CAS (again)

- Compare **old** with data at memory location
- If and only if data at memory equals **old** overwrite data with **new**
- Return previous memory value

int CAS (memref a, int old, int new)

atomic

```
previous = mem[a];
if (old == previous)
    Mem[a] = new;
return previous;
```

CAS is implemented wait-free(!) by hardware.

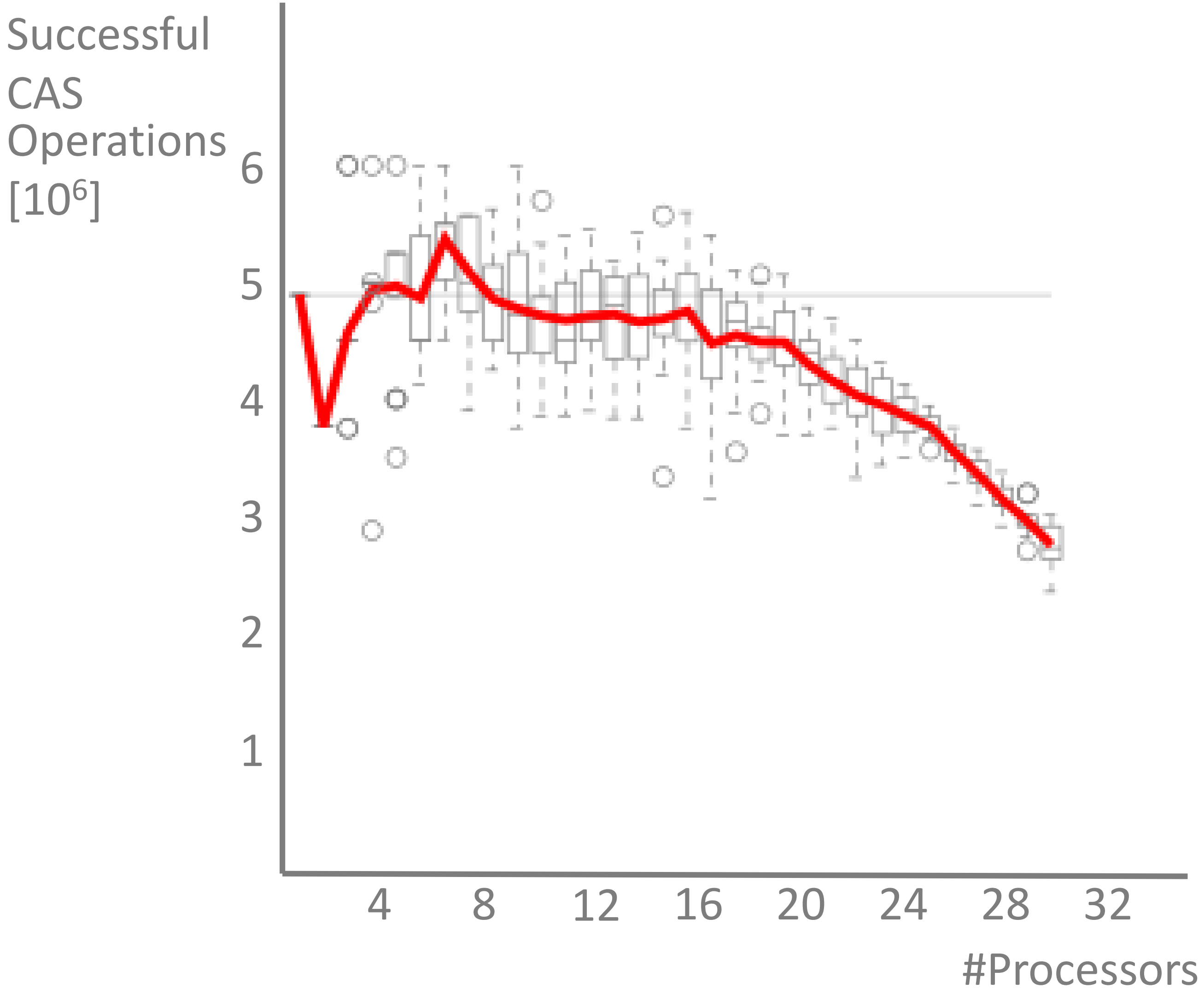
Simple Example: Non-blocking counter

```
PROCEDURE Increment(VAR counter: LONGINT): LONGINT;
VAR previous, value: LONGINT;
BEGIN
   REPEAT
     previous := CAS(counter,0,0);
     value := CAS(counter, previous, previous + 1);
   UNTIL value = previous;
   return previous;
END Increment;
```

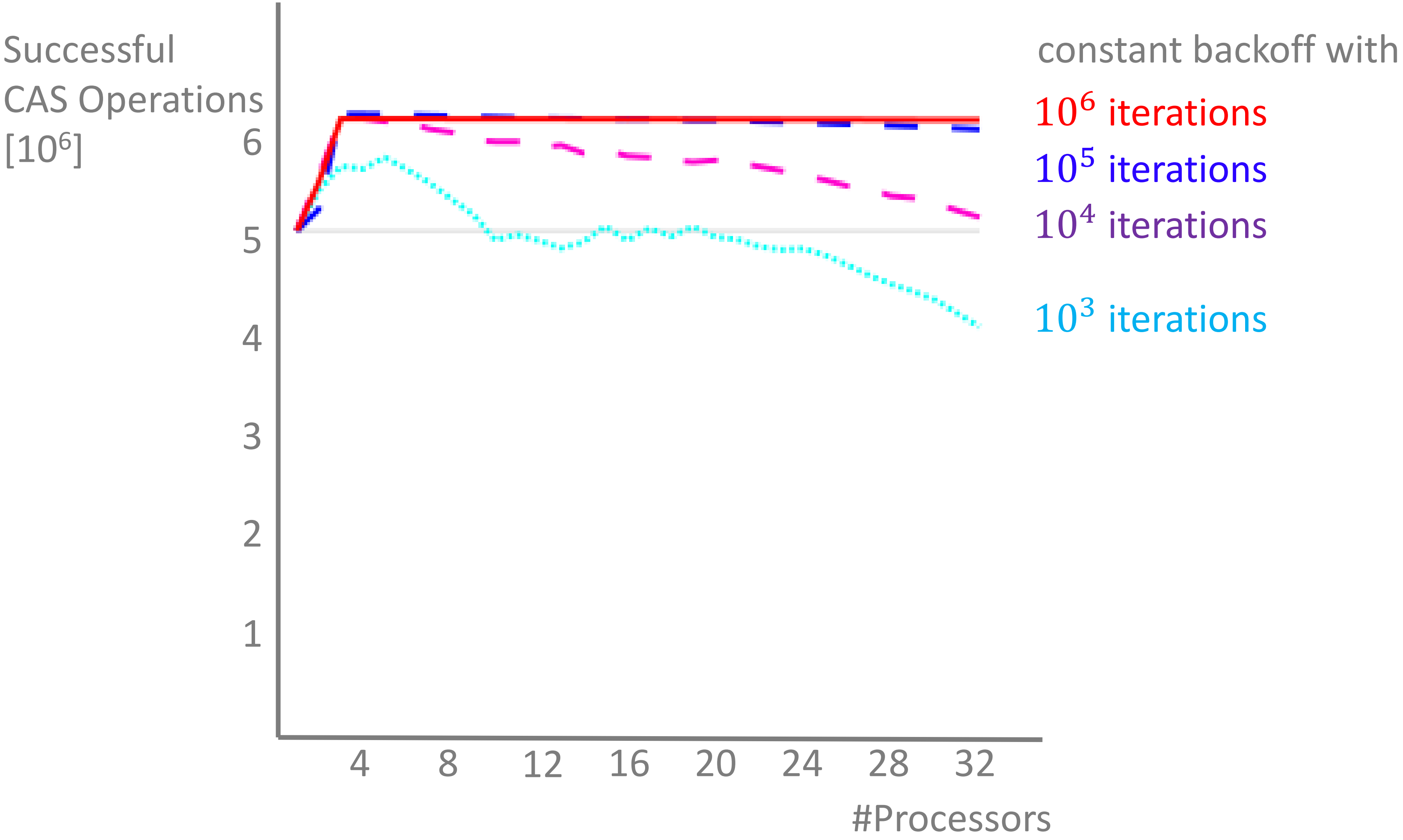
Lock-Free Programming

Performance of CAS

- on the H/W level, CAS triggers a memory barrier
- performance suffers with increasing number of contenders to the same variable



CAS with backoff



Memory Model for Lockfree Active Oberon

Only **two rules**

1. Data shared between two or more activities at the same time has to be protected using exclusive blocks unless the data is read or modified using the compare-and-swap operation
2. Changes to shared data visible to other activities after leaving an exclusive block or executing a compare-and-swap operation. Implementations are free to reorder all other memory accesses as long as their effect equals a sequential execution within a single activity.

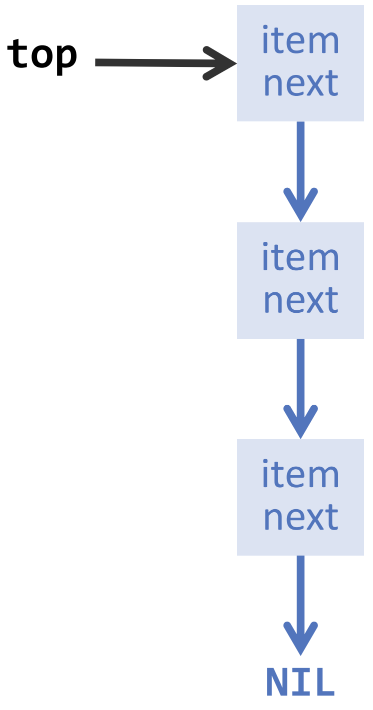
Inbuilt CAS

- CAS instruction as statement of the language
  `PROCEDURE CAS(variable, old, new: BaseType): BaseType`
  - Operation executed atomically, result visible instantaneously to other processes
  - CAS(variable, x, x) constitutes an atomic read
- Compilers required to implement CAS as a synchronisation barrier
  - Portability, even for non-blocking algorithms
  - Consistent view on shared data, even for systems that represent words using bytes

Stack

```
Node = POINTER TO RECORD
  item: Object;
  next: Node;
END;

Stack = OBJECT
VAR top: Node;
  PROCEDURE Pop(VAR head: Node): BOOLEAN;
  PROCEDURE Push(head: Node);
END;
```



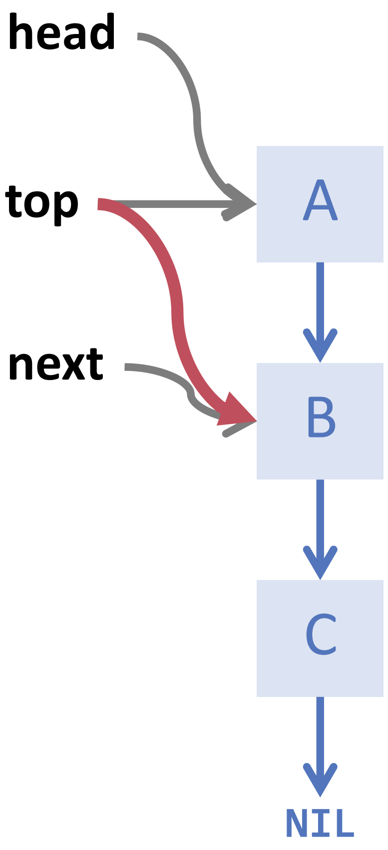
Stack -- Blocking

```
PROCEDURE Push(node: Node): BOOLEAN;
BEGIN{EXCLUSIVE}
   node.next := top;
   top := node;
END Push;

PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN{EXCLUSIVE}
   head := top;
   IF head = NIL THEN
      RETURN FALSE
   ELSE
      top := head.next;
      RETURN TRUE;
   END;
END Pop;
```

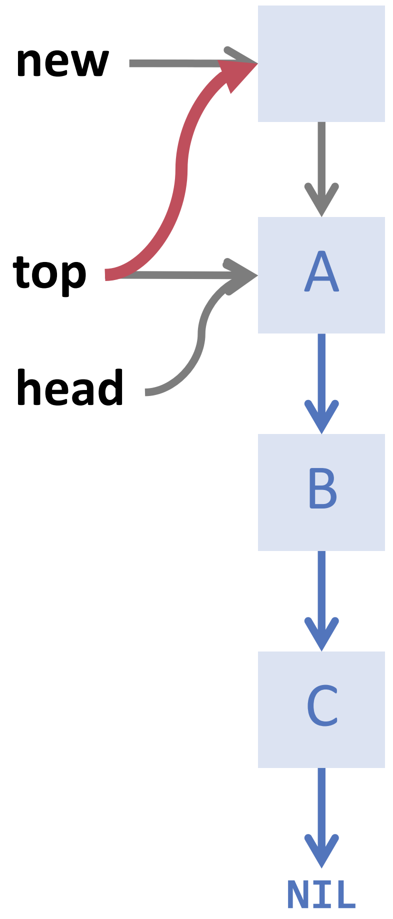
Stack -- Lockfree

```
PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN
  LOOP
    head := CAS(top, NIL, NIL);
    IF head = NIL THEN
      RETURN FALSE
    END;
    next := CAS(head.next, NIL, NIL);
    IF CAS(top, head, next) = head THEN
      RETURN TRUE
    END;
    CPU.Backoff
  END;
END Pop;
```



Stack -- Lockfree

```
PROCEDURE Push(new: Node);
BEGIN
   LOOP
      head := CAS(top, NIL, NIL);
      CAS(new.next, new.next, head);
      IF CAS(top, head, new) = head THEN
         EXIT
      END;
      CPU.Backoff;
   END;
END Push;
```

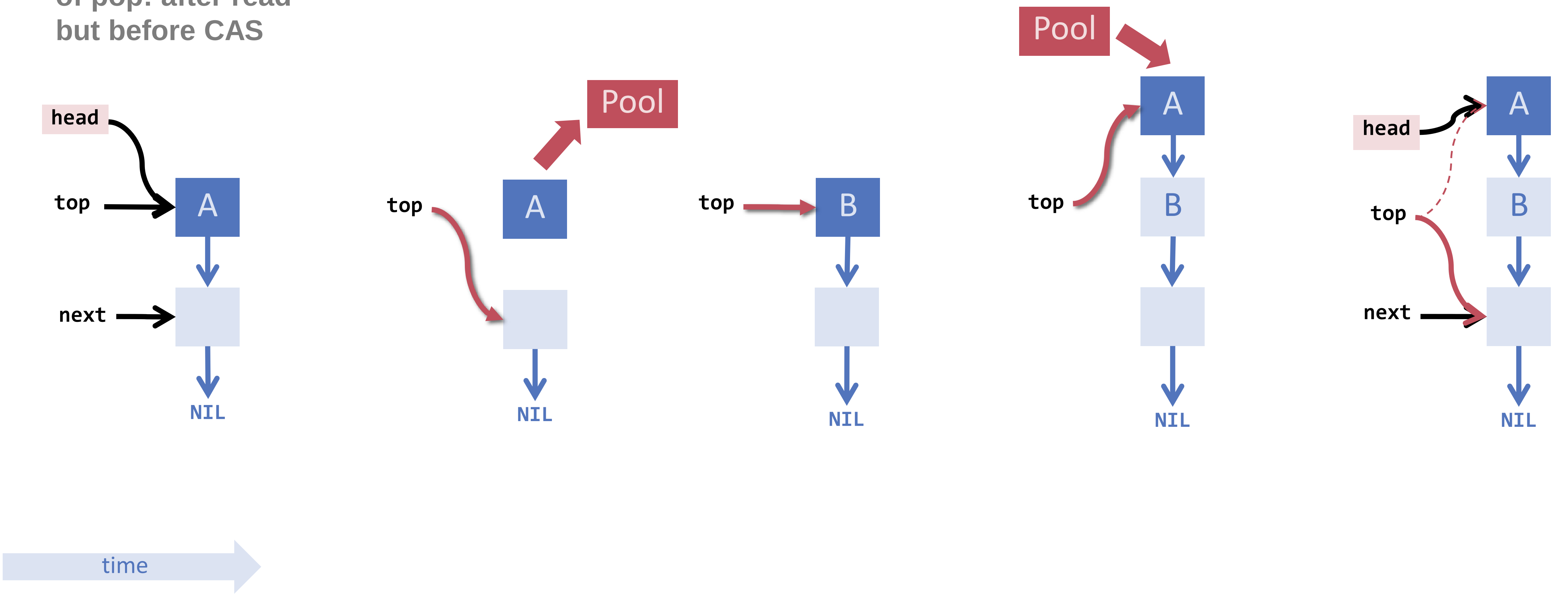


Node Reuse

Assume we do not want to allocate a new node for each Push and maintain a Node-pool instead. Does this work?

NO !

ABA Problem



Thread X
in the middle
of pop: after read
but before CAS

Thread Y
pops A

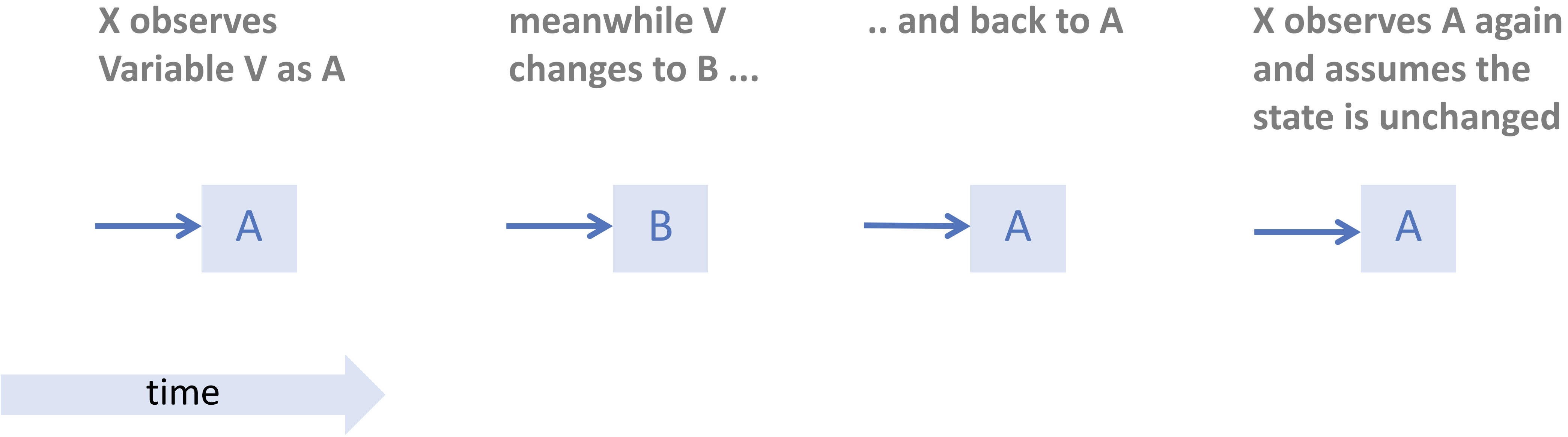
Thread Z
pushes B

Thread Z'
pushes A

Thread X
completes pop

The **ABA**-Problem

"The ABA problem ... occurs when one activity fails to recognise that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overal state has not been changed."



X observes Variable V as A

meanwhile V changes to B ...

.. and back to A

X observes A again and assumes the state is unchanged

How to solve the ABA problem?

- DCAS (double compare and swap)
  - not available on most platforms
- Hardware transactional memory
  - not available on most platforms
- Garbage Collection
  - relies on the existence of a GC
  - impossible to use in the inner of a runtime kernel
  - can you implement a lock-free garbage collector relying on garbage collection?
- **Pointer Tagging**
  - does not cure the problem, rather delay it
  - can be practical
- **Hazard Pointers**

Pointer Tagging

ABA problem usually occurs with CAS on *pointers*

Aligned addresses (values of pointers) make some bits available for *pointer tagging.*

Example: pointer aligned modulo 32 → 5 bits available for tagging

MSB	...	X	X	X	X	X	X	X	X	0	0	0	0	0

Each time a pointer is stored in a data structure, the tag is increased by one. Access to a data structure via address x – x mod 32

This makes the ABA problem very much less probable because now 32 versions of each pointer exist.

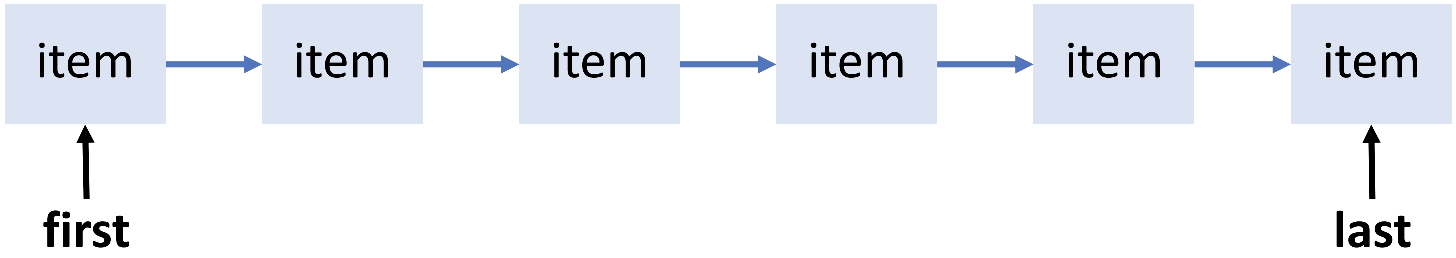
Hazard Pointers

The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Idea to solve:

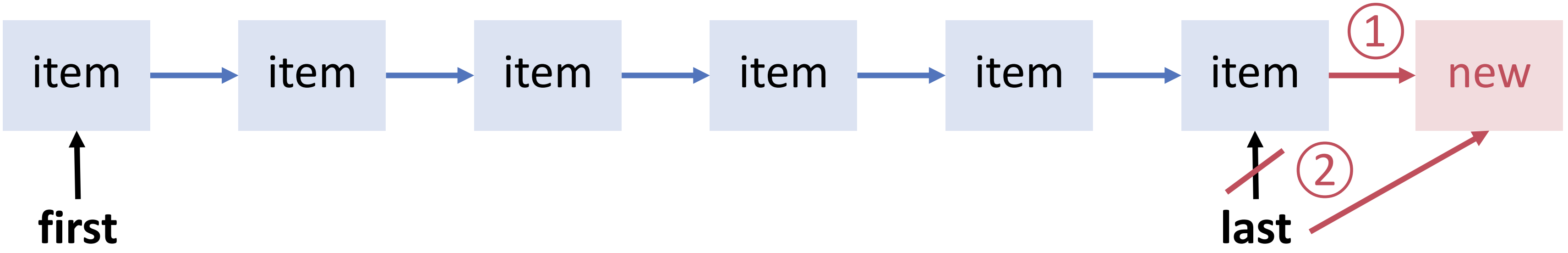
- Before X reads P, it marks it **hazarduous** by entering it in a thread-dedicated slot of the n (n= number threads) slots of an array associated with the data structure (e.g. the stack)
- When finished (after the CAS), process X removes P from the array
- Before a process Y tries to reuse P, it checks all entries of the hazard array

Unbounded Queue (FIFO)



Enqueue

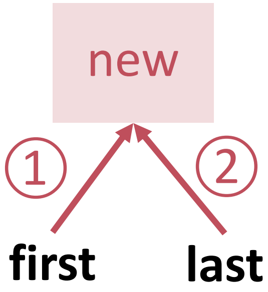
case last != NIL



item item item item item item ① new

first ② last

case last = NIL



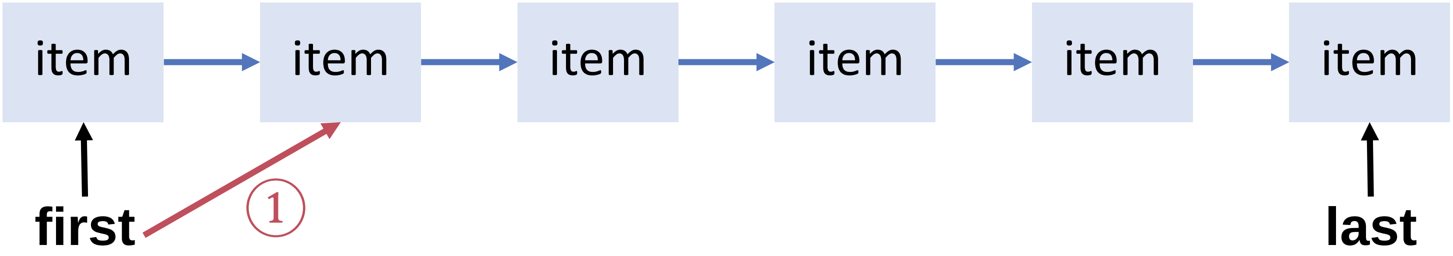
new

① ②

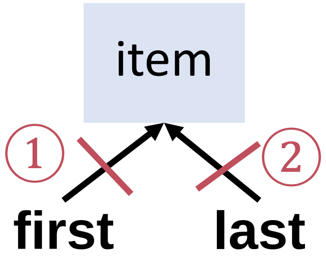
first last

Dequeue

last != first



last == first



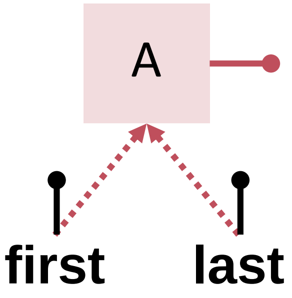
Naive Approach

Enqueue (q, new)

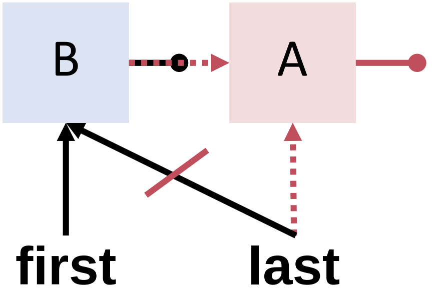
```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last != NIL THEN
e2      CAS(last.next, NIL, new);
    ELSE
e3      CAS(q.first, NIL, new);
    END
```

Dequeue (q)

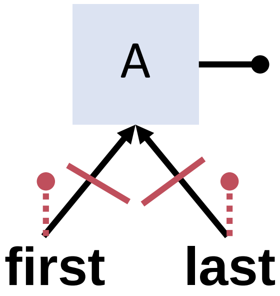
```
    REPEAT
        first= CAS(q.first, null, null);
d1      IF first = NIL THEN RETURN NIL END;
        next = CAS(first.next, NIL,NIL)
d2  UNTIL CAS(q.first, first, next) = first;
    IF next == NIL THEN
d3      CAS(q.last, first, NIL);
    END
```



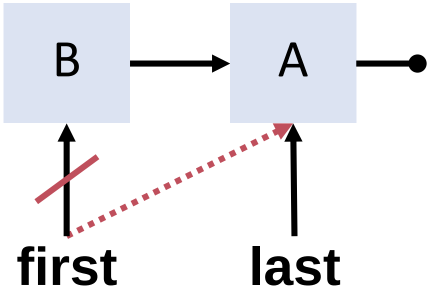
e1 + e3



e1 + e2



d2 + d3

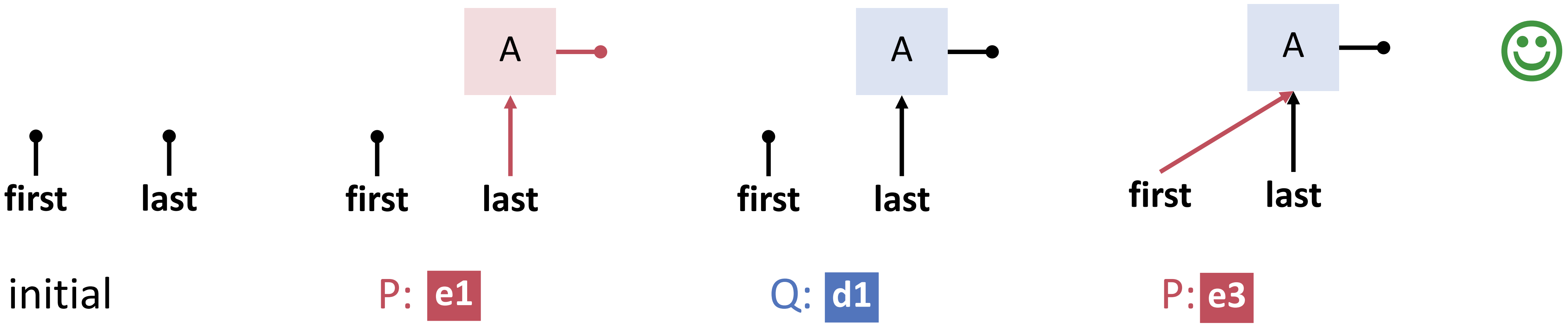


d2

Scenario

Process P enqueues A

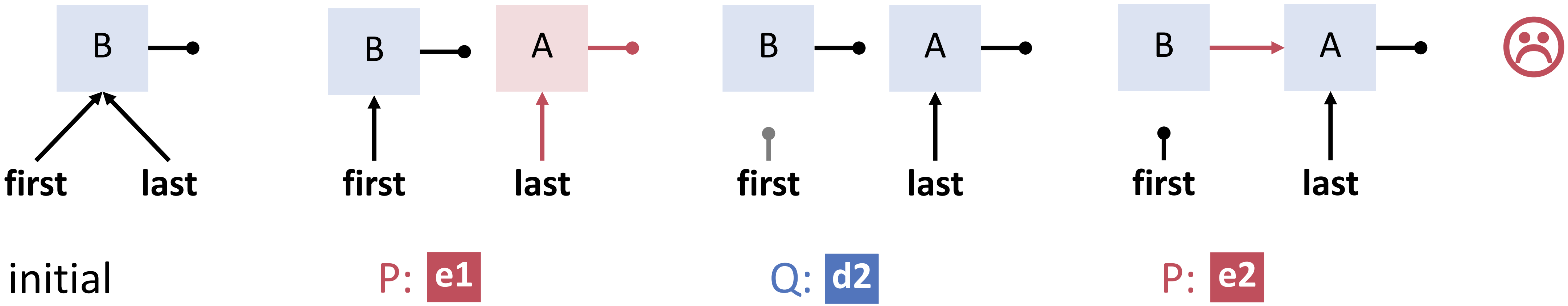
Process Q dequeues



Scenario

Process P enqueues A

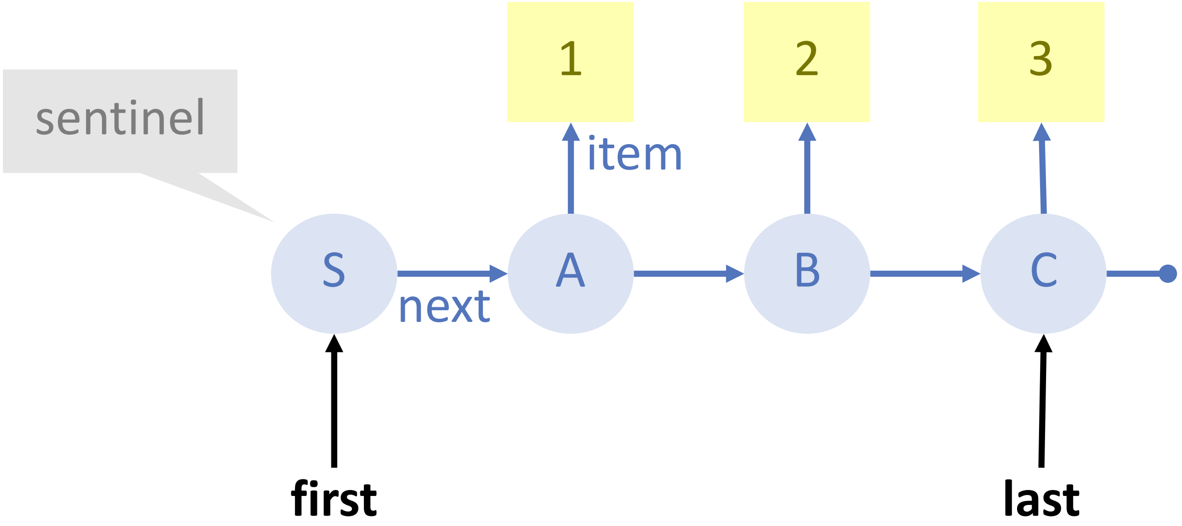
Process Q dequeues



Analysis

- The problem is that enqueue and dequeue do under some circumstances have to update **several pointers at once** [first, last, next]
- The transient inconsistency can lead to permanent data structure corruption
- Solutions to this particular problem are not easy to find if no double compare and swap (or similar) is available
- Need another approach: Decouple enqueue and dequeue with a sentinel. A consequence is that the **queue cannot be in-place.**

Queues with Sentinel



Queue empty: first = last

Queue nonempty: first # last

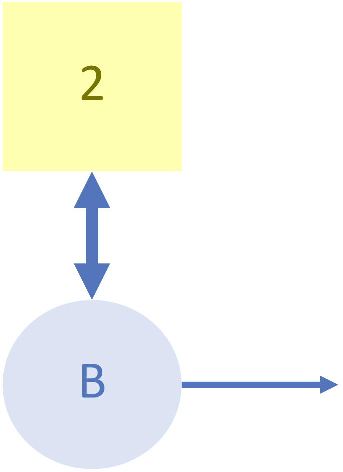
Invariants: first # NIL

last # NIL

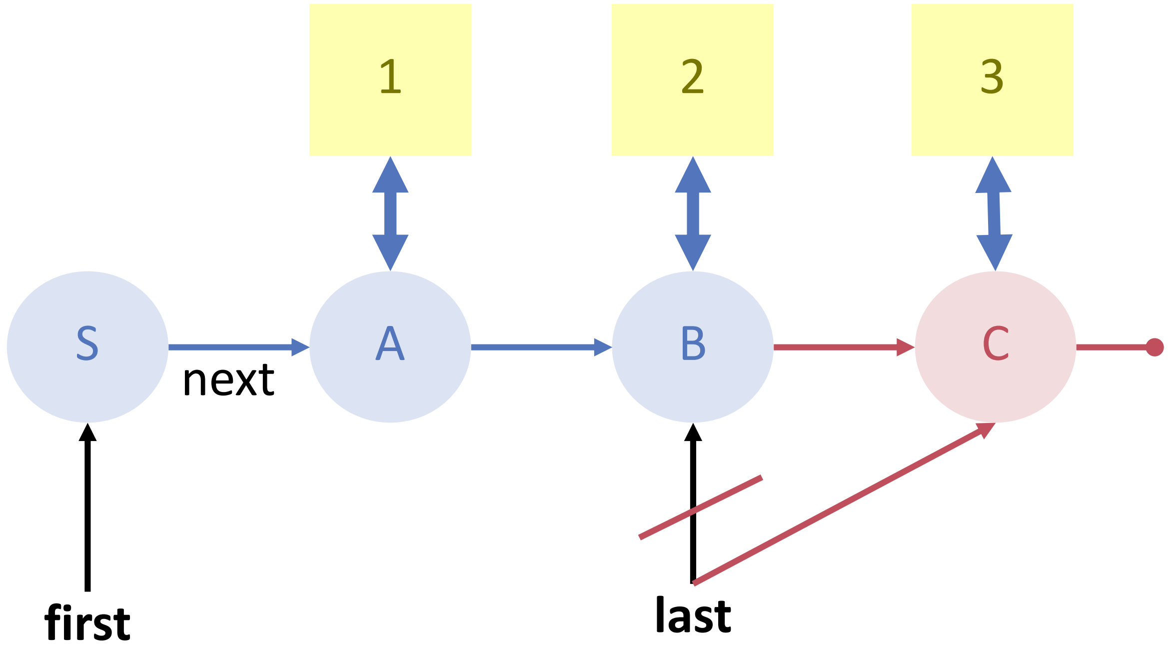
Node Reuse

simple idea:

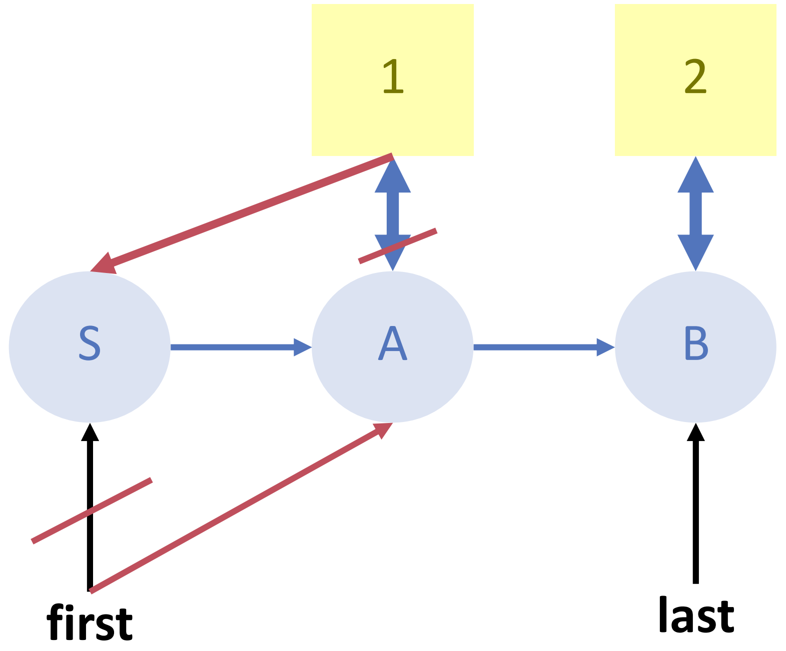
link from node to item
and from item to node



Enqueue and Dequeue with Sentinel

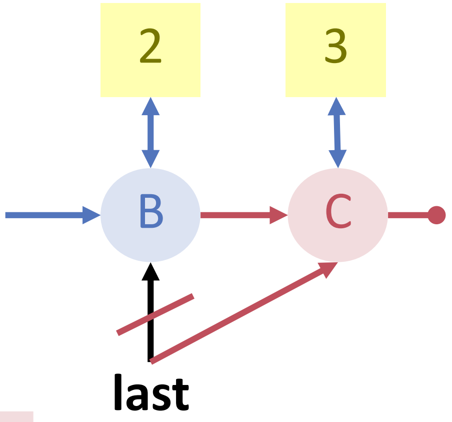


Item enqueued together with associated node.



A becomes the new sentinel.
S associated with free item.

Enqueue



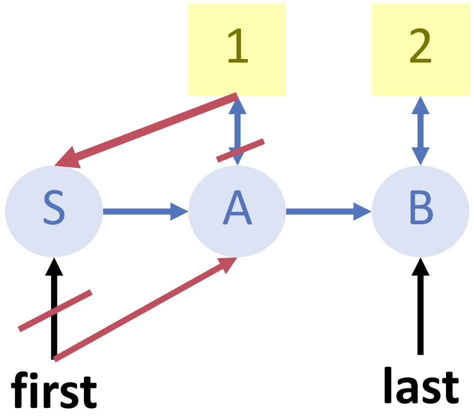
```
PROCEDURE Enqueue- (item: Item; VAR queue: Queue);
VAR node, last, next: Node;
BEGIN
   node := Allocate();
   node.item := Item:
   LOOP
      last := CAS (queue.last, NIL, NIL);
      next := CAS (last.next, NIL, node);
      IF next = NIL THEN EXIT END;
      IF CAS (queue.last, last, next) # last THEN CPU.Backoff END;
   END;
   ASSERT (CAS (queue.last, last, node) # NIL);
END Enqueue;
```

Set last node's next pointer

If failed, then **help other processes to set last node → Progress guarantee**

Set last node, can fail but then others have already helped

Dequeue



```
PROCEDURE Dequeue- (VAR item: Item; VAR queue: Queue): BOOLEAN;
VAR first, next, last: Node;
BEGIN
   LOOP
      first := CAS (queue.first, NIL, NIL);
      next := CAS (first.next, NIL, NIL);
      IF next = NIL THEN RETURN FALSE END;
      last := CAS (queue.last, first, next);
      item := next.item;
      IF CAS (queue.first, first, next) = first THEN EXIT END;
      CPU.Backoff;
   END;
   item.node := first;
   RETURN TRUE;
END Dequeue;
```

Remove inconsistency, **help other processes to set last pointer**

set first pointer

associate node with first

ABA

Problems of unbounded lock-free queues

- unboundedness → dynamic memory allocation is inevitable
  - if the memory system is not lock-free, we are back to square 1
  - **reusing nodes** to avoid memory issues causes the **ABA problem** (where ?!)

- Employ **Hazard Pointers** now.

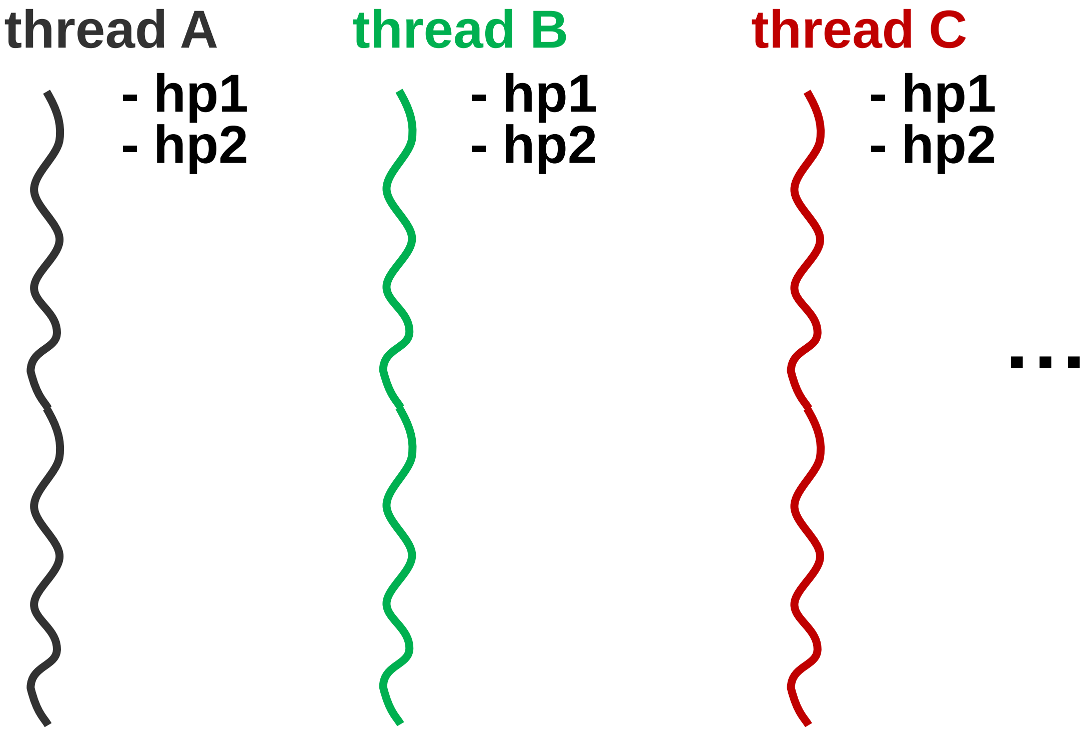
Hazard Pointers

- Store pointers of memory references about to be accessed by a thread
- Memory allocation checks all hazard pointers to avoid the ABA problem

Number of threads unbounded

→ time to check hazard pointers also unbounded!

→ difficult dynamic bookkeeping!



thread A
- hp1
- hp2

thread B
- hp1
- hp2

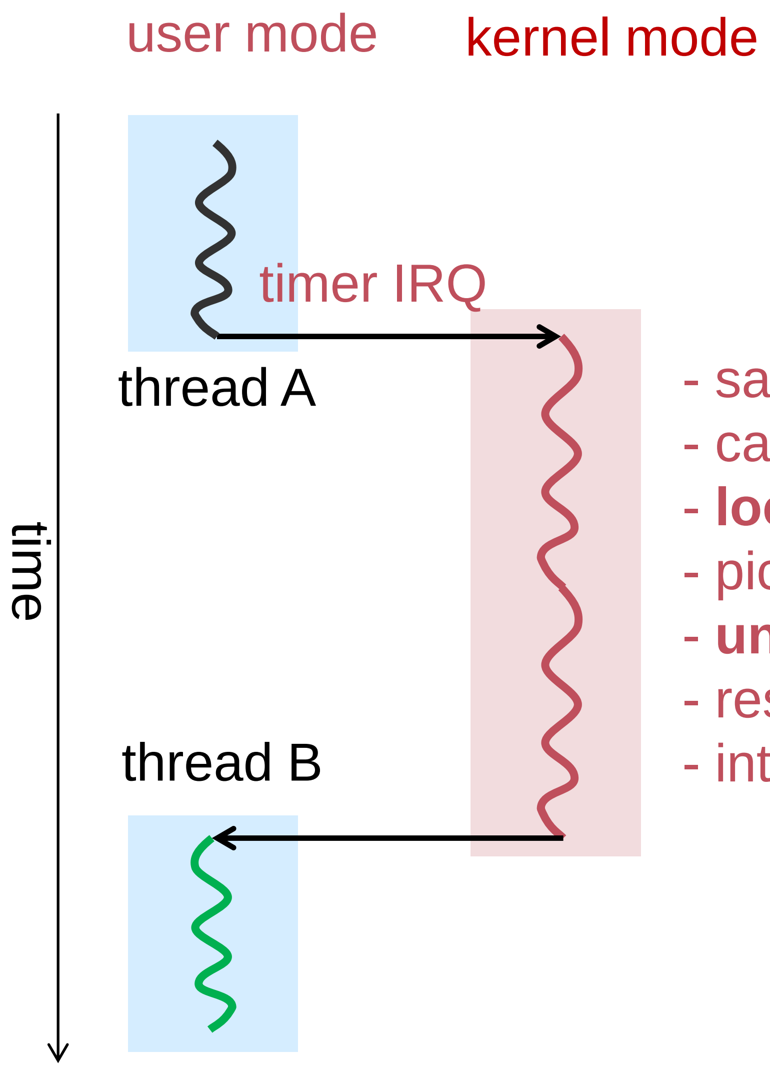
thread C
- hp1
- hp2

…

Key idea of Cooperative MT & Lock-free Algorithms

Use the **guarantees of cooperative multitasking** to implement efficient unbounded lock-free queues

Time Sharing

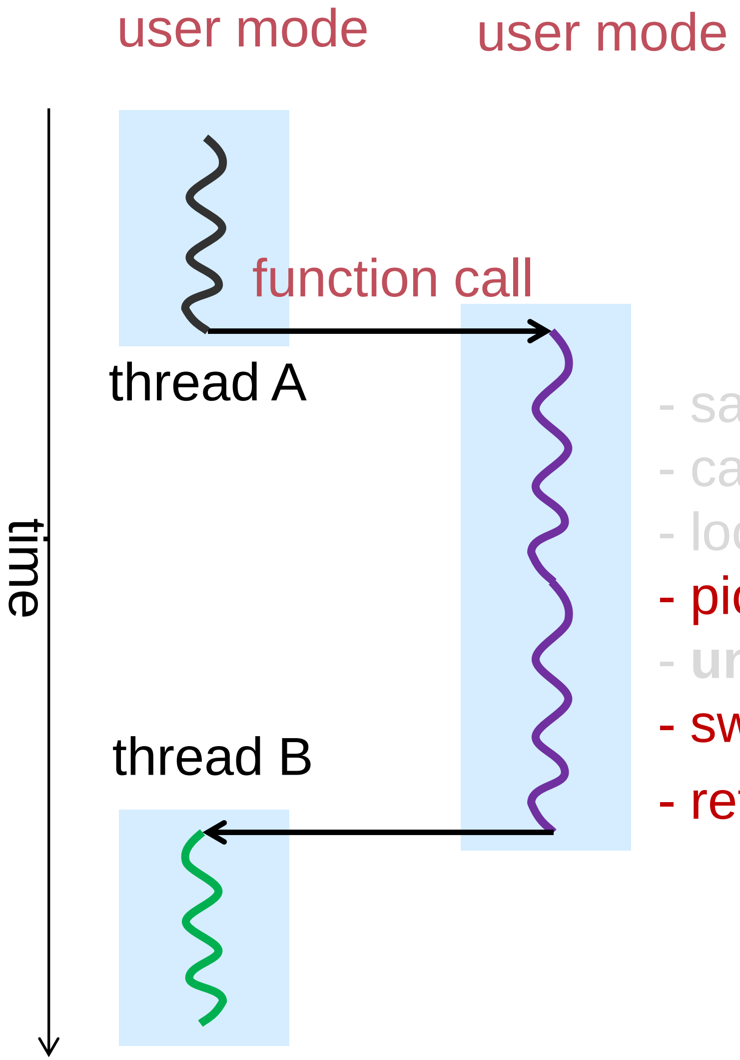


- save processor registers (**assembly**)
- call timer handler (**assembly**)
- **lock scheduling queue**
- pick new process to schedule
- **unlock scheduling queue**
- restore processor registers (**assembly**)
- interrupt return (**assembly**)

inherently hardware dependent
(timer programming
context save/restore)

inherently non-parallel
(scheduler lock)

Cooperative Multitasking



- save processor registers (assembly)
- call timer handler (assembly)
- lock scheduling queue
- pick new process to schedule (lockfree)
- **unlock scheduling queue**
- switch base pointer
- return from function call

hardware independent

(no timer required,
standard procedure calling convention
takes care of register save/restore)

finest granularity

(no lock)

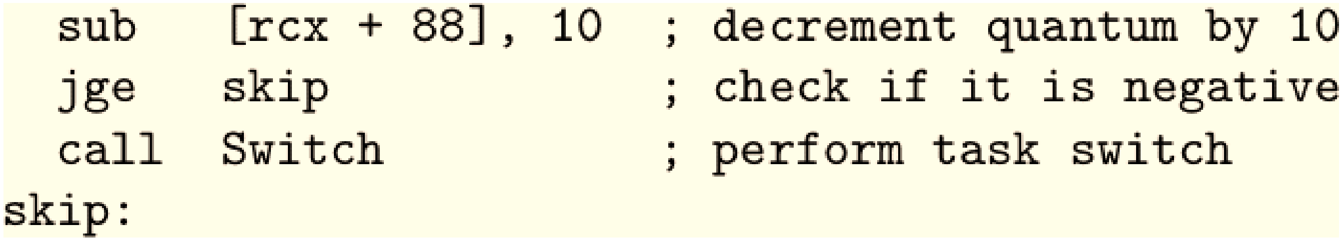
Implicit Cooperative Multitasking

Ensure cooperation

- Compiler automatically inserts code at specific points in the code

Details

- Each process has a quantum
- At regular intervals, the compiler inserts code to decrease the quantum and calls the scheduler if necessary



```
  sub   [rcx + 88], 10   ; decrement quantum by 10
  jge   skip             ; check if it is negative
  call  Switch           ; perform task switch
skip:
```

implicit cooperative multitasking – AMD64

uncooperative

```
PROCEDURE Enqueue- (item: Item; VAR queue: Queue);
BEGIN {UNCOOPERATIVE}
   ...
   (* no scheduling here ! *)
   ...
END Enqueue;
```

zero overhead processor local "locks"

Implicit Cooperative Multitasking

Pros

- extremely light-weight – cost of a regular function call
- allow for global optimization – calls to scheduler known to the compiler
- **zero overhead processor local locks**

Cons

- overhead of inserted scheduler code
- currently sacrifice one hardware register (`rcx`)
- require a special compiler and access to the source code

Cooperative MT & Lock-free Algorithms

Guarantees of cooperative MT

- No more than M threads are executing inside an `uncooperative` block (M = # of processors)
- No thread switch occurs while a thread is running on a processor

→ hazard pointers can be associated with the processor

- Number of hazard pointers limited by M
- Search time constant

thread-local storage → processor local storage

No Interrupts?

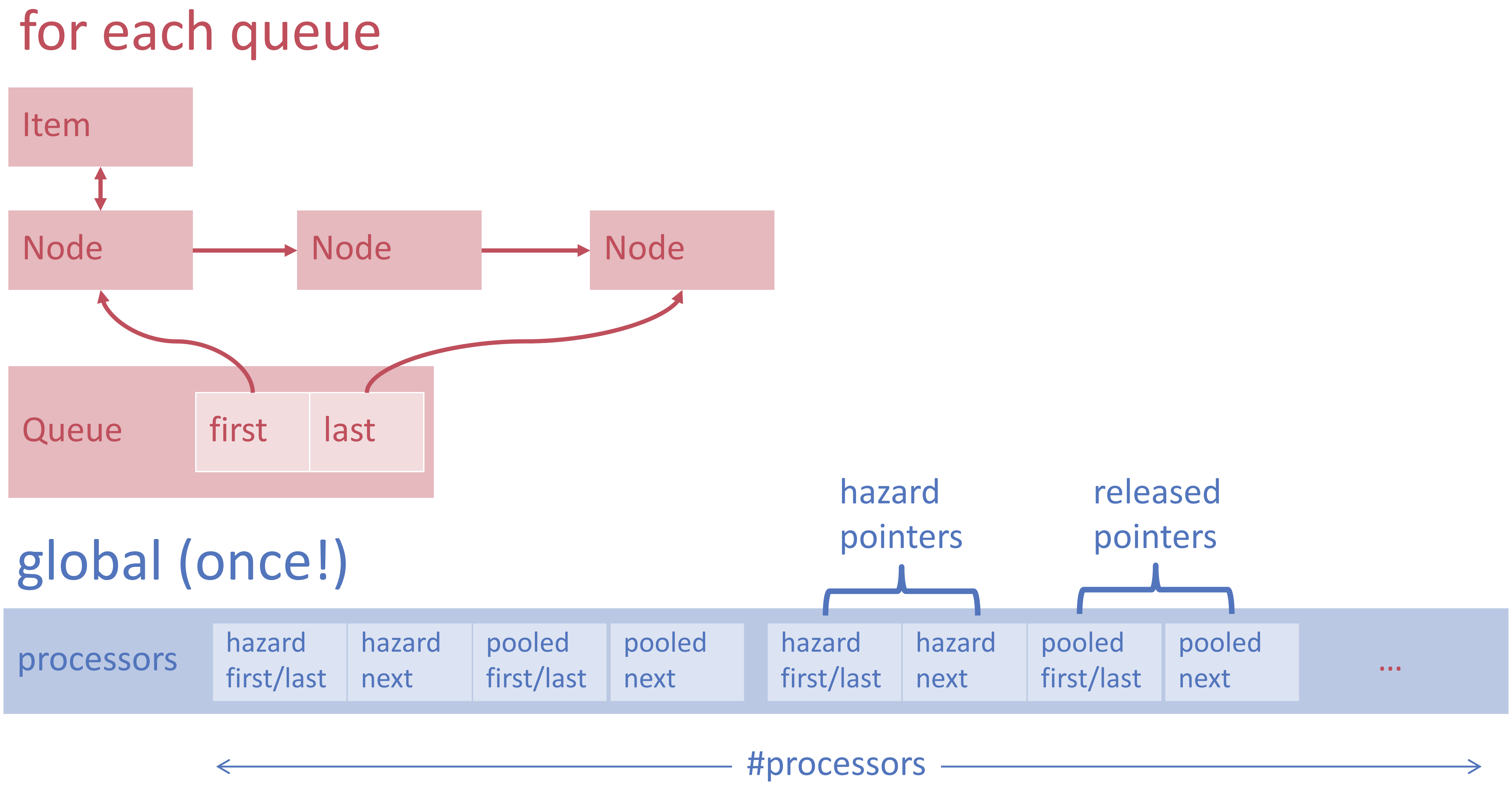
Device drivers are interrupt-driven

- breaks all assumptions made so far (number of contenders limited by the number of processors)

Key idea: model interrupt handlers as virtual processors

- M = # of physical processors + # of potentially concurrent interrupts

Queue Data Structures



for each queue

Item

Node

Node

Node

Queue

first

last

global (once!)

hazard
pointers

released
pointers

processors

hazard
first/last

hazard
next

pooled
first/last

pooled
next

hazard
first/last

hazard
next

pooled
first/last

pooled
next

...

#processors

Marking Hazarduous

```
PROCEDURE Access (VAR node, reference: Node; pointer: SIZE);
VAR value: Node; index: SIZE;
BEGIN {UNCOOPERATIVE, UNCHECKED}
   index := Processors.GetCurrentIndex ();
   LOOP
      processors[index].hazard[pointer] := node;
      value := CAS (reference, NIL, NIL);
      IF value = node THEN EXIT END;
      node := value;
   END;
END Access;

PROCEDURE Discard (pointer: SIZE);
BEGIN {UNCOOPERATIVE, UNCHECKED}
   processors[Processors.GetCurrentIndex ()].hazard[pointer] := NIL;
END Discard;
```

guarantee: no change to reference after node was set hazarduous

Node Reuse

```
PROCEDURE Acquire (VAR node {UNTRACED}: Node): BOOLEAN;
VAR index := 0: SIZE;
BEGIN {UNCOOPERATIVE, UNCHECKED}
   WHILE (node # NIL) & (index # Processors.Maximum) DO
      IF node = processors[index].hazard[First] THEN
         Swap (processors[index].pooled[First], node); index := 0;
      ELSIF node = processors[index].hazard[Next] THEN
         Swap (processors[index].pooled[Next], node); index := 0;
      ELSE
         INC (index)
      END;
   END;
   RETURN node # NIL;
END Acquire;
```

wait free algorithm to find non-hazarduous node for reuse (if any)

Lock-Free Enqueue with Node Reuse

```
node := item.node;
IF ~Acquire (node) THEN
   NEW (node);
END;
node.next := NIL; node.item := item;

LOOP
   last := CAS (queue.last, NIL, NIL);
   Access (last, queue.last, Last);
   next := CAS (last.next, NIL, node);
   IF next = NIL THEN EXIT END;
   IF CAS (queue.last, last, next) # last THEN CPU.Backoff END;
END;
ASSERT (CAS (queue.last, last, node) # NIL, Diagnostics.InvalidQueue);
Discard (Last);
```

reuse

mark last hazarduous

unmark last

Lock-Free Dequeue with Node Reuse

```
LOOP
   first := CAS (queue.first, NIL, NIL);
   Access (first, queue.first, First);                          mark first hazarduous
   next := CAS (first.next, NIL, NIL);
   Access (next, first.next, Next);                              mark next hazarduous
   IF next = NIL THEN
      item := NIL; Discard (First); Discard (Next); RETURN FALSE    unmark first and next
   END;
   last := CAS (queue.last, first, next);
   item := next.item;
   IF CAS (queue.first, first, next) = first THEN EXIT END;
   Discard (Next); CPU.Backoff;                                  unmark next
END;
first.item := NIL; first.next := first; item.node := first;
Discard (First); Discard (Next); RETURN TRUE;                   unmark first and next
```

Scheduling -- Activities

```
TYPE Activity* = OBJECT {DISPOSABLE} (Queues.Item)
VAR
```

access to current processor

stack management

quantum and scheduling

active object

```
END Activity;
```

accessed via activity register

`(cf. Activities.Mod)`

Lock-free scheduling

Use non-blocking Queues and discard coarser granular locking.

Problem: Finest granular protection makes races possible that did not occur previously:

```
current := GetCurrentTask()
next := Dequeue(readyqueue)
Enqueue(current, readyqueue)
SwitchTo(next)
```

Other thread can dequeue and run (on the stack of) the currently executing thread!

Task Switch Finalizer

```
PROCEDURE Switch-;
VAR currentActivity {UNTRACED}, nextActivity: Activity;
BEGIN {UNCOOPERATIVE, SAFE}
  currentActivity := SYSTEM.GetActivity ()(Activity);
  IF Select (nextActivity, currentActivity.priority) THEN
      SwitchTo (nextActivity, Enqueue, ADDRESS OF readyQueue[currentActivity.priority]);
      FinalizeSwitch;
  ELSE
      currentActivity.quantum := Quantum;
  END;
END Switch;
```

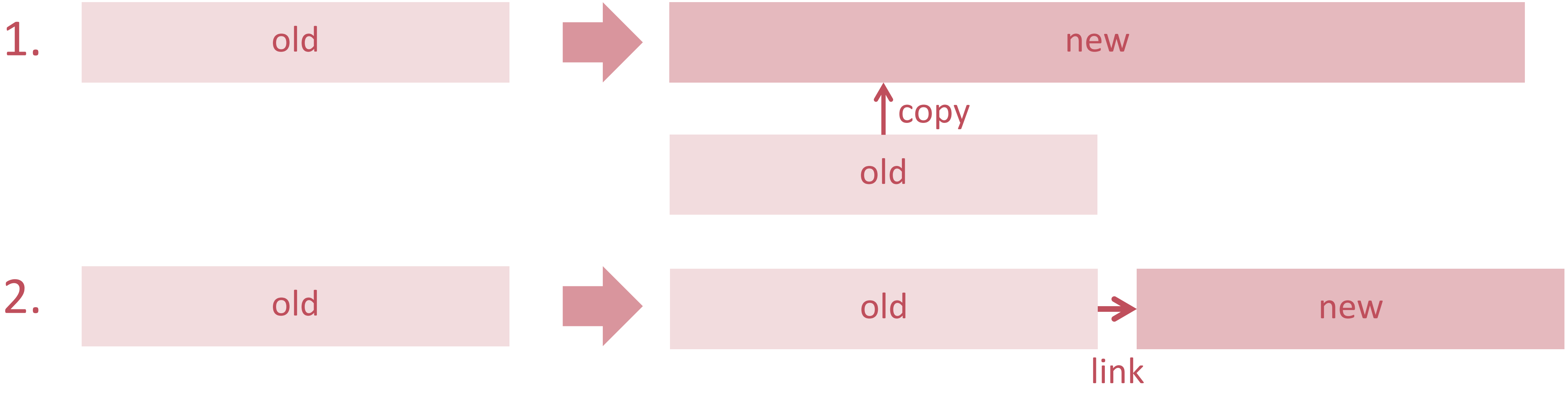
Enqueue runs on new thread

Stack Management

Stacks organized as Heap Blocks.

Stack check instrumented at beginning of each procedure.

Stack expansion possibilities



Copying stack

Must keep track of all pointers from stack to stack

Requires book-keeping of

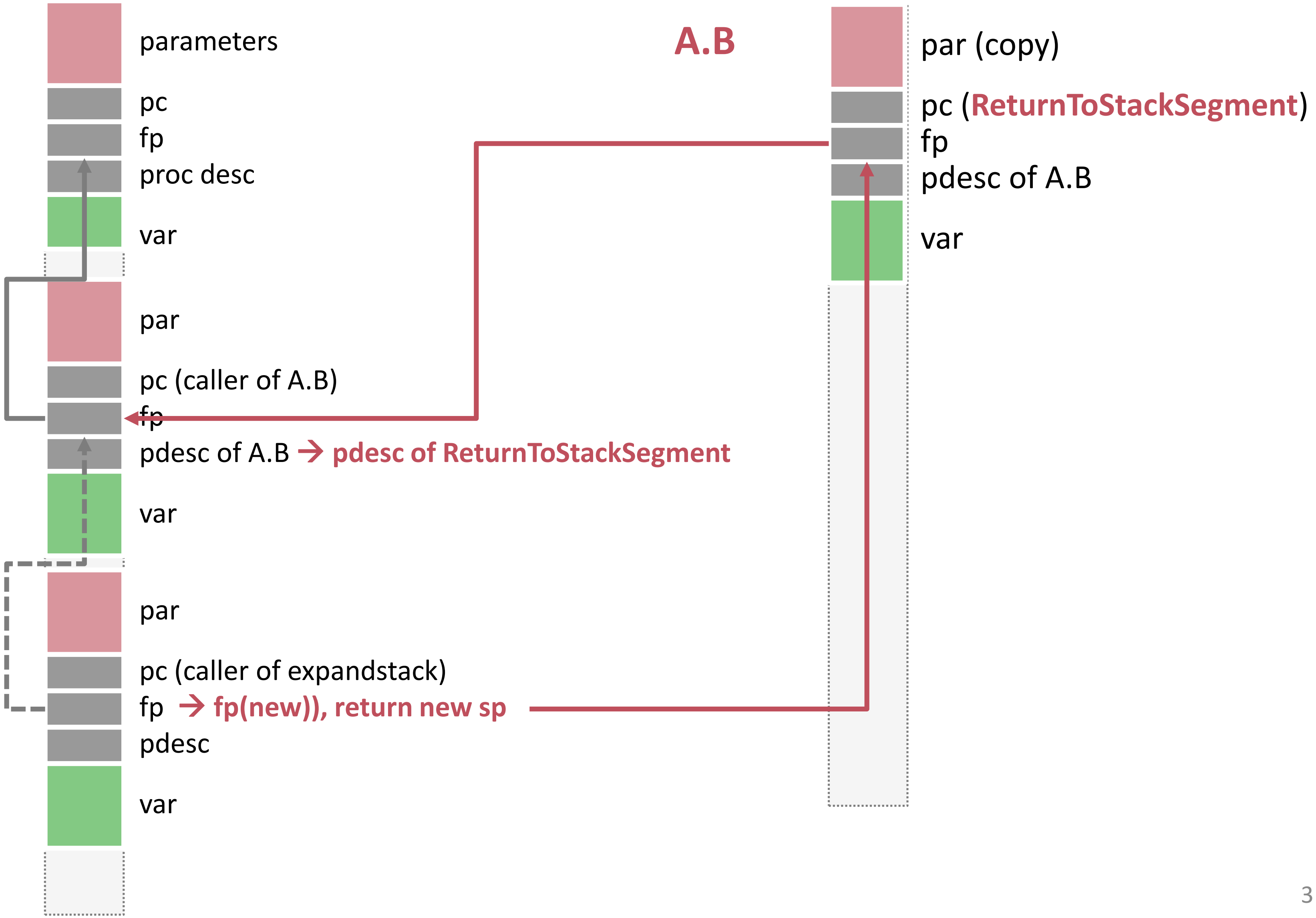
- call-by-reference parameters
  - open arrays
  - records
- unsafe pointer on stack
  - e.g. file buffers

turned out to be **prohibitively expensive**

Linked Stack

- Instrumented call to ExpandStack
- End of current stack segment pointer included in process descriptor
- Link stacks on demand with new stack segment
- Return from stack segment inserted into call chain backlinks

Linked Stacks



caller of
A.B

parameters
pc
fp
proc desc
var

A.B
becomes frame of
ReturnToStackSegment

par
pc (caller of A.B)
fp
pdesc of A.B → **pdesc of ReturnToStackSegment**
var

ExpandStack

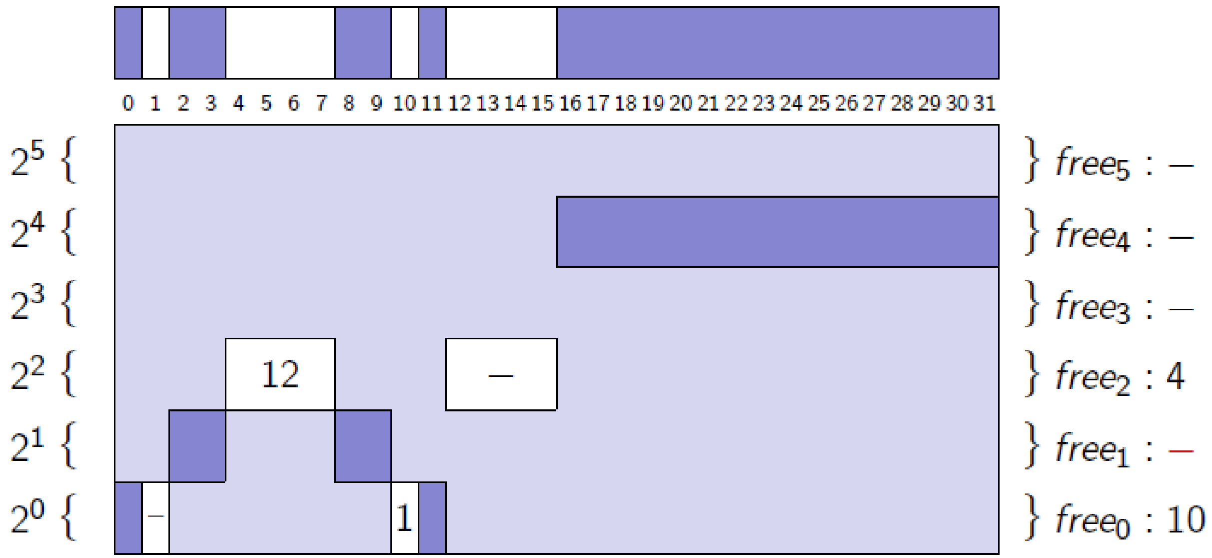
par
pc (caller of expandstack)
fp → **fp(new)), return new sp**
pdesc
var

A.B

par (copy)
pc (**ReturnToStackSegment**)
fp
pdesc of A.B
var

Lock-Free Memory Management

- Allocation / De-allocation implemented using only lock-free algorithms
- Buddy system with independent (lock-free) queues for the different block sizes
- Lock-free mark-sweep garbage collector
- Several garbage collectors can run in parallel

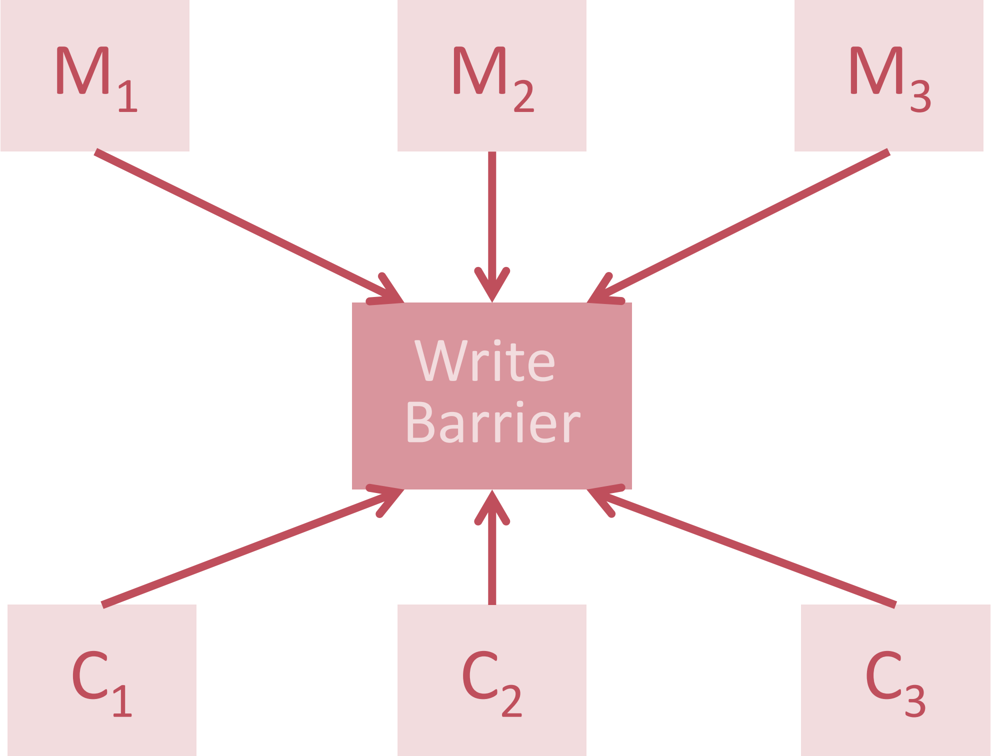


Lock-free Garbage Collector

- Mark & Sweep
- Precise
- Optional

- Incremental
- Concurrent
- Parallel

Synchronisation



Mutators

M_1 M_2 M_3

Mark

Write Barrier

Collectors

C_1 C_2 C_3

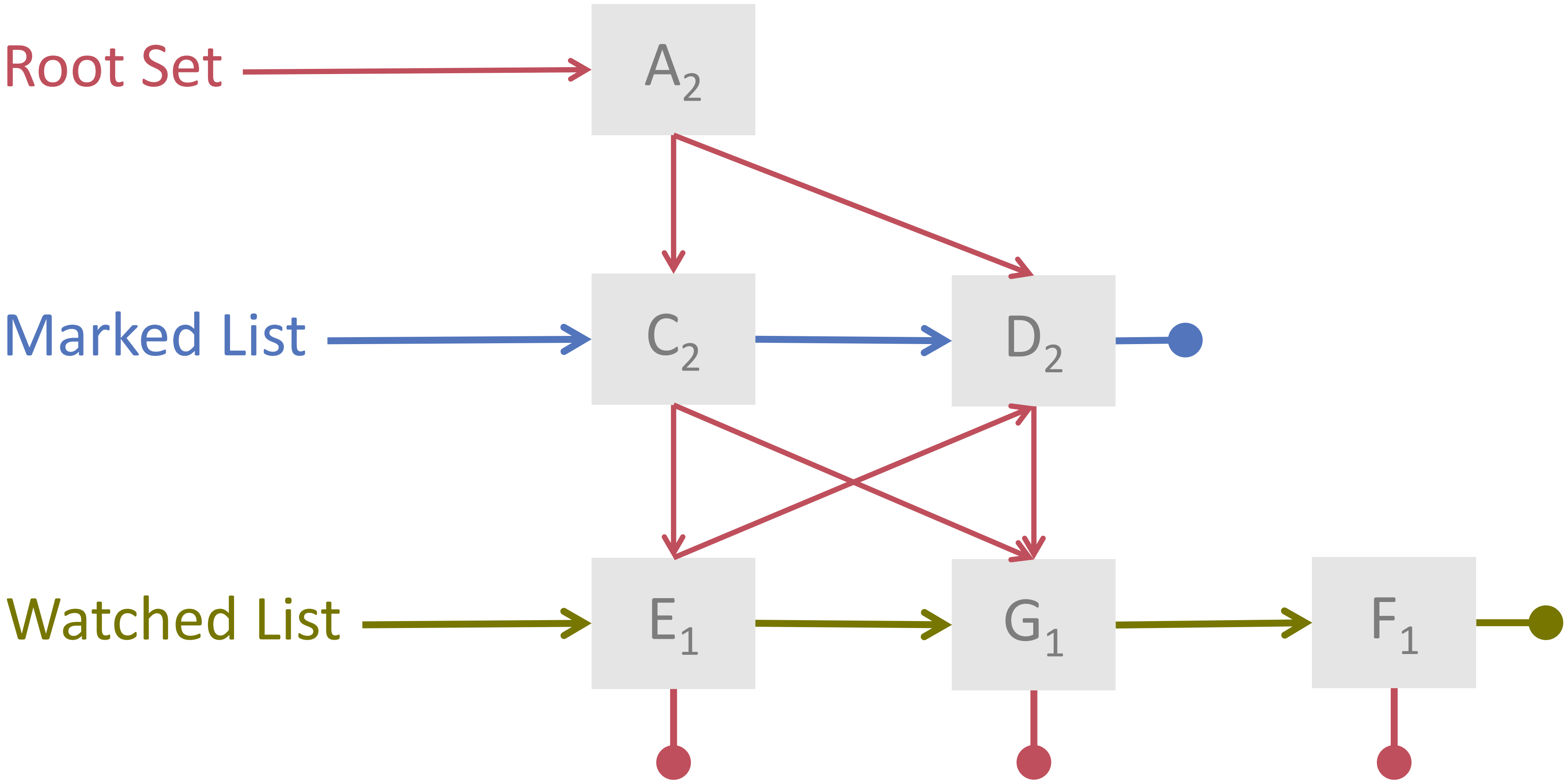
Traverse

Data Structures

	Global	**Per Object**
Mark Bit	Cycle Count	Cycle Count
Marklist	Marked First	Next Marked
Watchlist	Watched First	Next Watched
Root Set	Global References	Local Refcount

Example

Cycle Count = 2



Achieving (Almost) Complete Portability

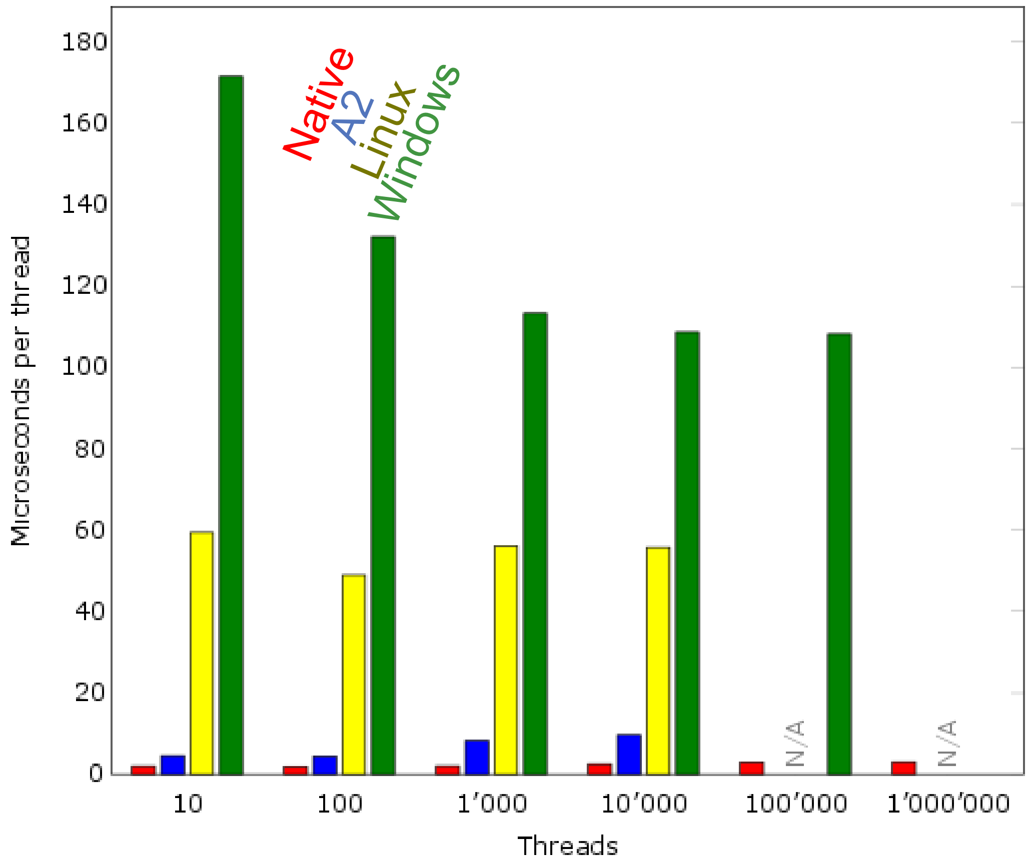
- **Lock-free A2 kernel written exclusively in a high-level language**
- no timer interrupt required → scheduler hardware independent
- no virtual memory → no separate address spaces → everything runs in user mode, all the time
- hardware-dependent functions (CAS) are pushed into the language
- “almost”:
  - we need a **minimal** stub written in assembly code to
    - initialize memory mappings
    - initialize all processors

How well does it perform? (Simplicity, Portability)

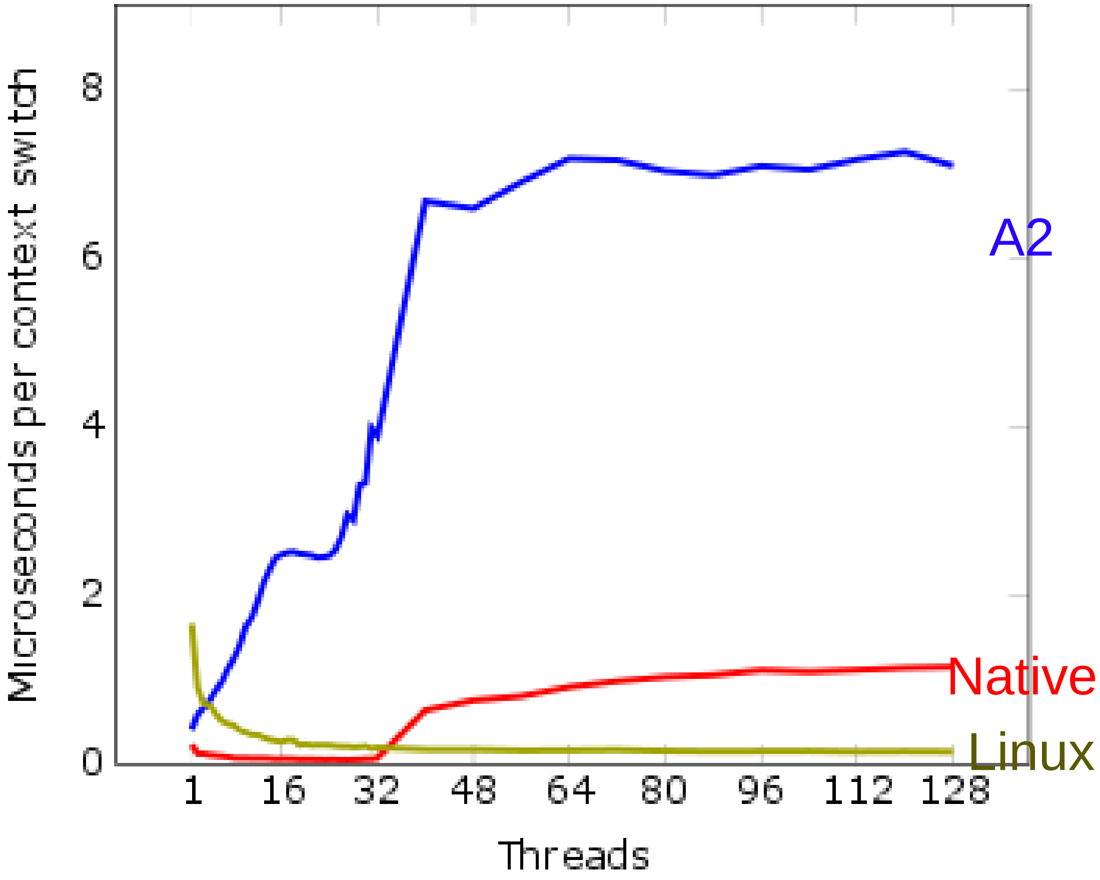
Component	**Lines of Code (Kernel)**
Interrupt Handling	301
Memory Management (including GC!)	352
Modules	82
Multiprocessing	213
Runtime Support	250
Scheduler	540
Total	1738 (28% of A2 orig)

How well does it perform? (Scheduler)

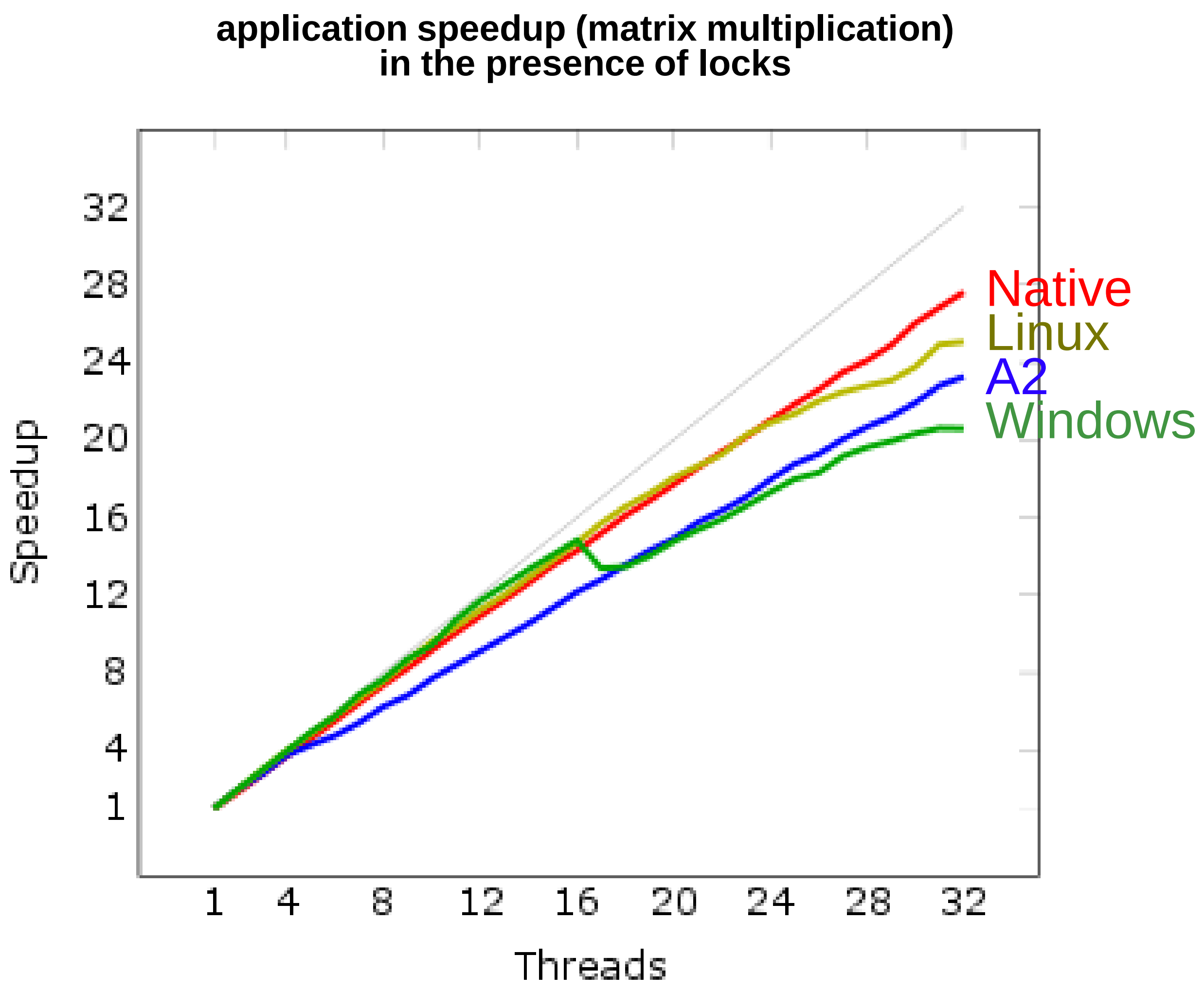
thread creation time

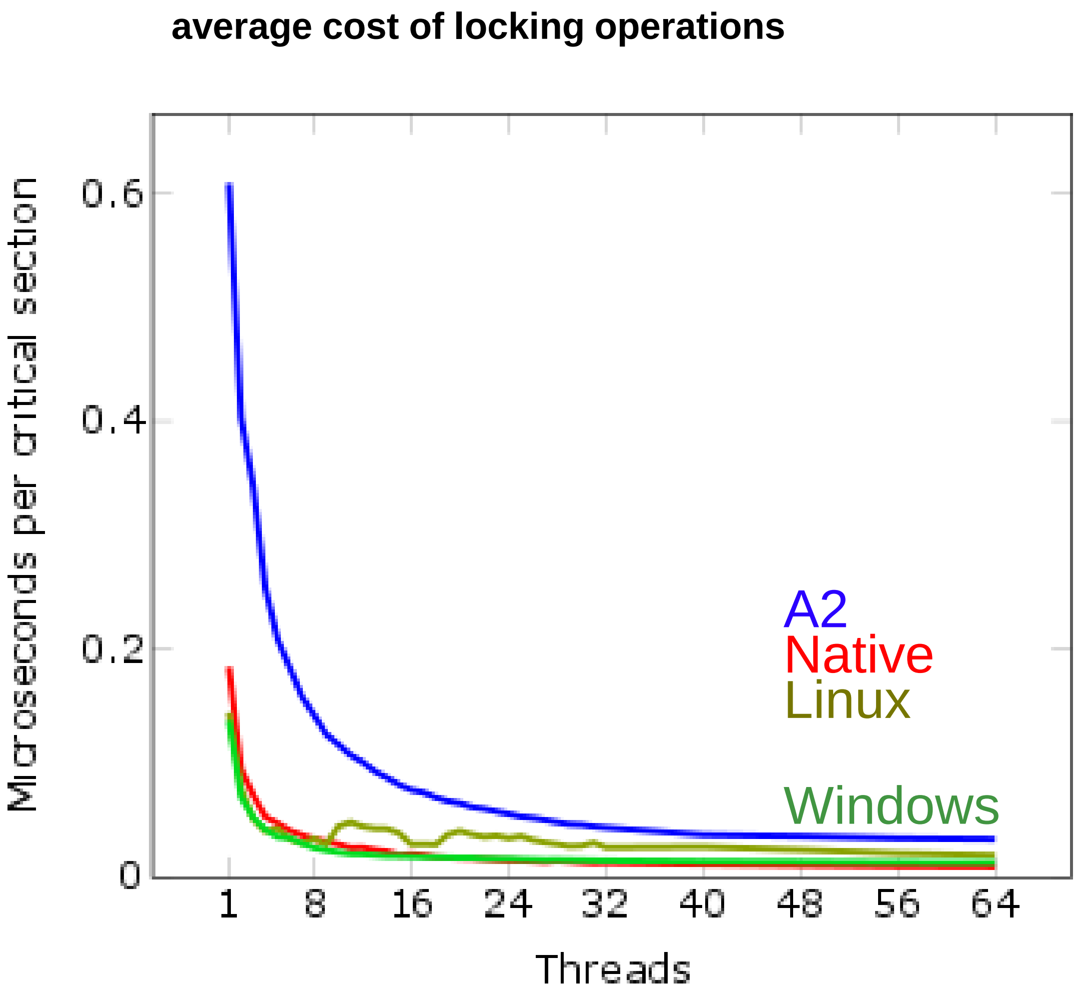


thread switching time

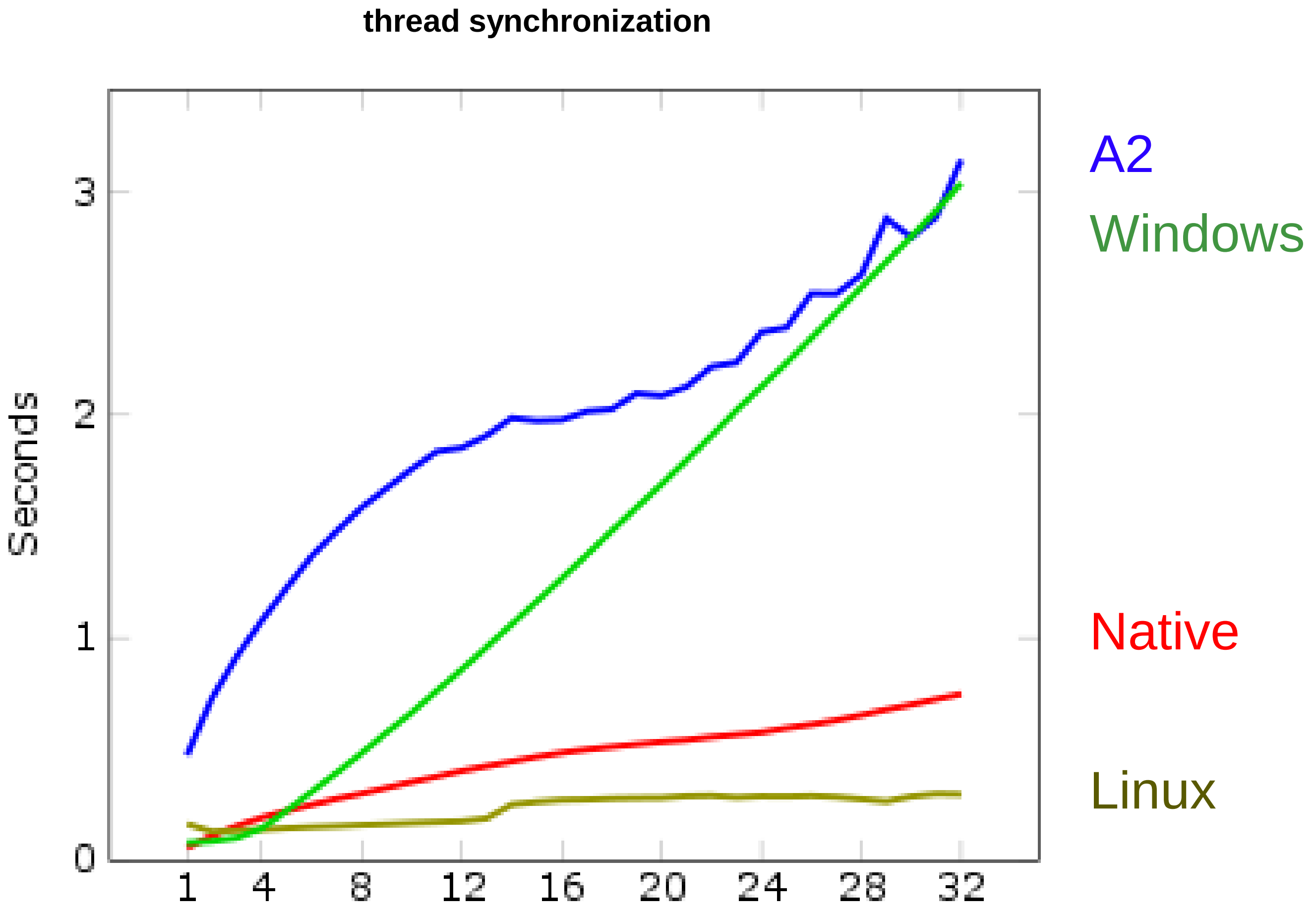


How well does it perform? (Scheduler)



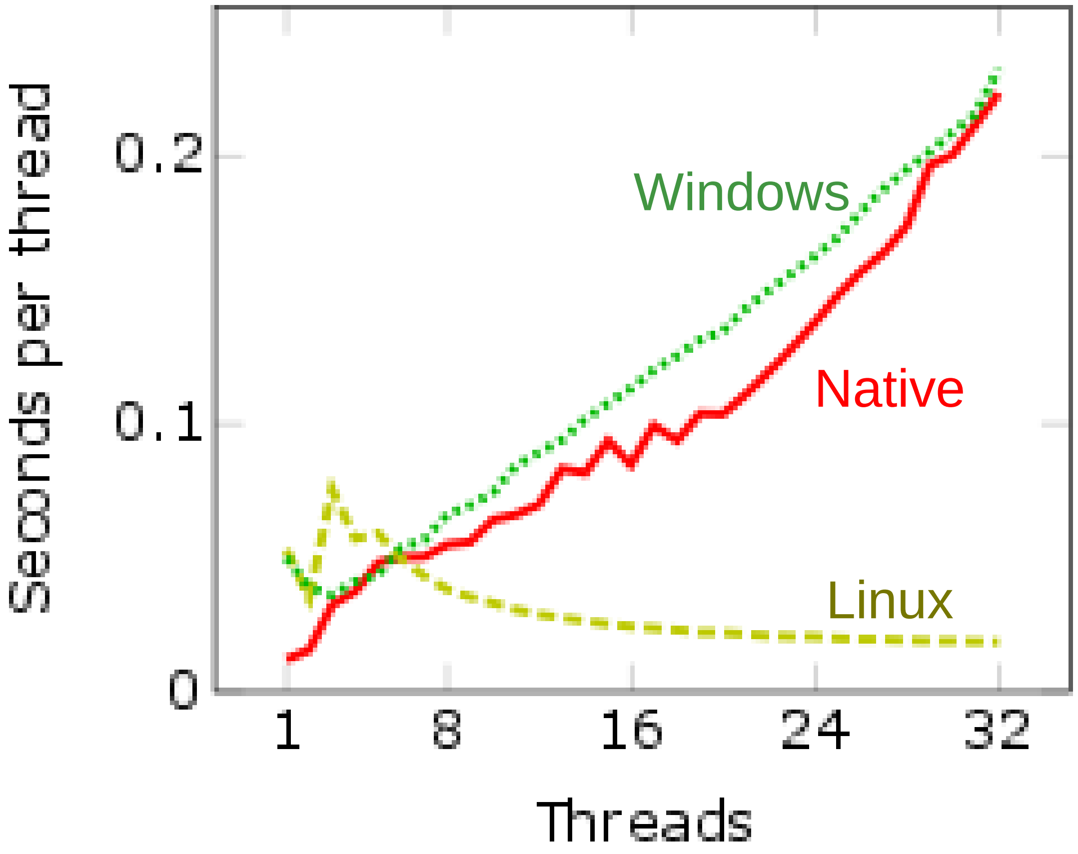


How well does it perform? (Scheduler)

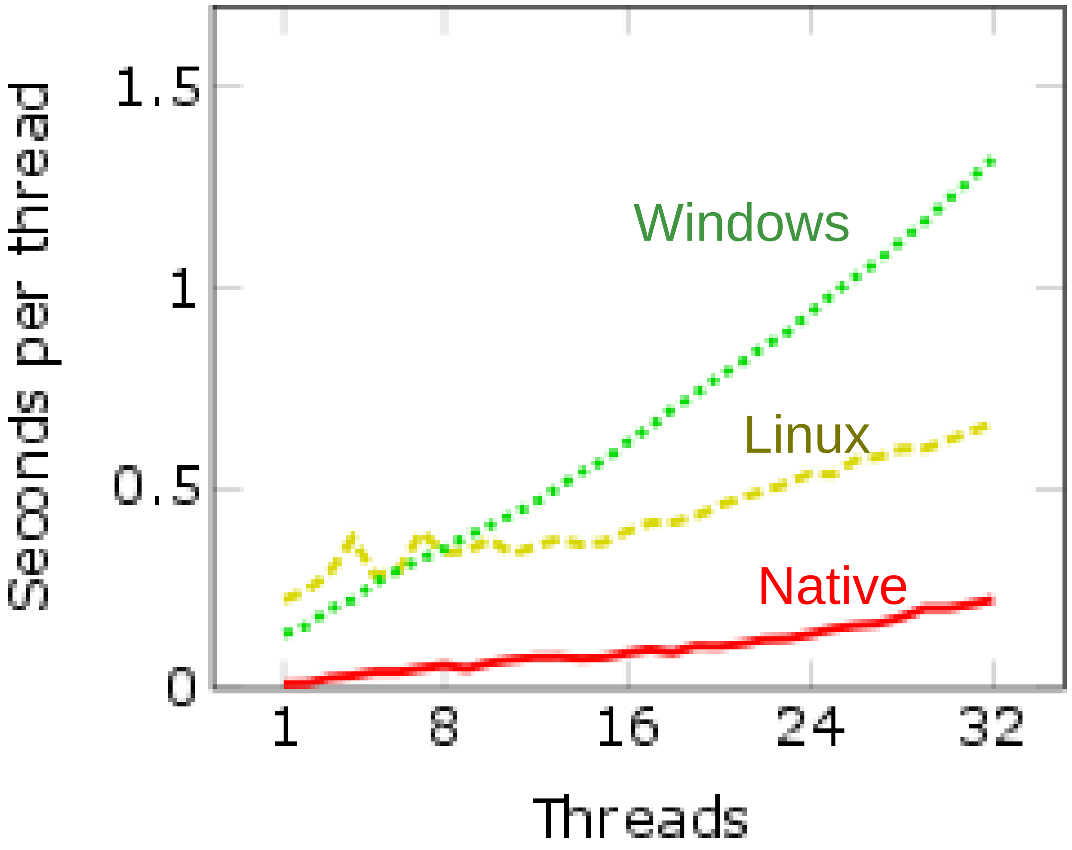


How well does it perform? (Memory Manager)

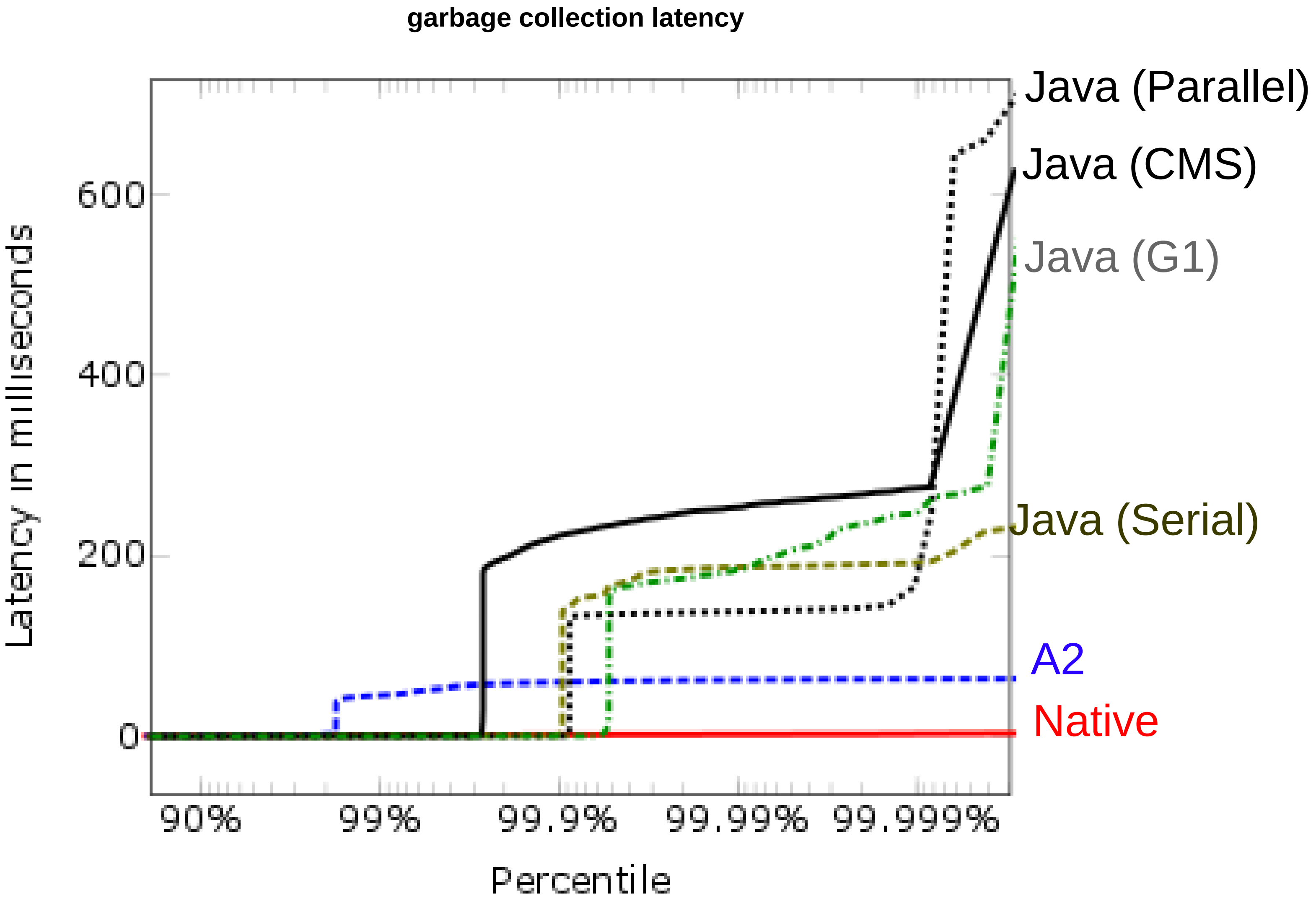
memory allocation of 1’000 byte blocks



memory allocation of 10’000 byte blocks



How well does it perform? (Memory Manager)



Lessons Learned

- Lock-free programming: new kind of problems in comparison to lock-based programming:

- Atomic update of several pointers / values impossible, leading to new kind of problems and solutions, such as threads that help each other in order to guarantee global progress
- ABA problem (which in many cases disappears with a Garbage Collector)

Conclusion

- **Lock-free Runtime**
  - consequent use of lock-free algorithms in the kernel
  - synchronization primitives (for applications) implemented on top
  - efficient unbounded lock-free queues
  - parallel and lock-free memory management with garbage collection

- **A completely lock-free runtime is feasible**
  - exploit guarantees of cooperative multitasking
  - performance is good considering
    - non-optimizing compiler
    - no load-balancing, no distributed run-queues