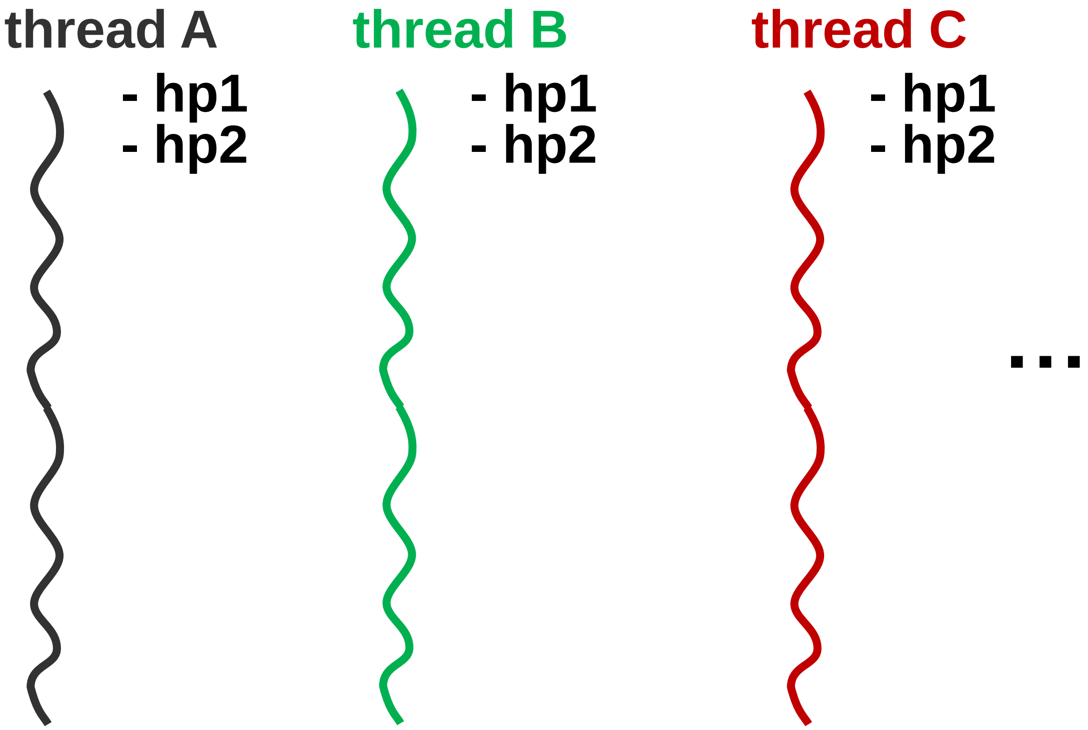
Hazard Pointers

- 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

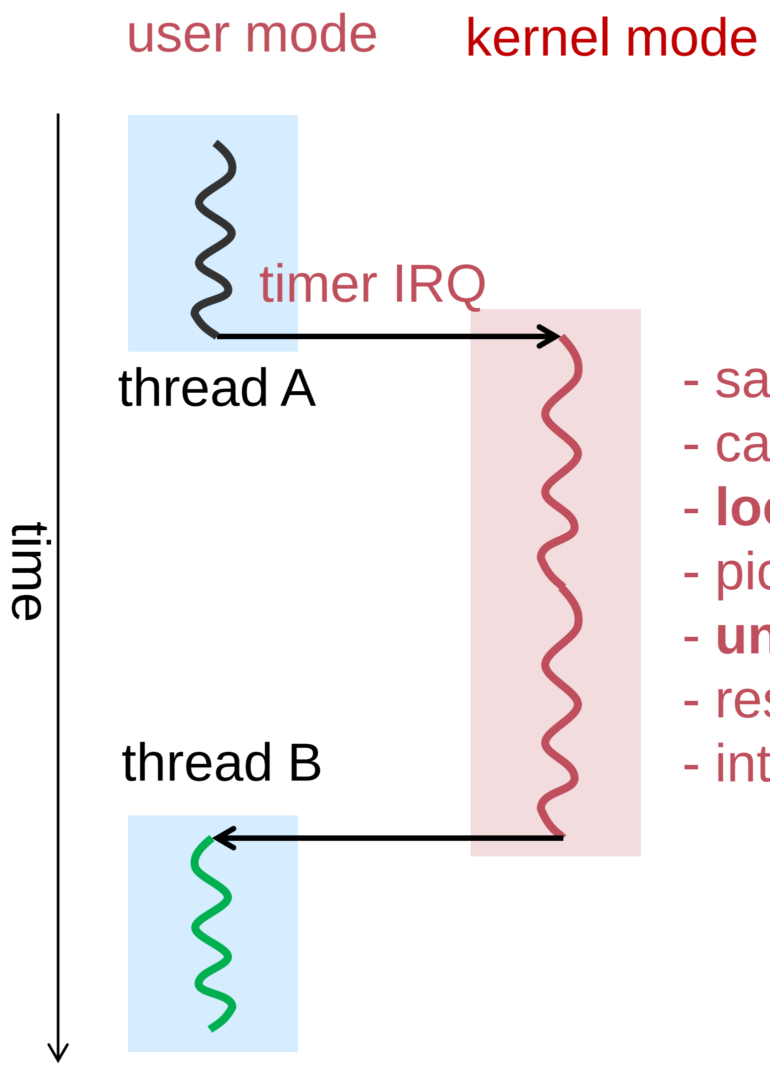
→time to check hazard pointers also unbounded!

→difficult dynamic bookkeeping!

Key idea of Cooperative MT & Lock-free Algorithms

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

Time Sharing

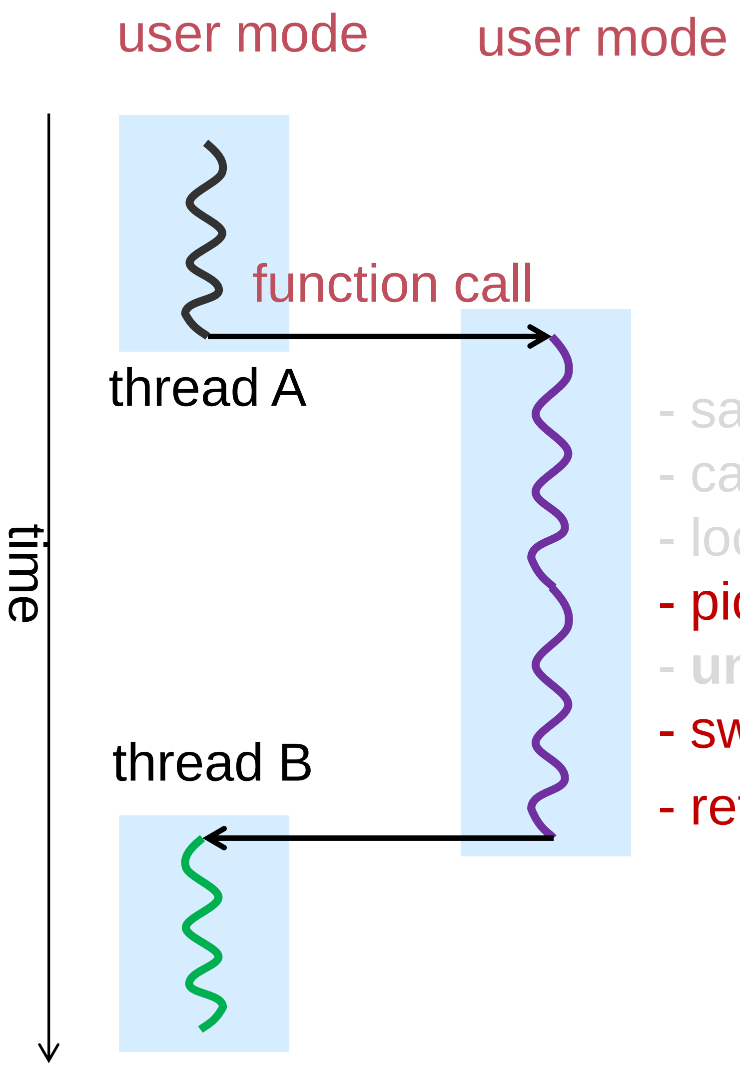


- save processor registers (**assembly**)
- call timer handler (**assembly**)
- **lock scheduling queue**
- pick new process to schedule
- **unlock scheduling queue**
- restore processor registers (**assembly**)
- interrupt return (**assembly**)

inherently hardware dependent
(timer programming
context save/restore)

inherently non-parallel
(scheduler lock)

Cooperative Multitasking



user mode

user mode

function call

thread A

thread B

time

- 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)

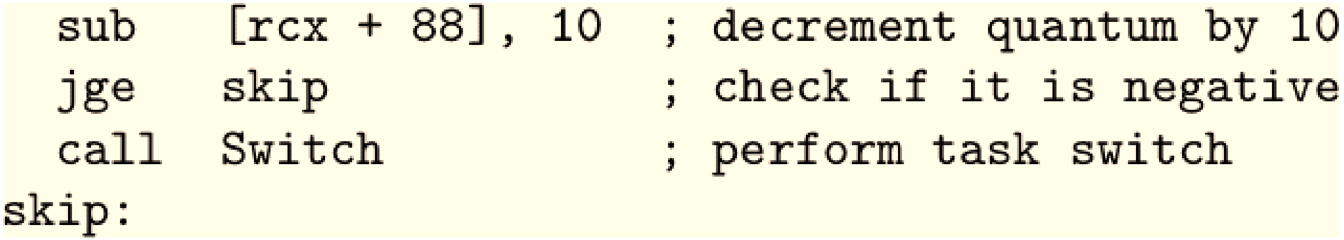
Implicit Cooperative Multitasking

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 (e.g. `rcx`)
- requires 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

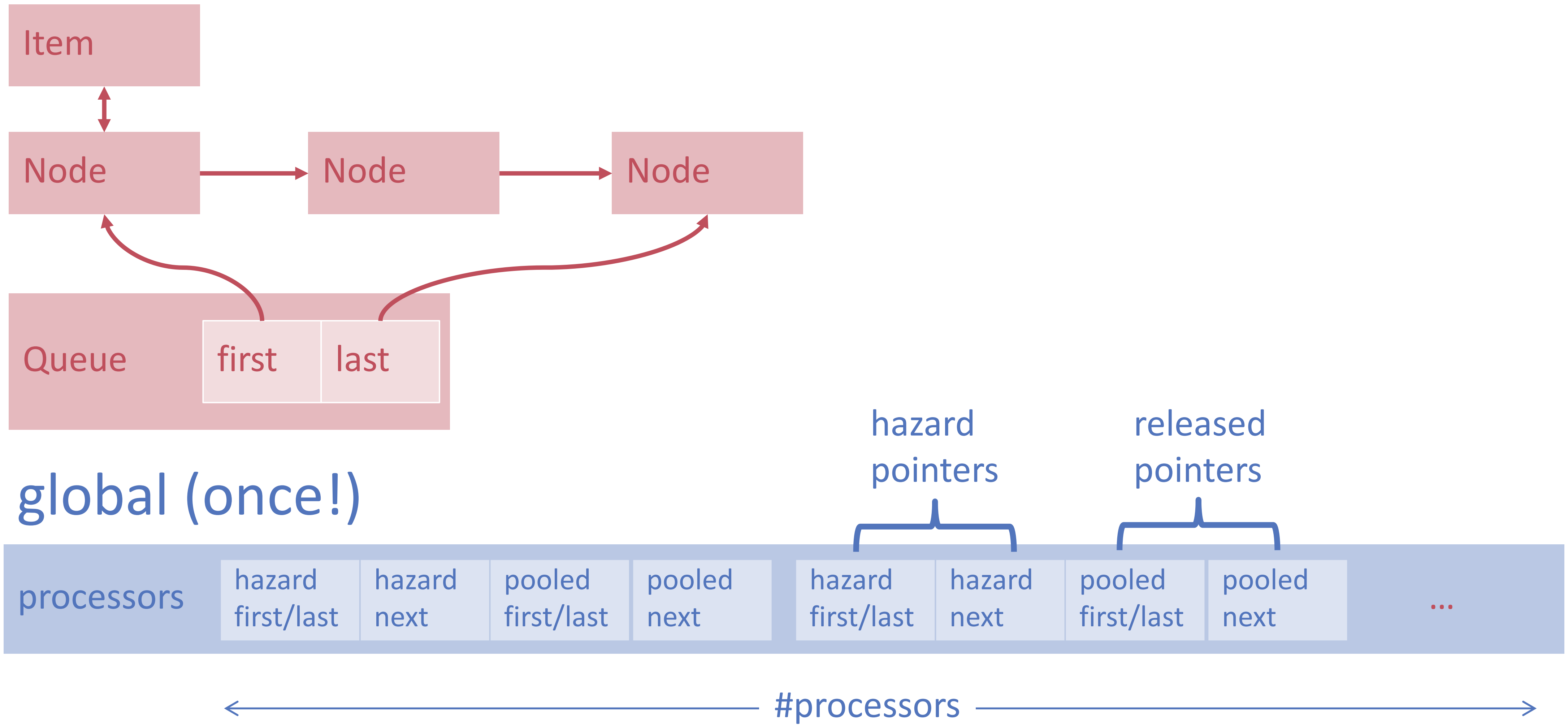
- breaks all assumptions made so far
  (number of contenders limited by the number of processors)

Key idea: model interrupt handlers as virtual processors

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

Queue Data Structures

for each queue



global (once!)

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);                                                reuse
END;
node.next := NIL; node.item := item;

LOOP
   last := CAS (queue.last, NIL, NIL);
   Access (last, queue.last, Last);                mark last hazarduous
   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);                                             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

Calls finalizer of previous thread

Task Switch Finalizer

```
PROCEDURE FinalizeSwitch-;
VAR currentActivity {UNTRACED}: Activity;
BEGIN {UNCOOPERATIVE, UNCHECKED}
  currentActivity := SYSTEM.GetActivity ()(Activity);
  IF currentActivity.finalizer # NIL THEN
    currentActivity.finalizer (currentActivity.previous, currentActivity.argument)
  END;
  currentActivity.finalizer := NIL;
  currentActivity.previous := NIL;
END FinalizeSwitch;
```

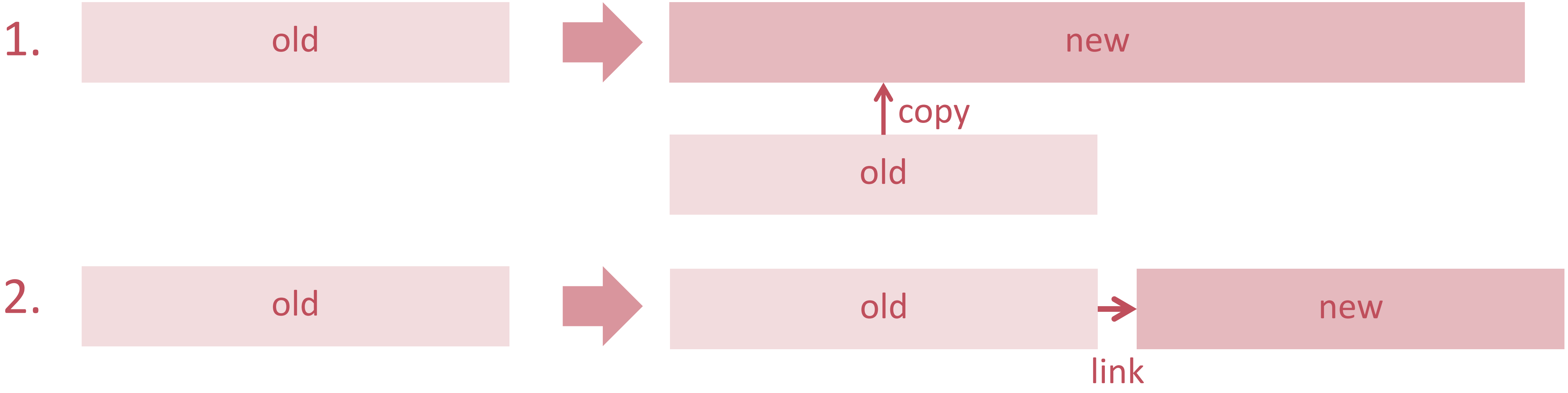
Enqueue!

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

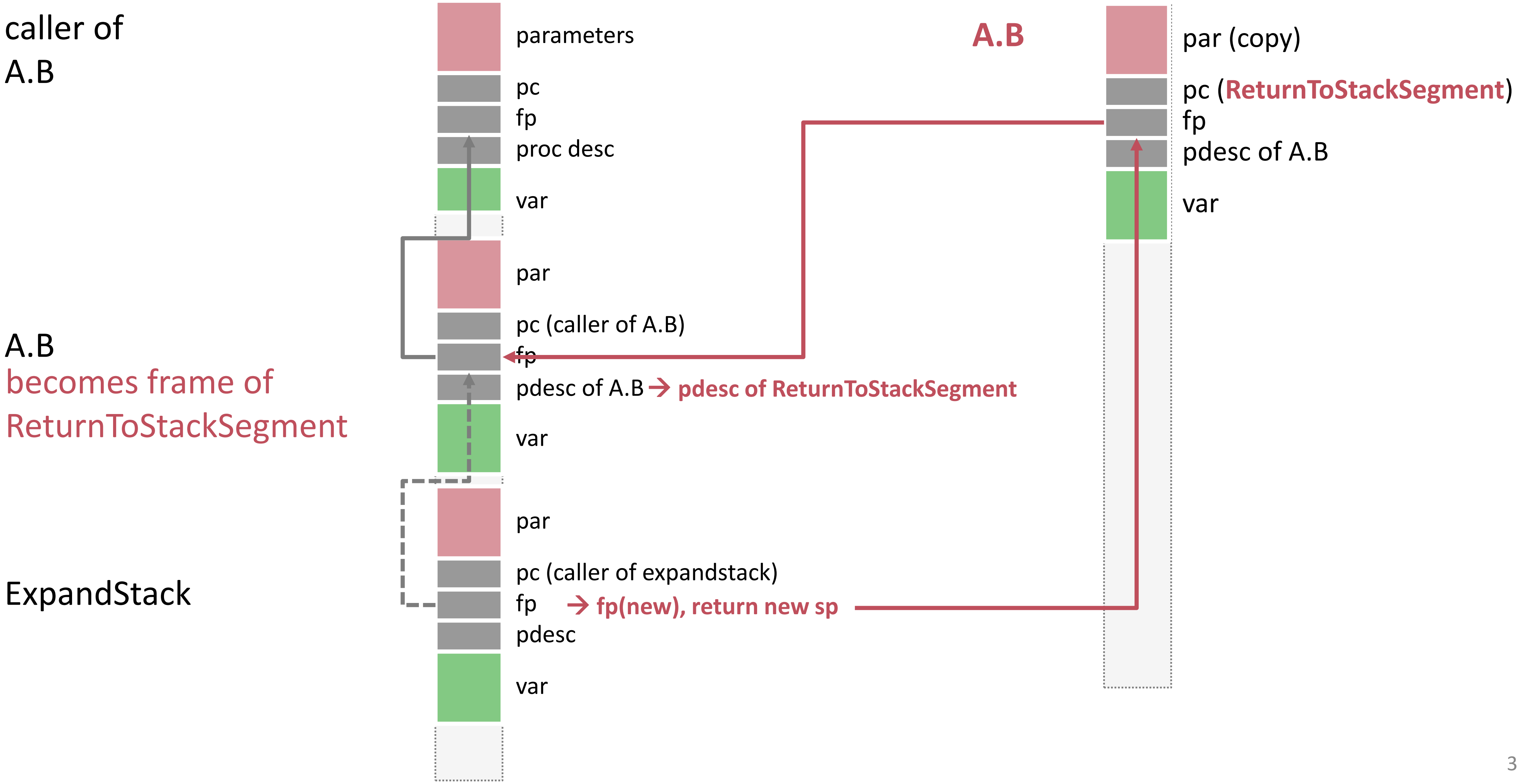
- call-by-reference parameters
  - open arrays
  - records
- unsafe pointer on stack
  - e.g. file buffers

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



Interrupts

First level IRQ handler registration must be made available by non-portable CPU module

```
previous := CPU.InstallInterrupt- (handler, index);
```

Second level IRQ handling with activities: Wait for interrupt

```
Interrupts.Await(interrupt);
```

First level IRQ code affecting scheduler queues runs on a virtual processor

```
PROCEDURE Handle (index: SIZE);
BEGIN {UNCOOPERATIVE, UNCHECKED}
   IF previousHandlers[index] # NIL THEN previousHandlers[index] (index) END;

   Activities.CallVirtual(NotifyNext,
                              ADDRESS OF awaitingQueues[index],processors[index]);
END Handle;
```

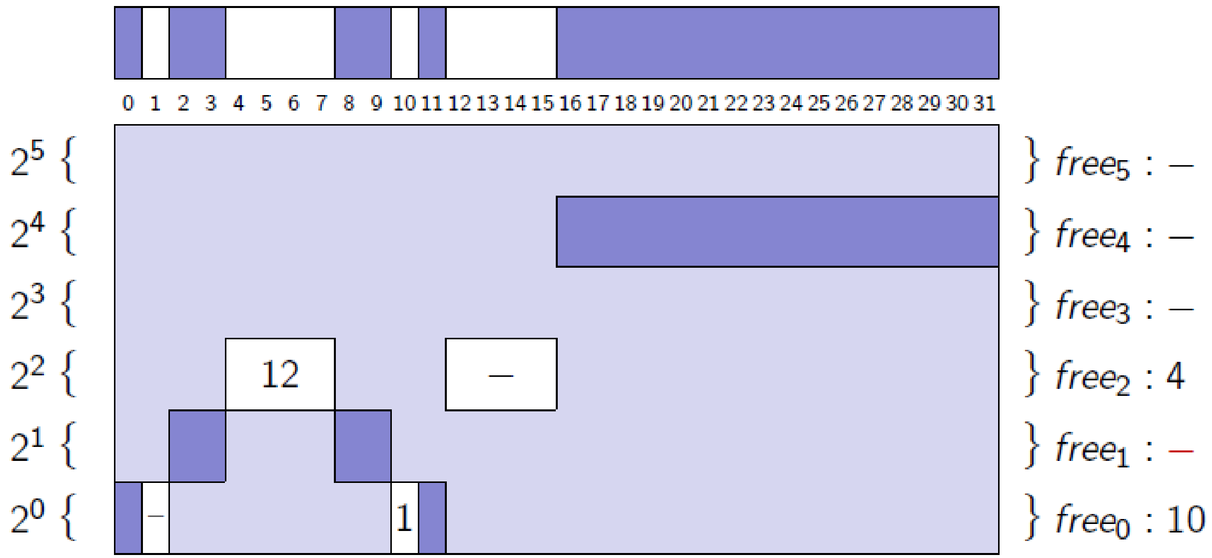
Example: Sleep on Timer Interrupt

```
PROCEDURE Sleep- (milliseconds: LONGINT);
VAR interrupt: Interrupts.Interrupt;
BEGIN {UNCOOPERATIVE, UNCHECKED}
   IF CAS (timerInterruptInstalled, 0, 1) = 0 THEN
      (* setup timer irq on hardware *)
   END;
   Interrupts.Install (interrupt, CPU.IRQ); INC (milliseconds, clock);
   WHILE clock - milliseconds < 0 DO Interrupts.Await (interrupt) END;
END Sleep;

PROCEDURE HandleTimer (index: SIZE);
BEGIN {UNCOOPERATIVE, UNCHECKED}
   IF previousTimerHandler # NIL THEN previousTimerHandler (index) END;
   IF 1 IN CPU.ReadMask (CPU.STCS) THEN
      (* re-enable timer irq on hardware *)
   END;
END HandleTimer;
```

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

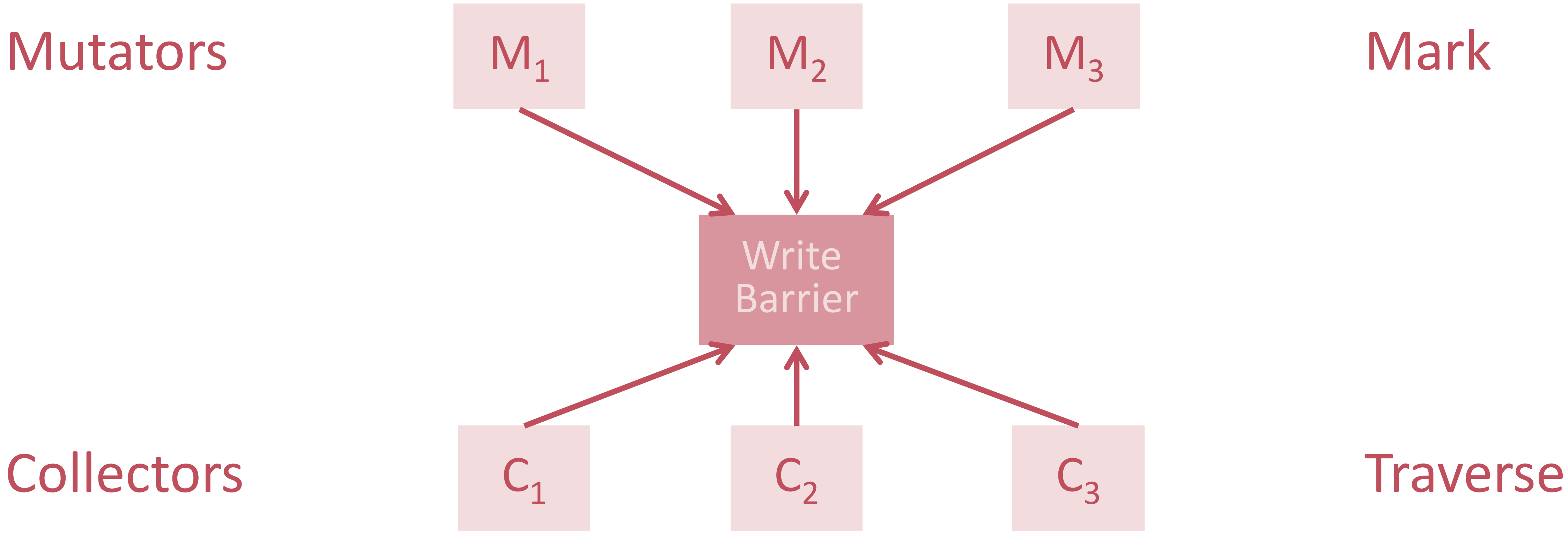


Lock-free Garbage Collector

- 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

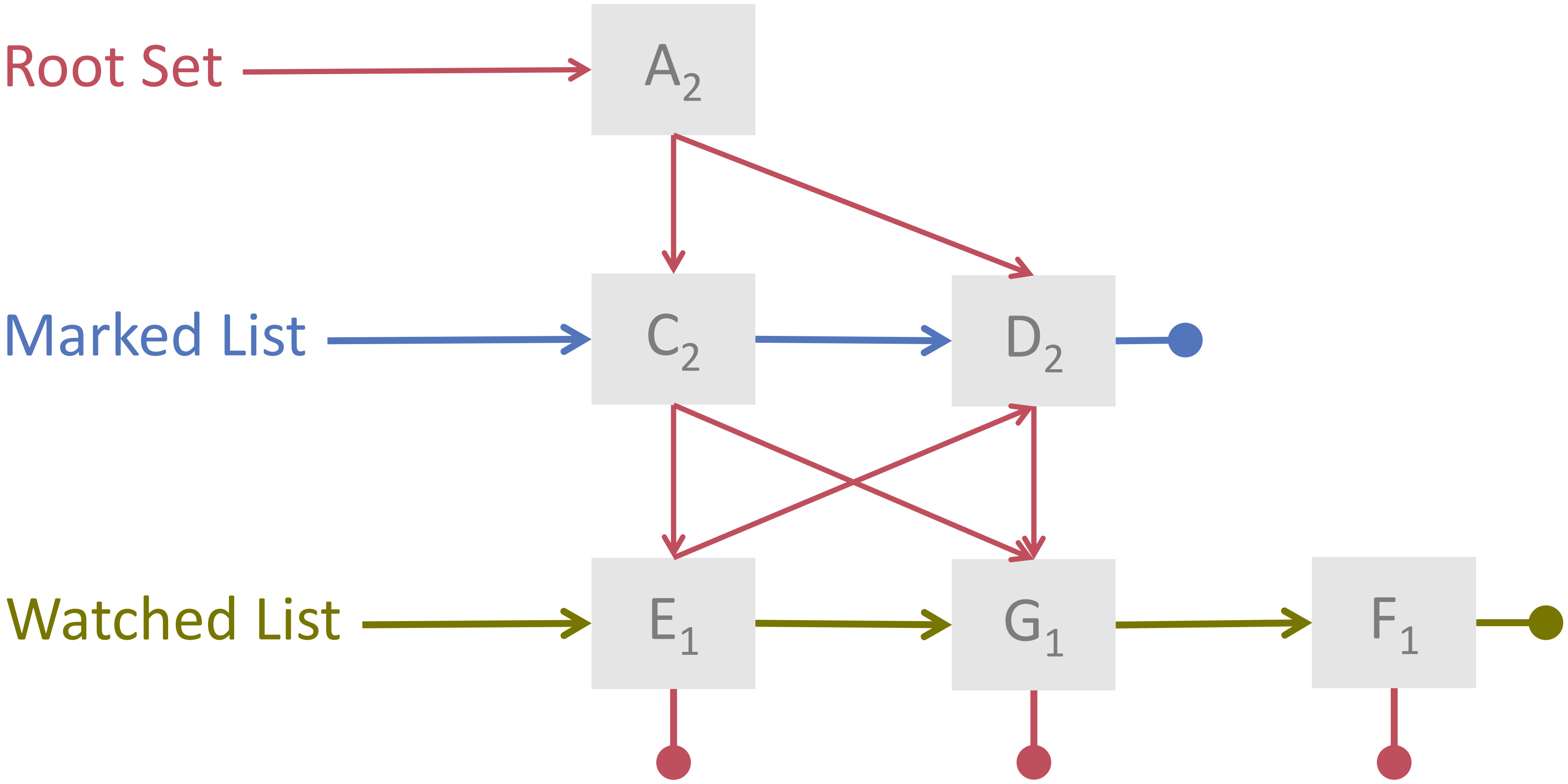
Traverse

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

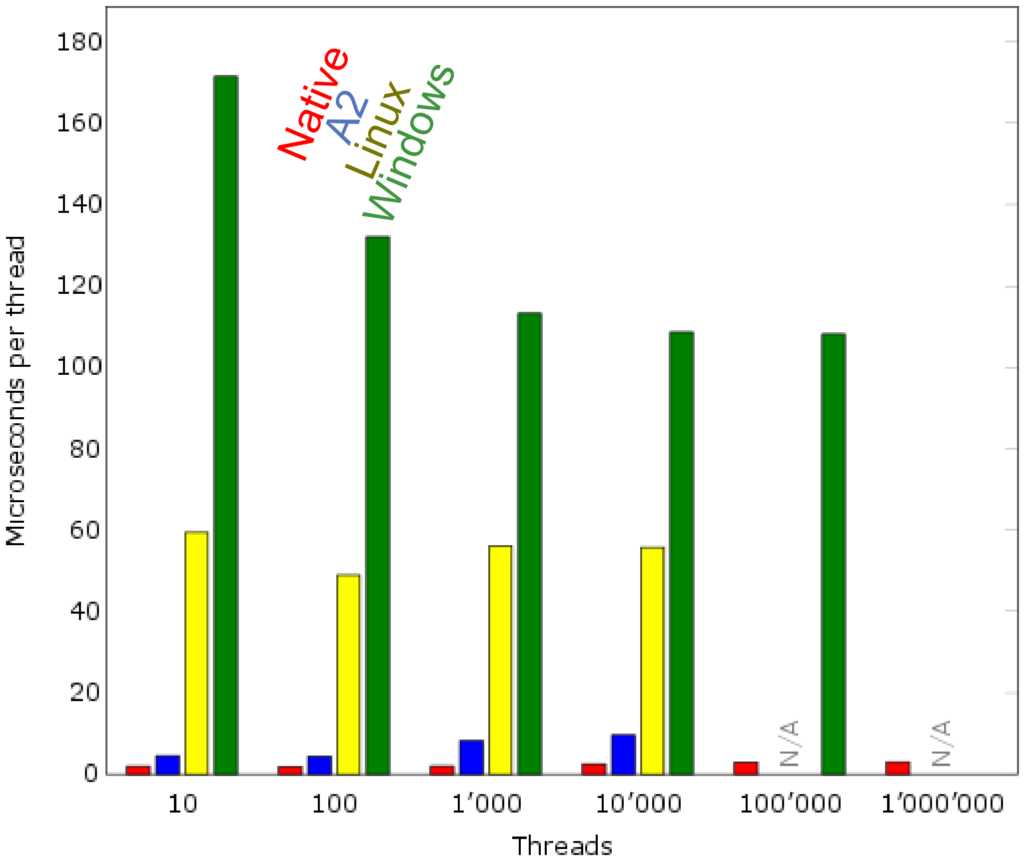
- No timer interrupt required → scheduler hardware independent
- No virtual memory → no separate address spaces → everything runs in user mode, all the time
- Hardware-dependent functions (CAS) are pushed into the language
- "Almost":
  we need a **minimal** stub written in assembly code to initialize memory mappings and initialize all processors

How well does it perform? (Simplicity, Portability)

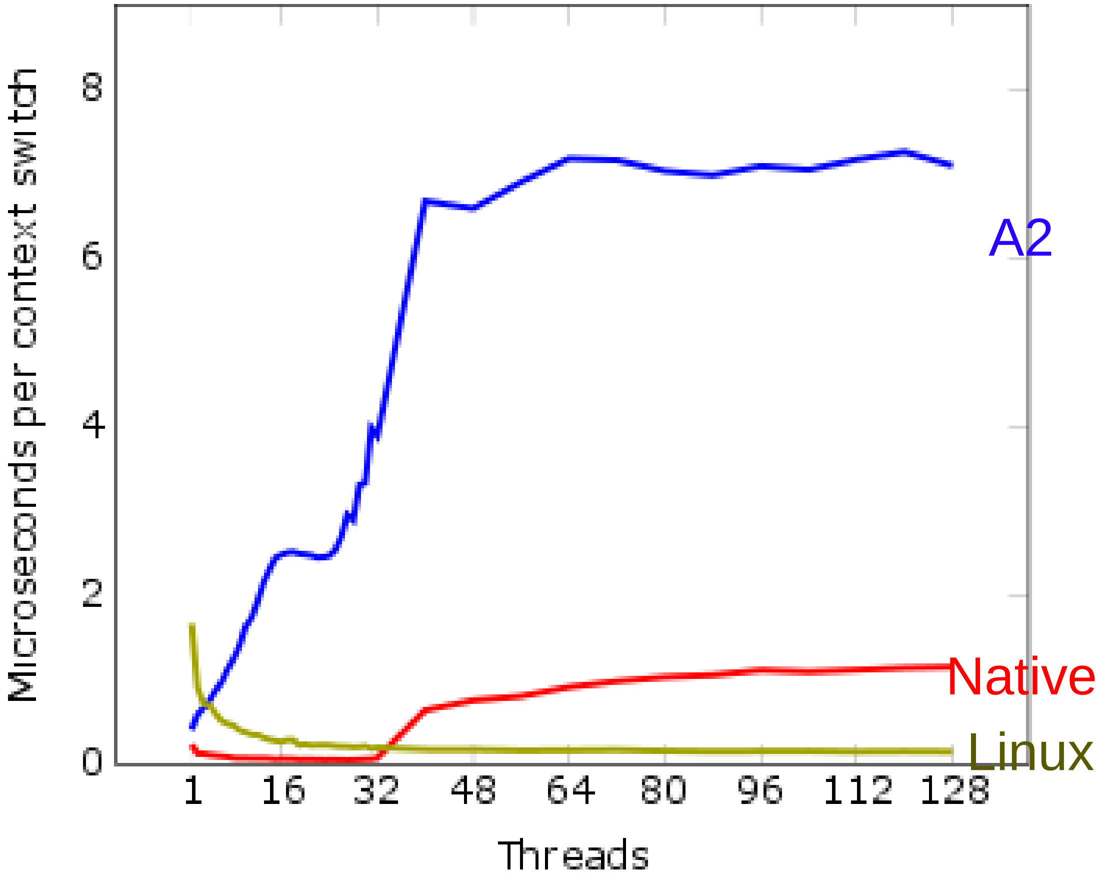
Component	**Lines of Code (Kernel)**
Interrupt Handling	301
Memory Management (including GC!)	352
Modules	82
Multiprocessing	213
Runtime Support	250
Scheduler	540
Total	1738 (28% of A2 orig)

How well does it perform? (Scheduler)

thread creation time

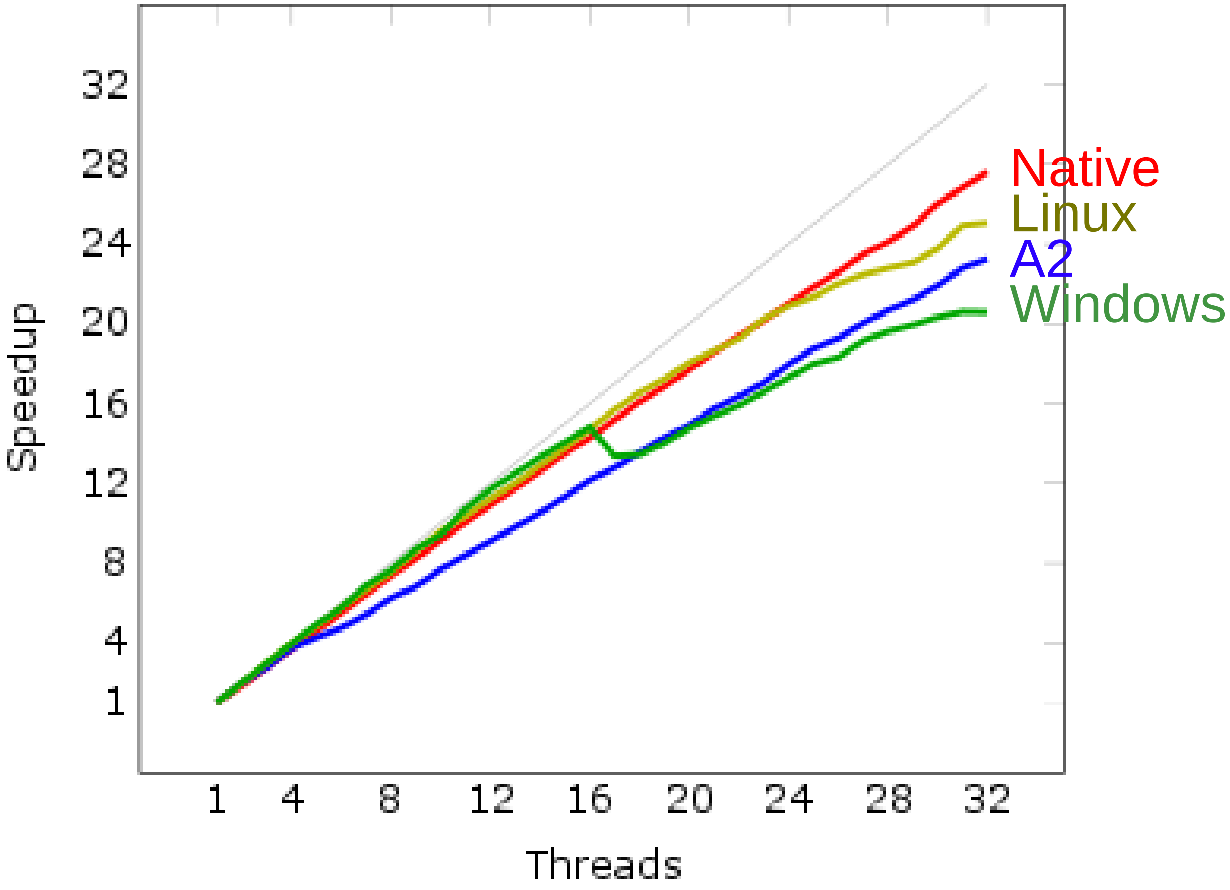


thread switching time

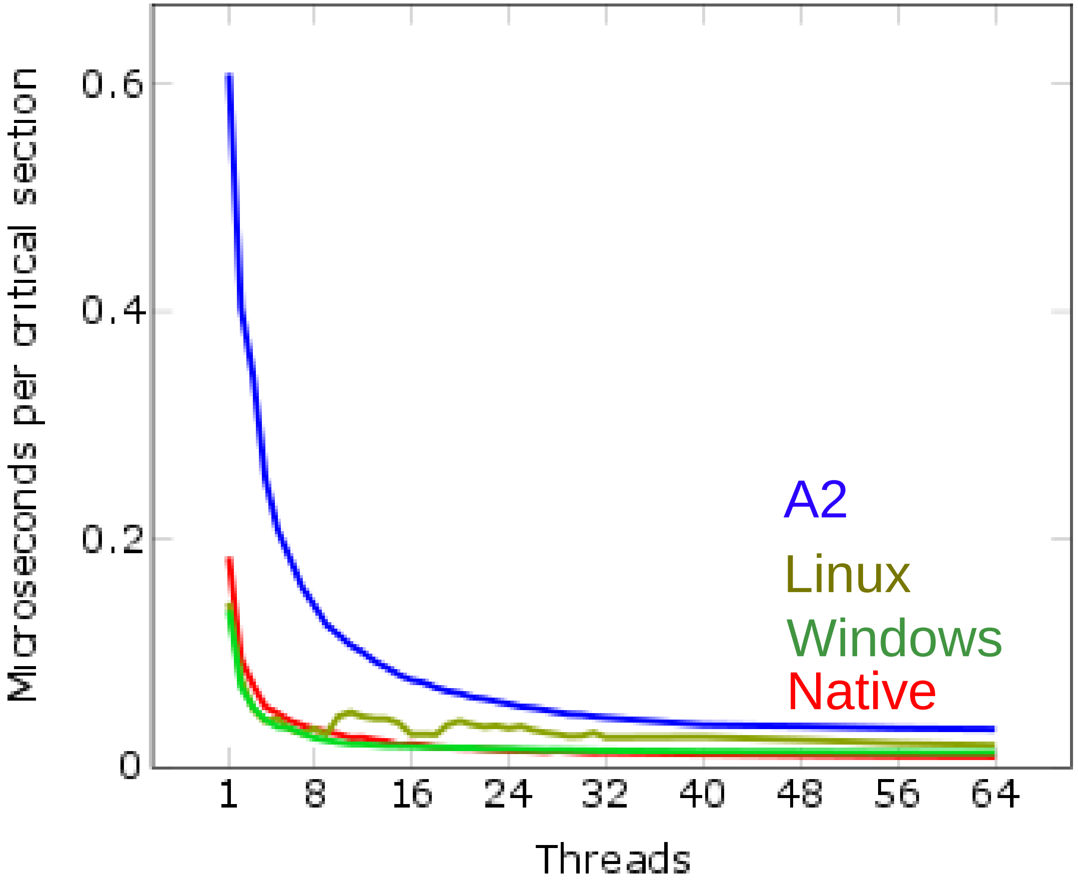


How well does it perform? (Scheduler)

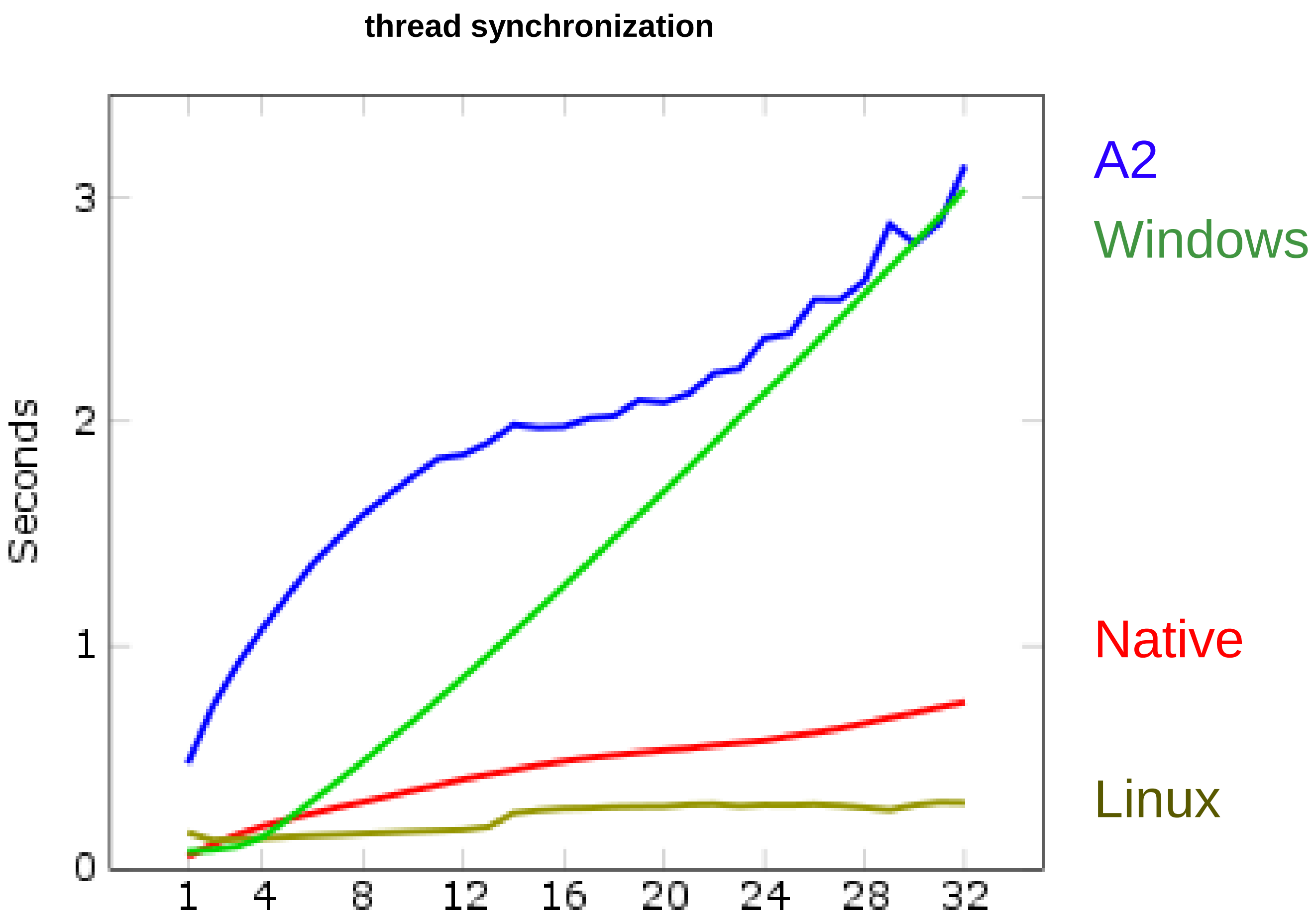
application speedup (matrix multiplication) in the presence of locks



average cost of locking operations

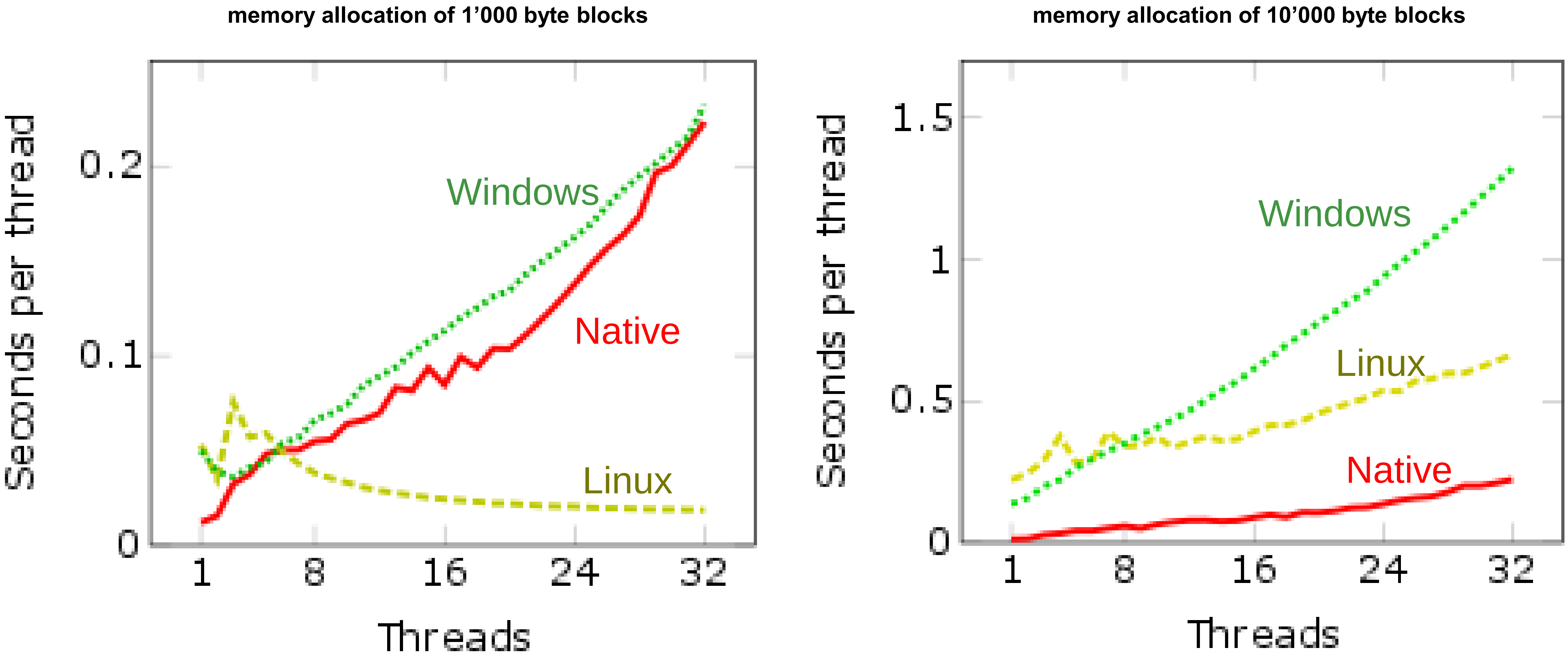


How well does it perform? (Scheduler)

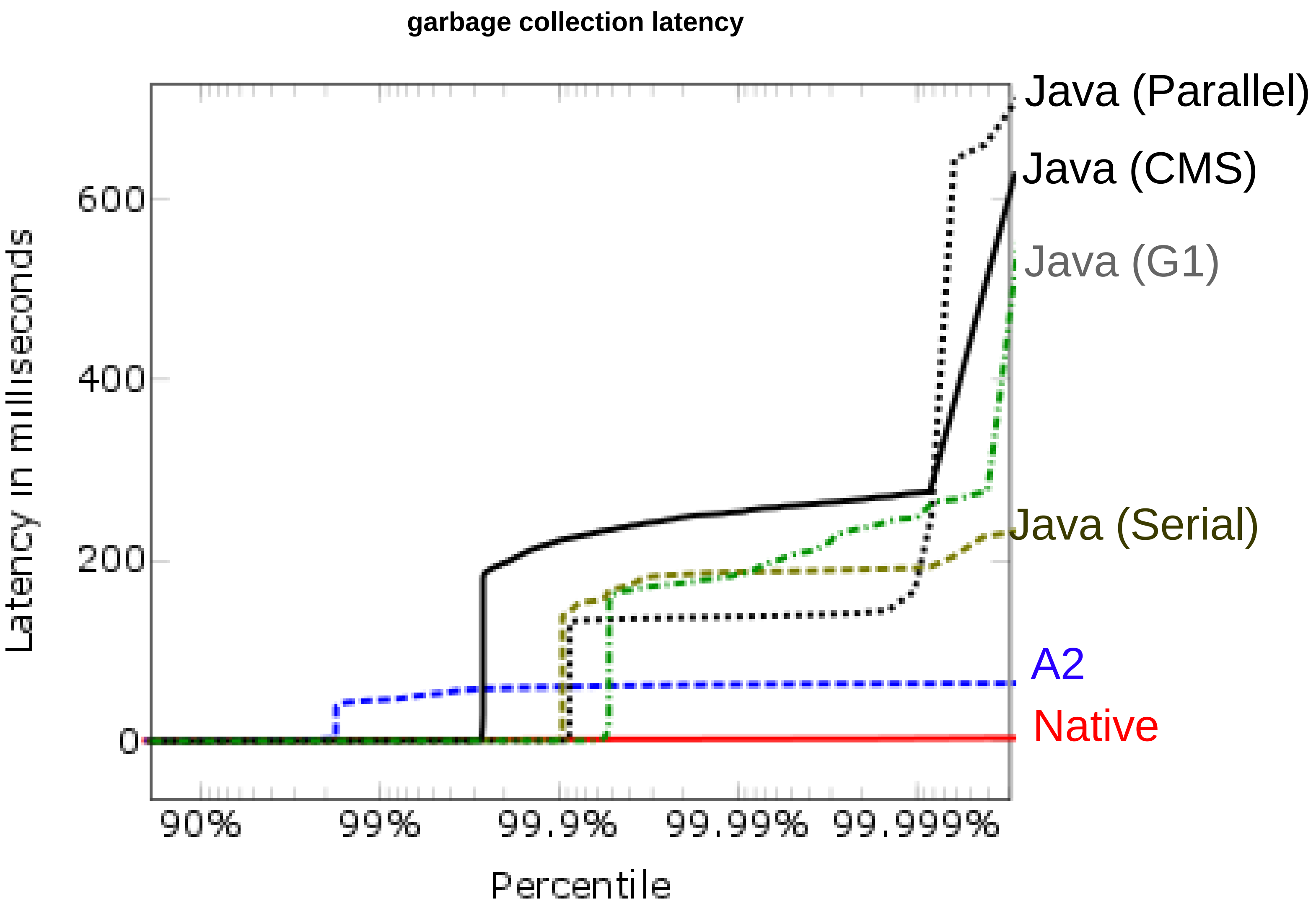


thread synchronization

How well does it perform? (Memory Manager)



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