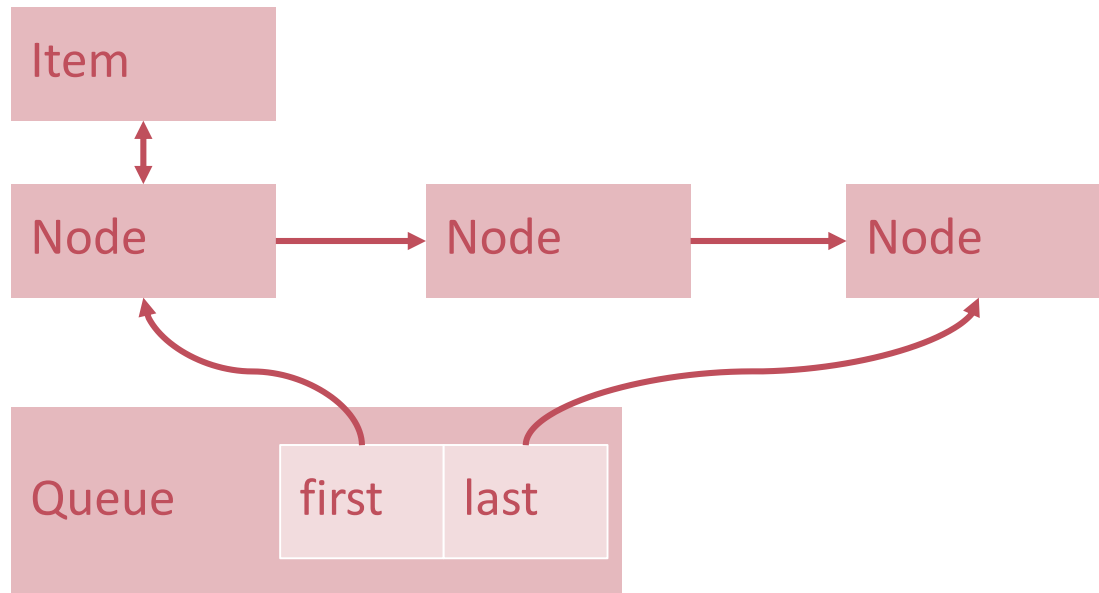
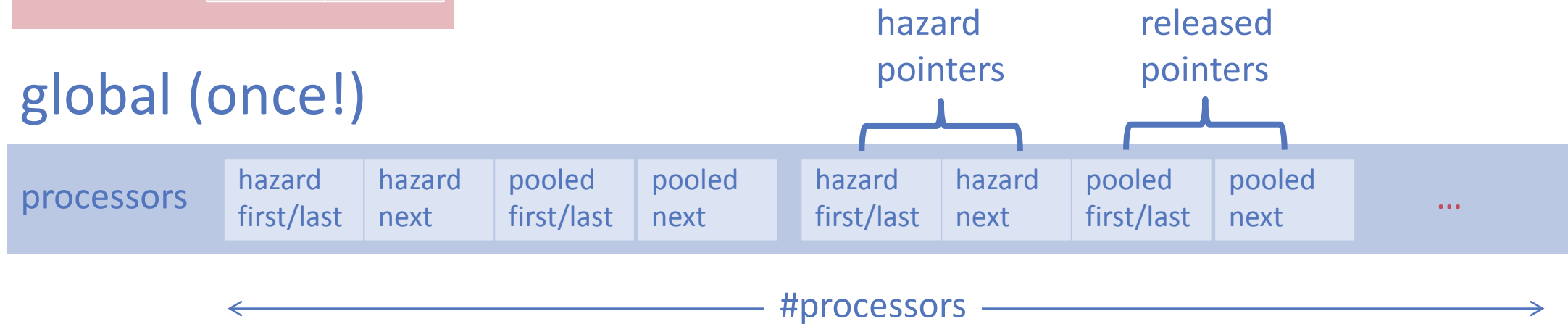


Queue Data Structures

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;
```

guarantee: no change to reference
after node was set hazardous

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

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;
```

reuse

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);
```

mark last hazardous

unmark last

Lock-Free Dequeue with Node Reuse

LOOP

```
first := CAS (queue.first, NIL, NIL);
```

```
Access (first, queue.first, First);
```

mark first hazardous

```
next := CAS (first.next, NIL, NIL);
```

```
Access (next, first.next, Next);
```

mark next hazardous

```
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
```

accessed via
activity register



```
END Activity;
```

```
(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

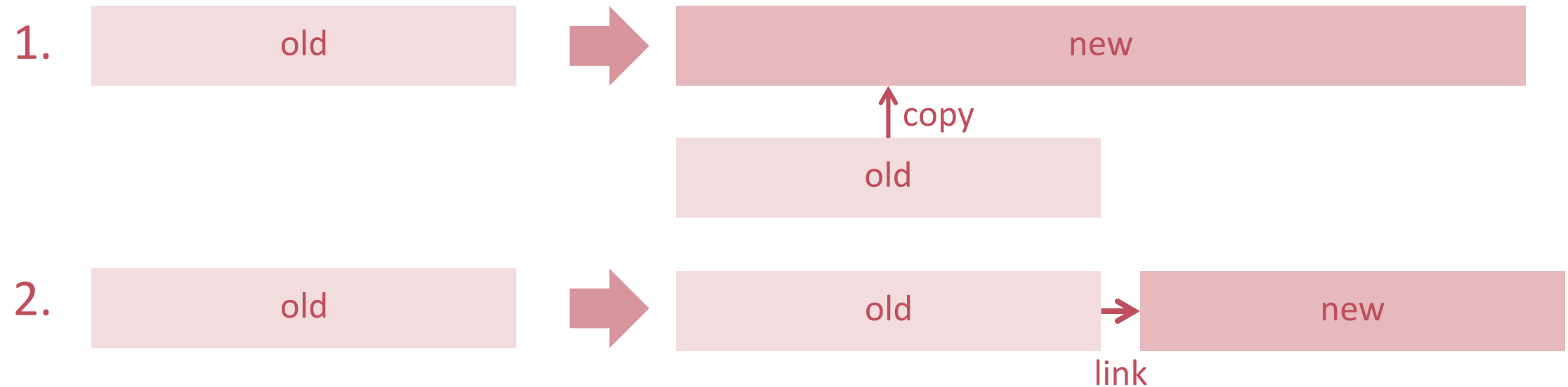
Calls finalizer of
previous thread

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

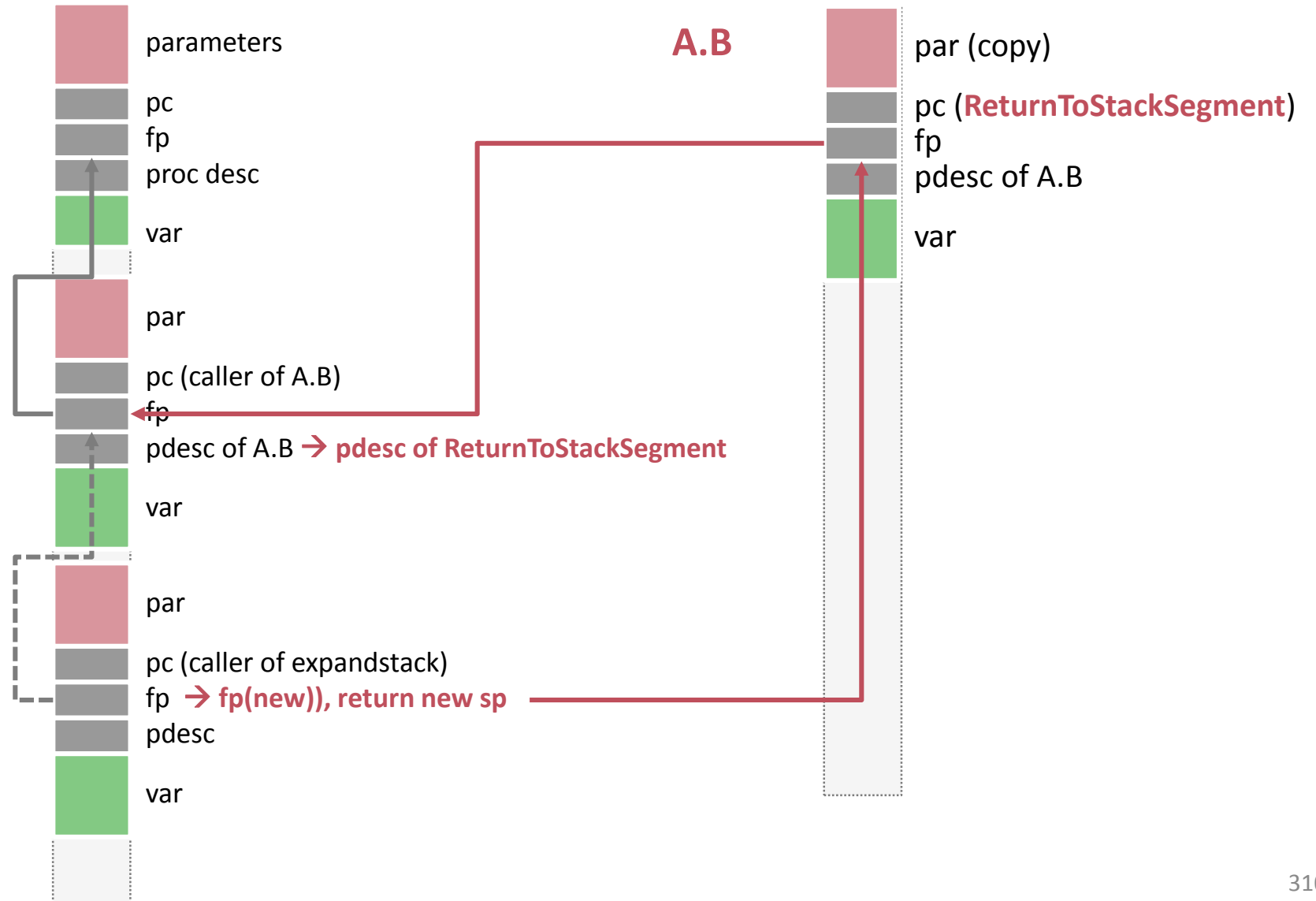
- Instrumented call to ExpandStack
- End of current stack segment pointer included in process descriptor
- Link stacks on demand with new stack segment
- Return from stack segment inserted into call chain backlinks

Linked Stacks

caller of
A.B

A.B
becomes frame of
ReturnToStackSegment

ExpandStack



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;
```

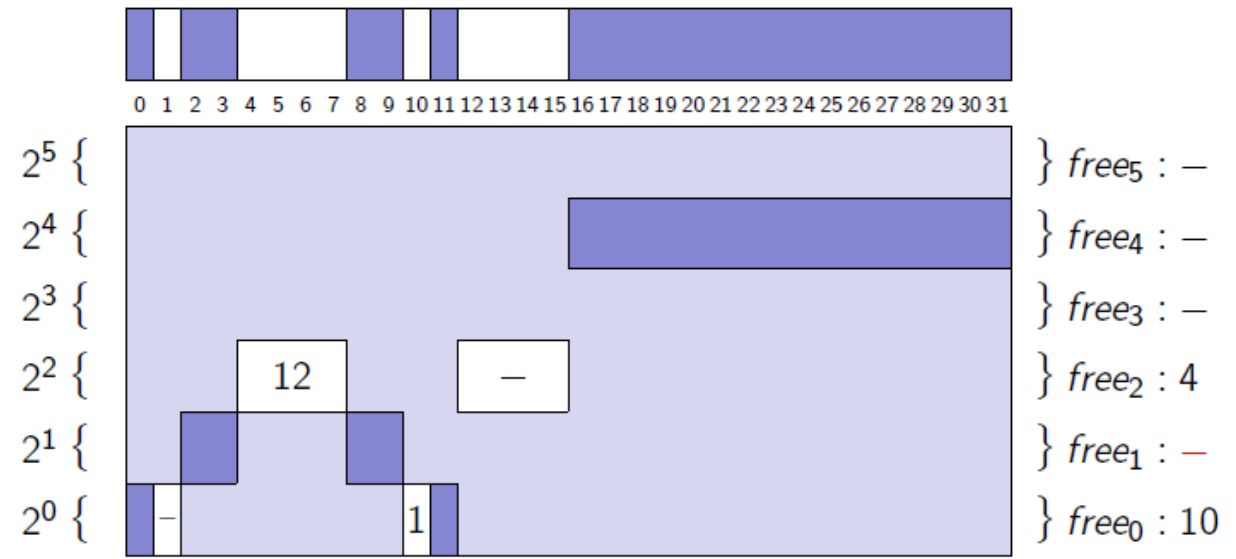
Interrupts (RPI)

```
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

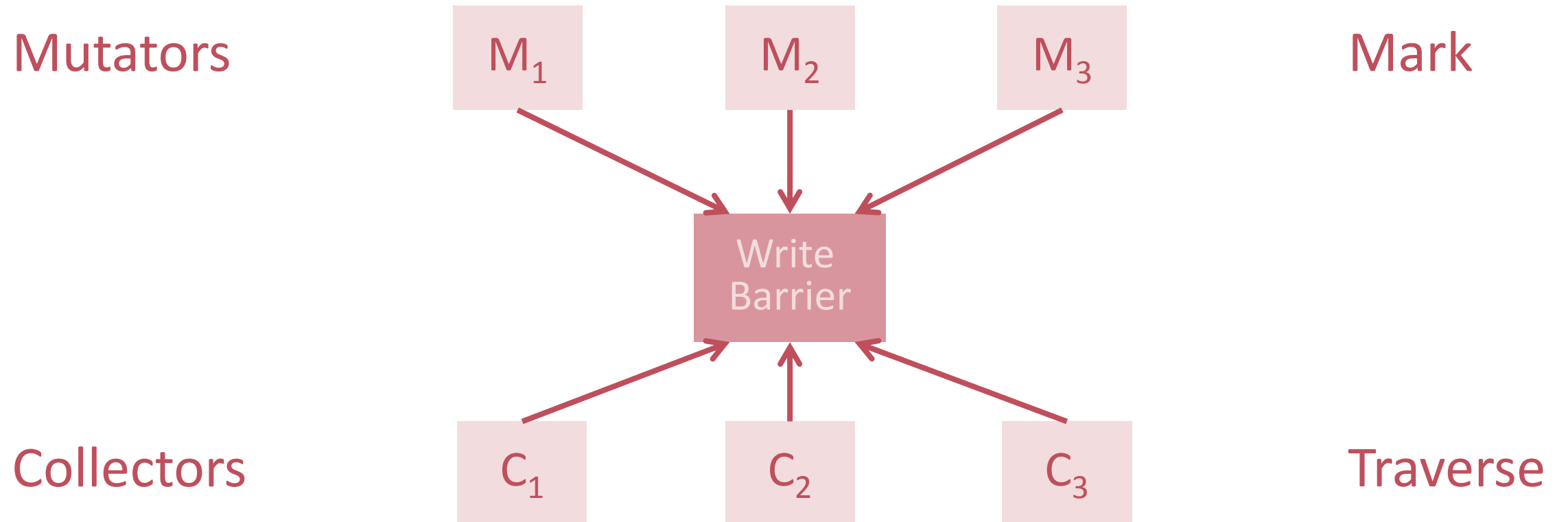
- Allocation / De-allocation implemented using only lock-free algorithms
- Buddy system with independent (lock-free) queues for the different block sizes
- Lock-free mark-sweep garbage collector
- Several garbage collectors can run in parallel



Lock-free Garbage Collector

- Mark & Sweep
- Precise
- Optional
- Incremental
- Concurrent
- Parallel

Synchronisation

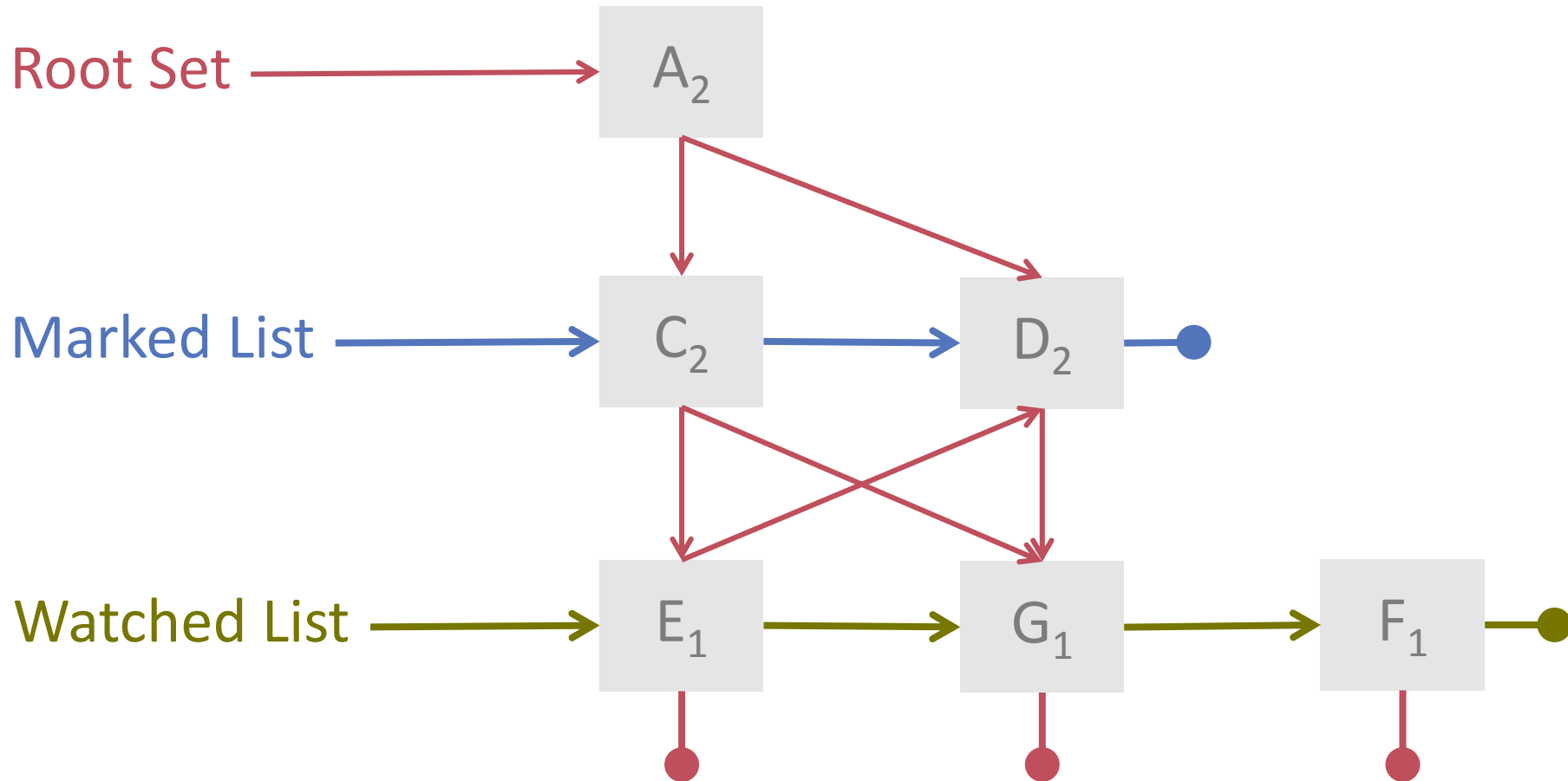


Data Structures

	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

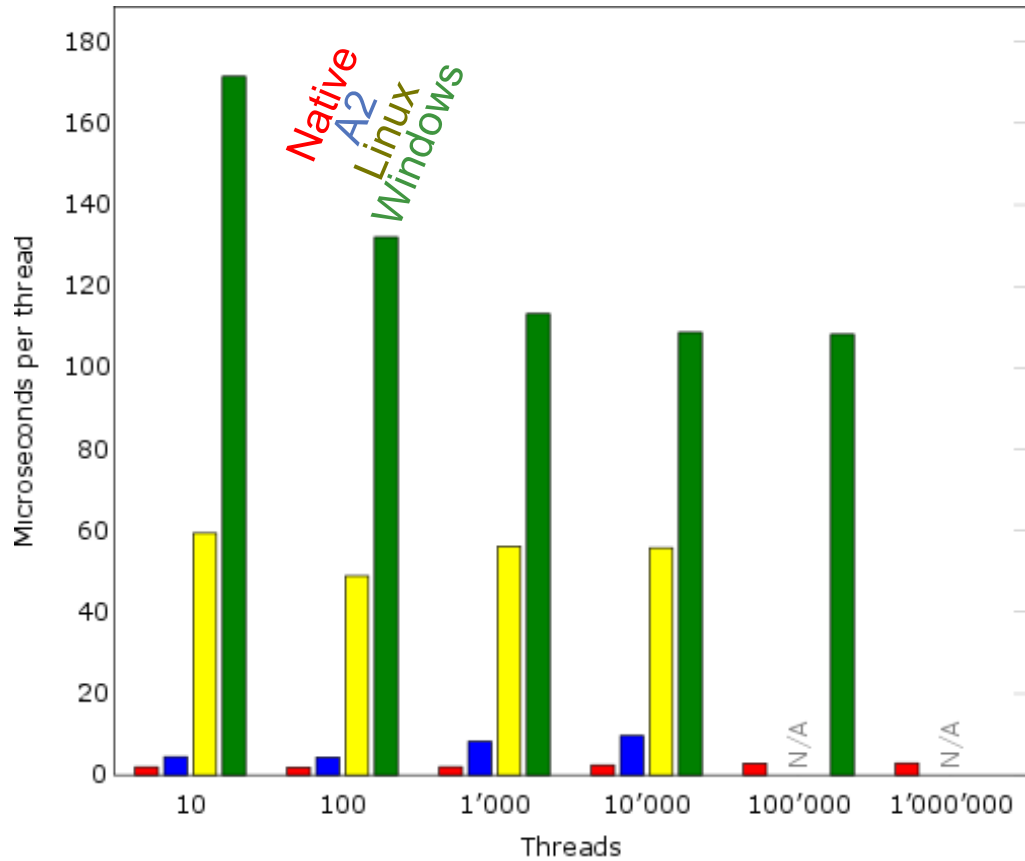
- No timer interrupt required → scheduler hardware independent
- No virtual memory → no separate address spaces → everything runs in user mode, all the time
- Hardware-dependent functions (CAS) are pushed into the language
- "Almost":
we need a **minimal** stub written in assembly code to initialize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

How well does it perform? (Simplicity, Portability)

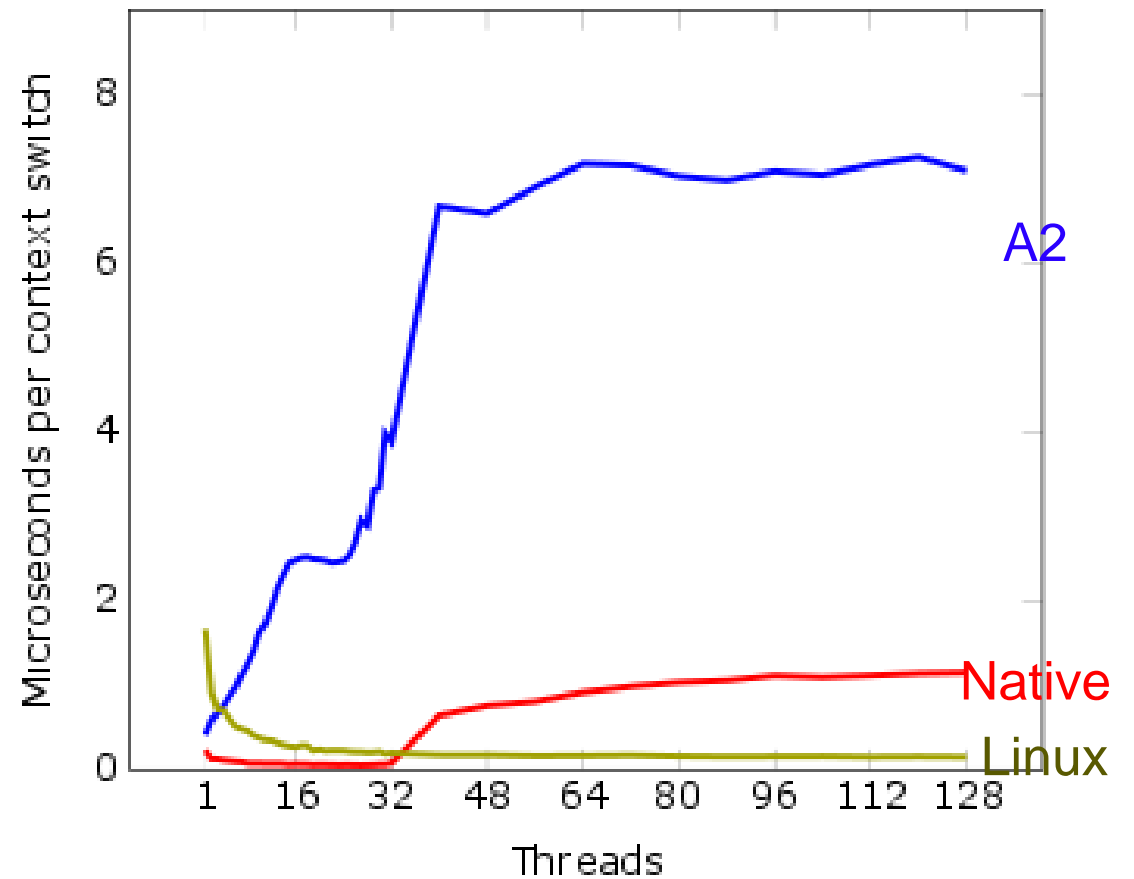
Component	Lines of Code (Kernel)
Interrupt Handling	301
Memory Management (including GC!)	352
Modules	82
Multiprocessing	213
Runtime Support	250
Scheduler	540
Total	1738 (28% of A2 orig)

How well does it perform? (Scheduler)

thread creation time

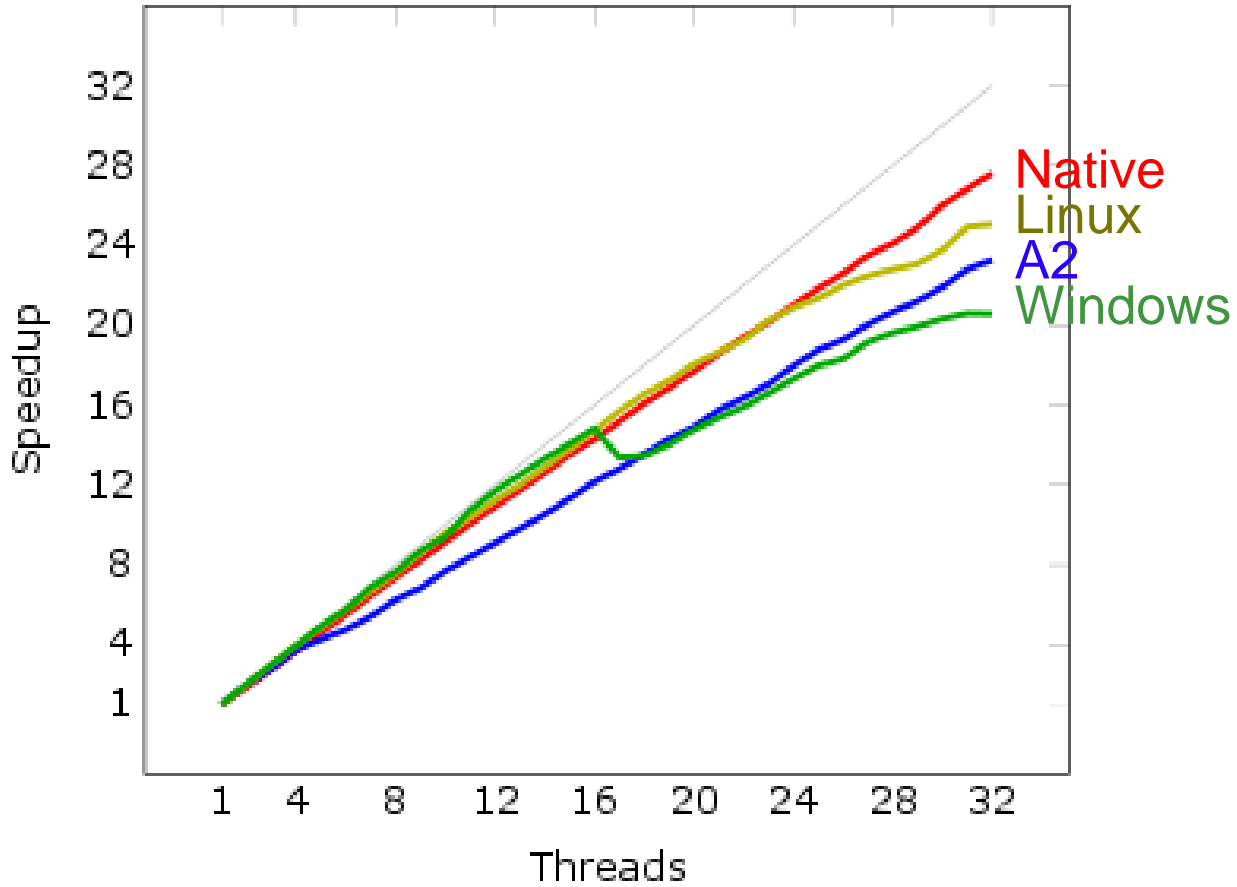


thread switching time

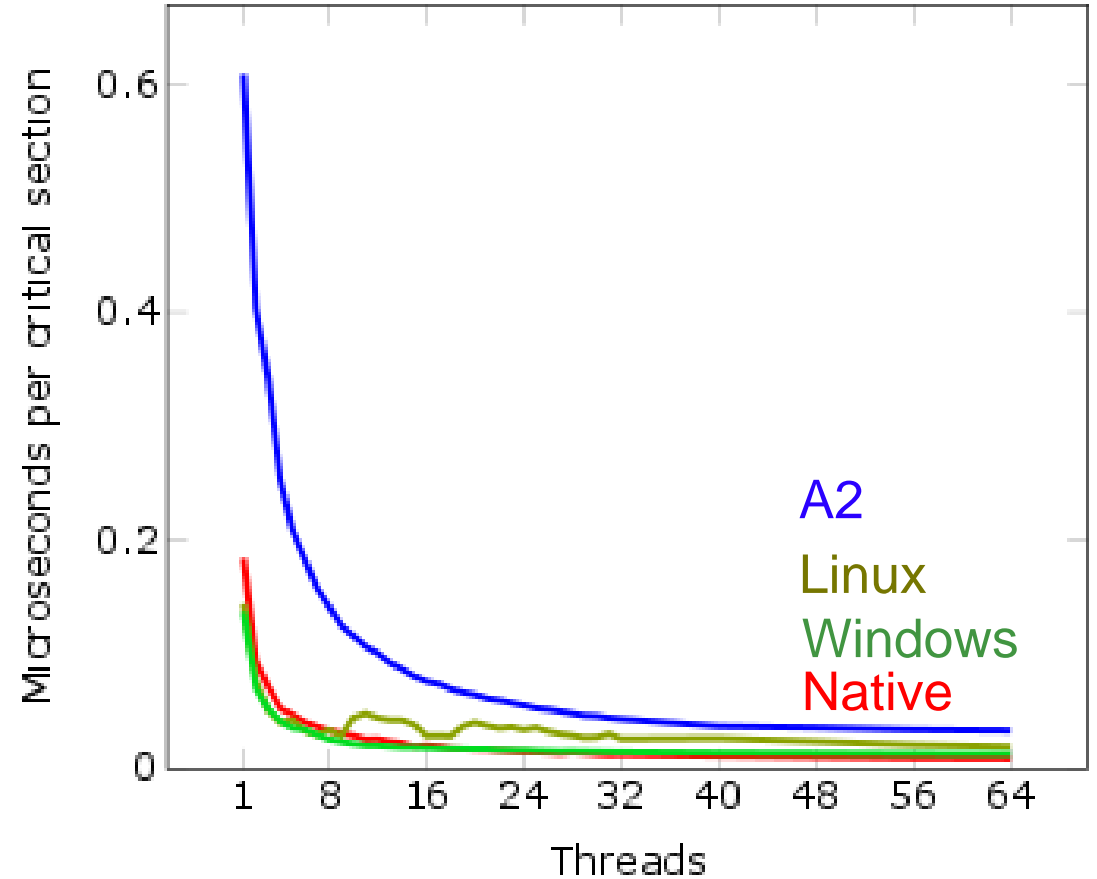


How well does it perform? (Scheduler)

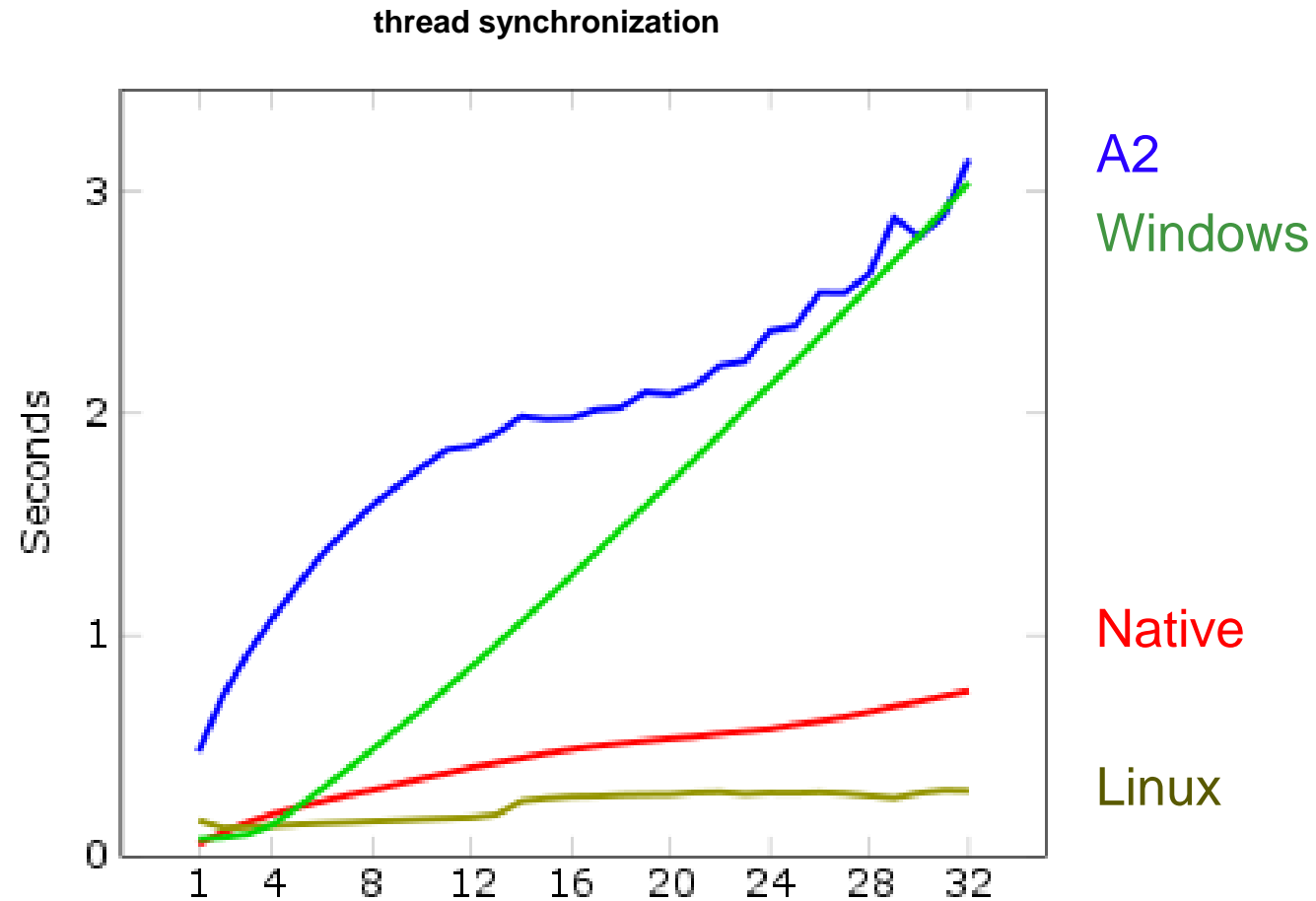
application speedup (matrix multiplication)
in the presence of locks



average cost of locking operations

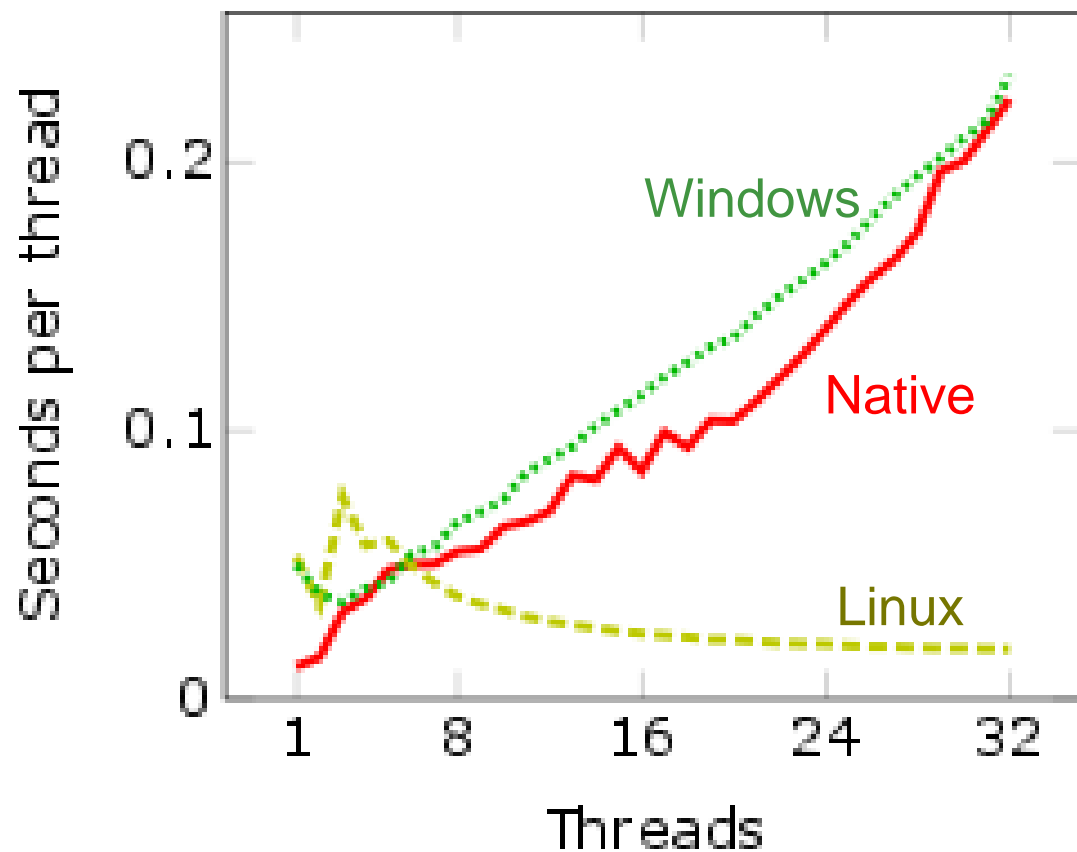


How well does it perform? (Scheduler)

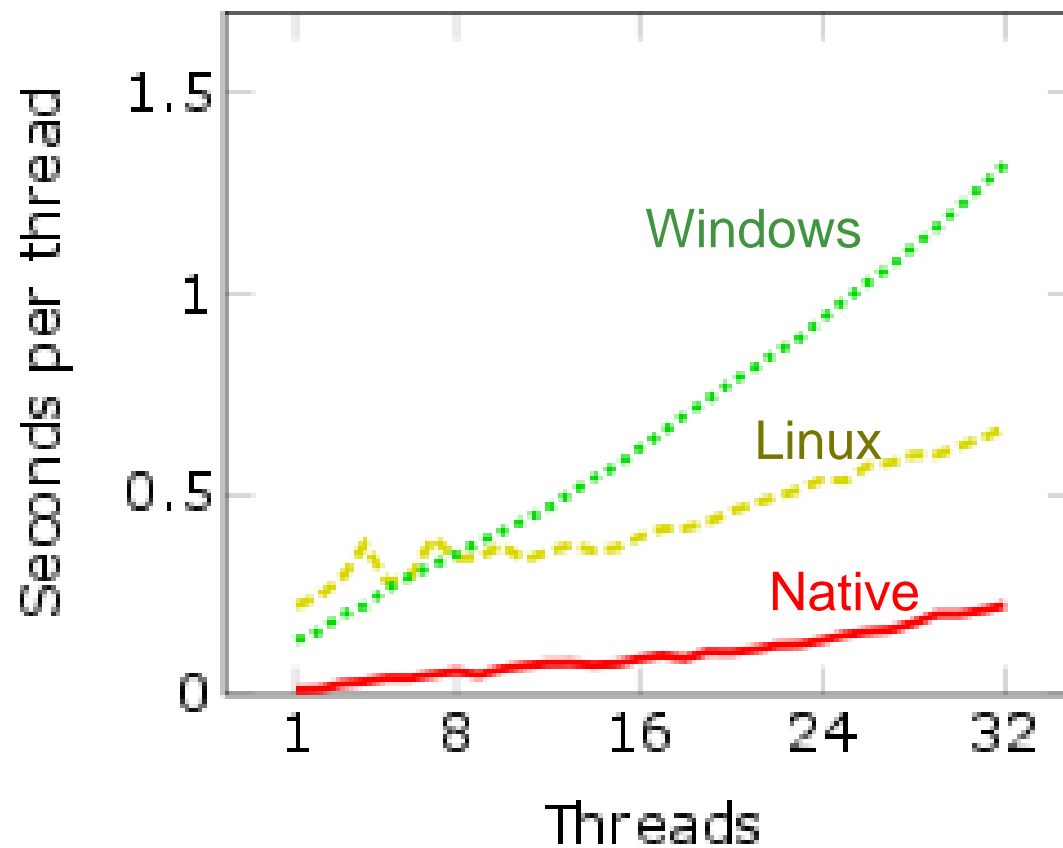


How well does it perform? (Memory Manager)

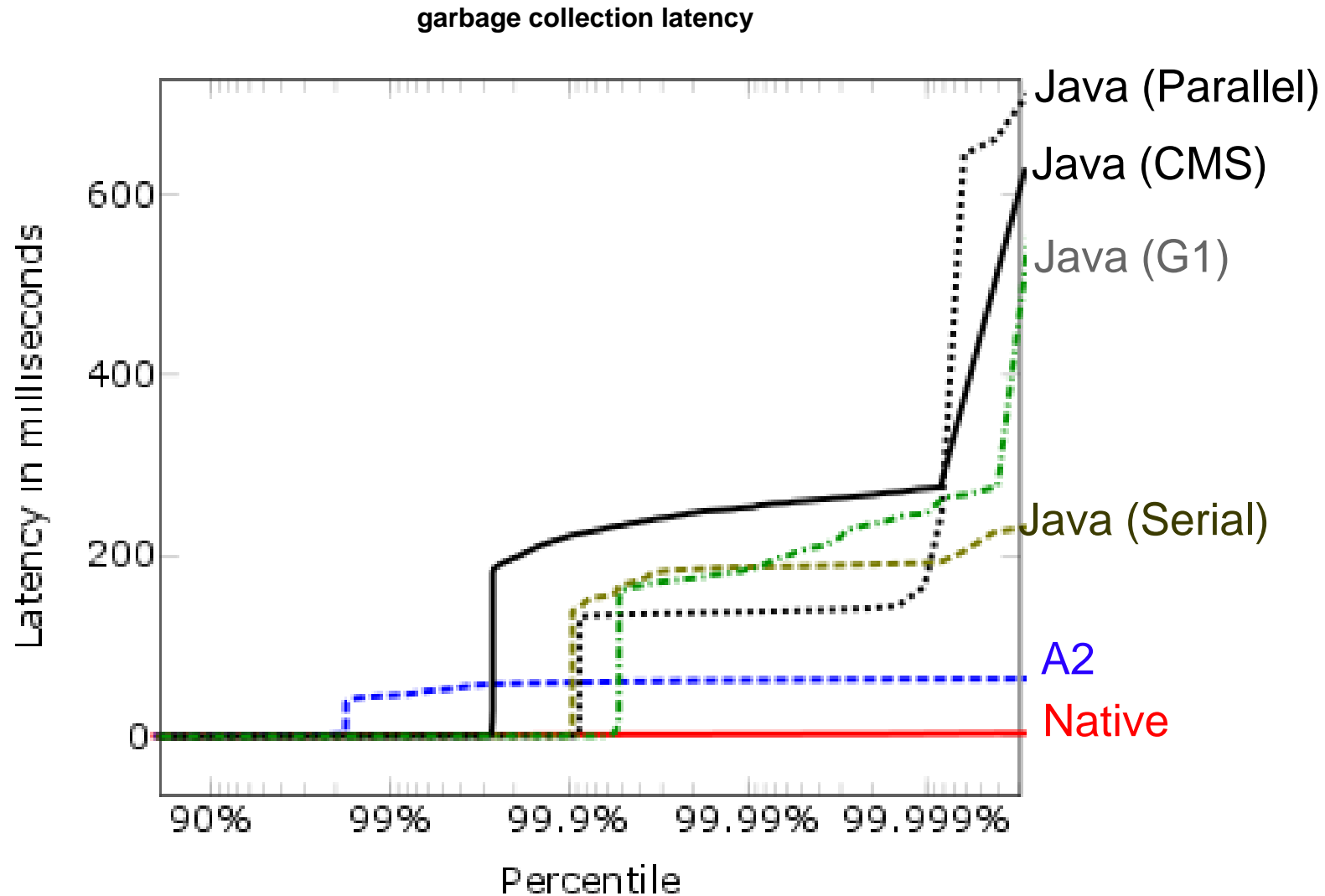
memory allocation of 1'000 byte blocks



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