#### ABA

#### **Problems of unbounded lock-free queues**

- unboundedness  $\rightarrow$  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

### 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



### 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**



# **Time Sharing**



# **Time Sharing**



#### **Cooperative Multitasking**



#### **Cooperative Multitasking**



# **Cooperative Multitasking**



#### **Ensure cooperation**

• Compiler automatically inserts code at specific points in the code

#### **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:
```

#### uncooperative

PROCEDURE Enqueue- (item: Item; VAR queue: Queue);
BEGIN {UNCOOPERATIVE}

```
(* no scheduling here ! *)
END Enqueue;
```

#### uncooperative

PROCEDURE Enqueue- (item: Item; VAR queue: Queue);
BEGIN {UNCOOPERATIVE}

```
(* no scheduling here ! *)
END Enqueue;
```

zero overhead processor local "locks"

#### 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

#### 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

### 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

#### $\rightarrow$ hazard pointers can be associated with the processor

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

### 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

#### $\rightarrow$ hazard pointers can be associated with the processor

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

thread-local storage  $\rightarrow$  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)

### 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



# 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;
                                                   guarantee: the node in reference
     value := CAS (reference, NIL, NIL);
                                                   was set hazardous before it was
     IF value = node THEN EXIT END;
                                                   here available in reference
     node := value;
  END;
END Access;
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 nonhazarduous node for reuse (if any)

```
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);
```

```
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

```
node := item.node;
IF ~Acquire (node) THEN
  NEW (node);
                                                                                    reuse
END;
node.next := NIL; node.item := item;
LOOP
  last := CAS (queue.last, NIL, NIL);
                                                                      mark last hazarduous
  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);
```

```
node := item.node;
IF ~Acquire (node) THEN
  NEW (node);
                                                                                    reuse
END;
node.next := NIL; node.item := item;
LOOP
  last := CAS (queue.last, NIL, NIL);
                                                                      mark last hazarduous
  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);
                                                                              unmark last
Discard (Last);
```

```
LOOP
```

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

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

mark first hazarduous

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

```
LOOP
  first := CAS (queue.first, NIL, NIL);
  Access (first, queue.first, First);
                                                                     mark first hazarduous
  next := CAS (first.next, NIL, NIL);
                                                                     mark next hazarduous
  Access (next, first.next, Next);
  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;
END;
first.item := NIL; first.next := first; item.node := first;
Discard (First); Discard (Next); RETURN TRUE;
```

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

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

# Scheduling -- Activities



(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)

# 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
                                                                 Enqueue runs on
   currentActivity.quantum := Quantum;
                                                                 new thread
 END;
END Switch;
                                                                 Calls finalizer of
                                                                 previous thread
(* Switch finalizer that enqueues the previous activity to the specified ready queue. *)
PROCEDURE Enqueue (previous {UNTRACED}: Activity; queue {UNTRACED}: POINTER {UNSAFE} TO Queues.Queue);
BEGIN {UNCOOPERATIVE, UNCHECKED}
         Queues.Engueue (previous, queue^);
         IF ADDRESS OF queue<sup>^</sup> = ADDRESS OF readyQueue[IdlePriority] THEN RETURN END;
         IF Counters.Read (working) < Processors.count THEN Processors.ResumeAllProcessors END;
                                                                                                         307
END Enqueue;
```

### 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;
```

### Stack Management

Stacks organized as Heap Blocks.

Stack check instrumented at beginning of each procedure.

Stack expansion possibilities

### Stack Management

Stacks organized as Heap Blocks.

Stack check instrumented at beginning of each procedure.

Stack expansion possibilities



### 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

# 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

# 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** 

Instrumented call to ExpandStack

- Instrumented call to ExpandStack
- End of current stack segment pointer included in process descriptor

- Instrumented call to ExpandStack
- End of current stack segment pointer included in process descriptor
- Link stacks on demand with new stack segment

- 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

caller of A.B



caller of A.B



A.B













#### Lock-free Garbage Collector

- Mark & Sweep
  - mark counter, sweeping blocks
- Precise
  - GC knows all pointers, no heuristics
- Optional
  - system can be built without GC

#### Incremental

- several instances of the GC traverse parts of the heap
- Concurrent
  - GC runs in cocurrently with mutator thread
- Parallel
  - Several instances of the GCs can run concurrently

#### Synchronisation



#### **Data Structures**

	Global		Per Object	Ш
				Ш
Mark Bit	Cycle Count		Cycle Count	11
				11
Marklist	Marked First		Next Marked	Ш
				11
Watchlist	Watched First		Next Watched	Ш
				11
Root Set	Global References		Local Refcount	11

#### Example

Cycle Count = 2



#### Lock-Free Runtime Conclusion

- Consequent use of lock-free algorithms in the kernel
- Oberon Synchronization primitives (for applications) implemented on top
- Efficient unbounded lock-free queues, ABA Problem solved using Hazard Pointers
- Implicit cooperative multitasking -> can switch off scheduling locally (uncooperative blocks)
- Parallel and lock-free memory management with garbage collection