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
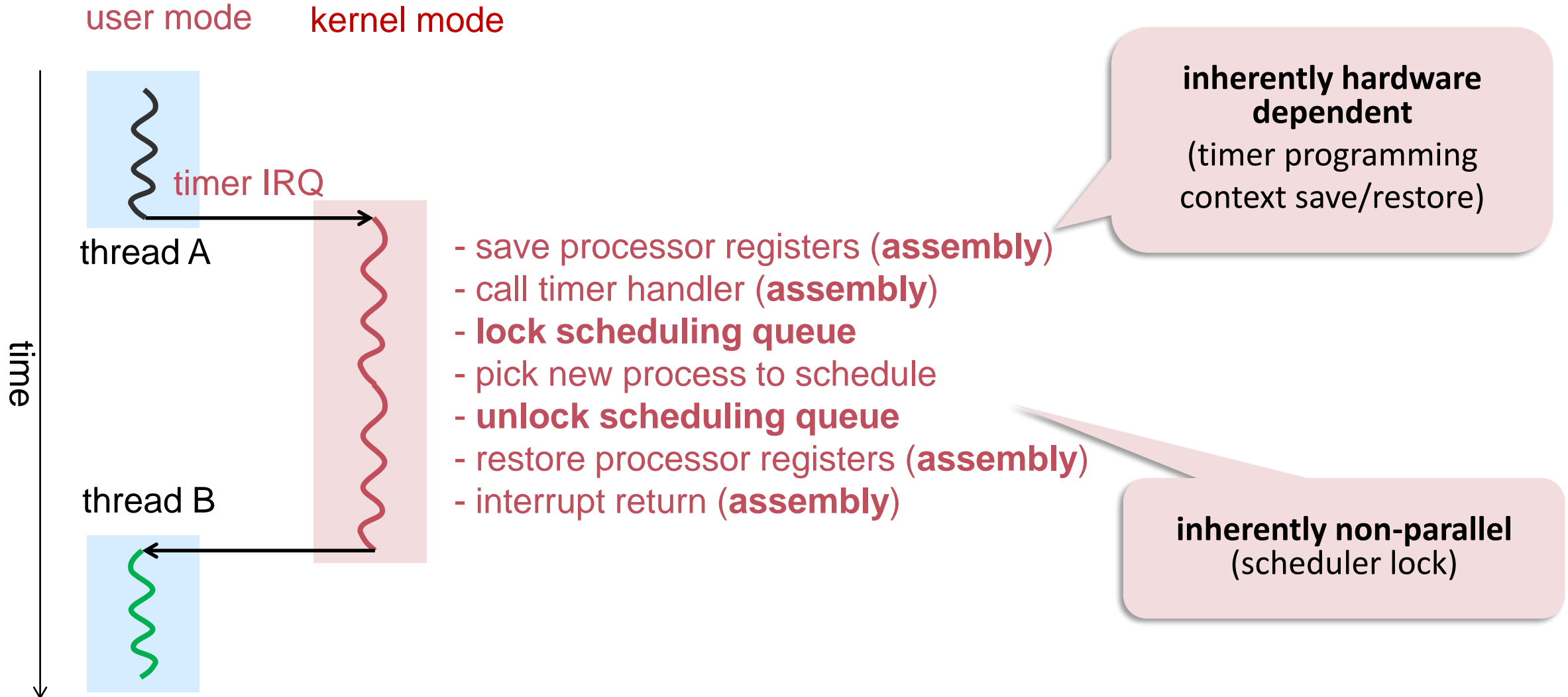
Number of threads unbounded

→time to check hazard pointers also unbounded!

→difficult dynamic bookkeeping!

thread A
- hp1
- hp2

thread B
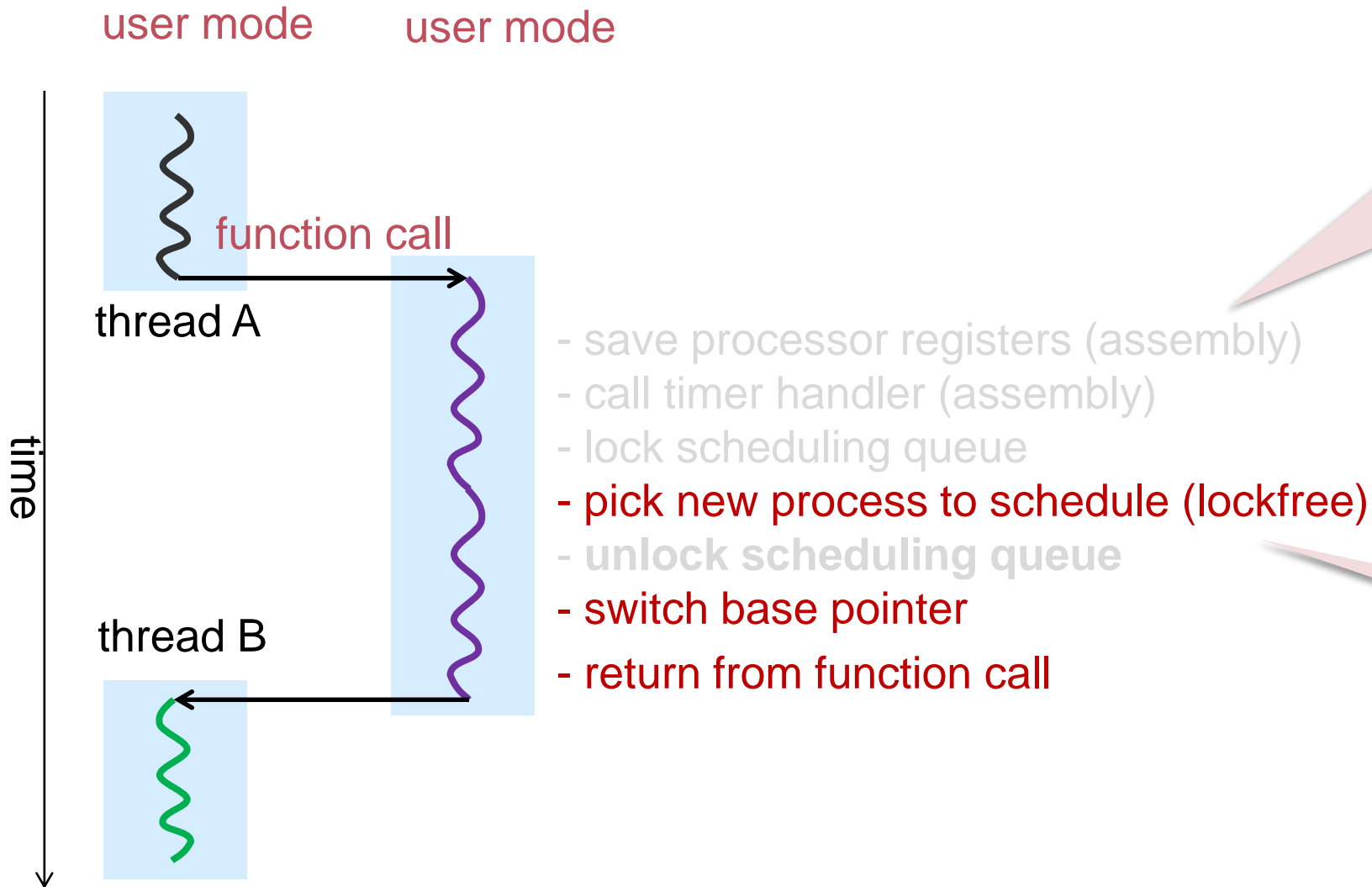- hp1
- hp2

thread C
- hp1
- hp2

...
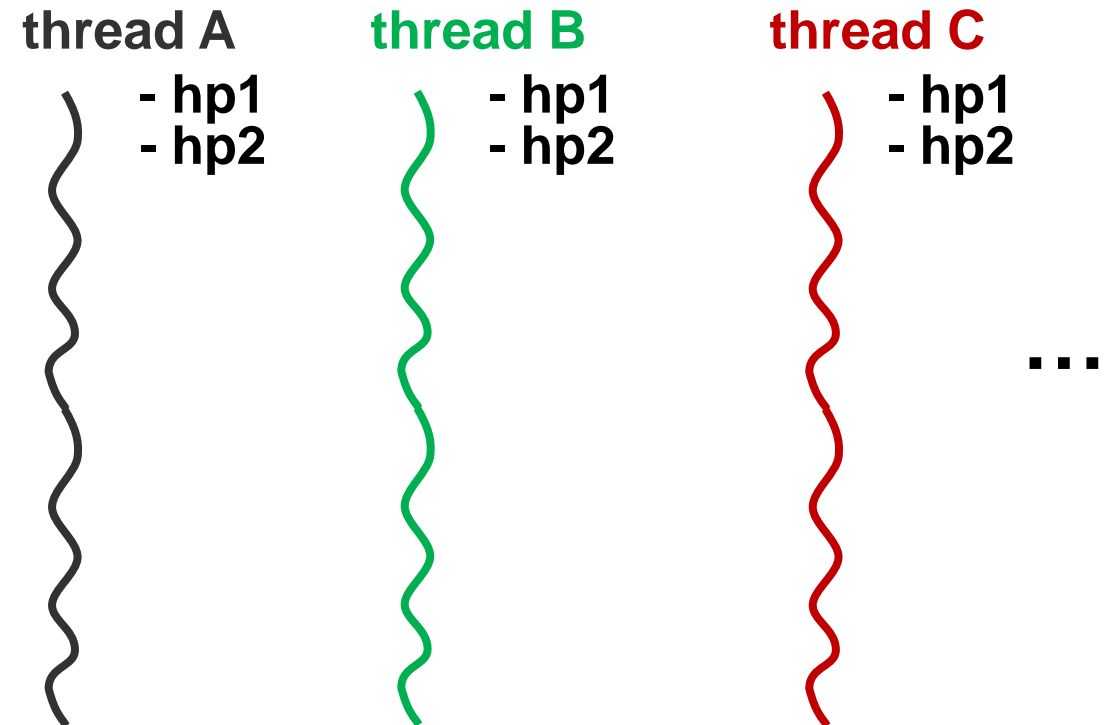
Key idea of Cooperative MT & Lock-free Algorithms

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

Time Sharing

Cooperative Multitasking

user mode   user mode

function call

thread A

- 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

thread B

time

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?

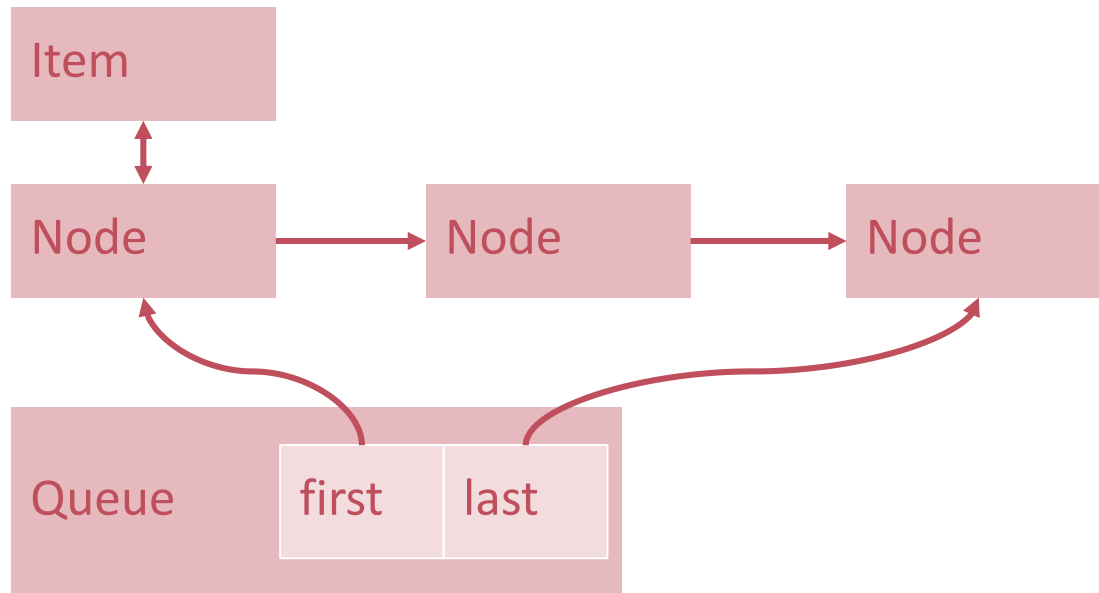Device drivers are interrupt-driven

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

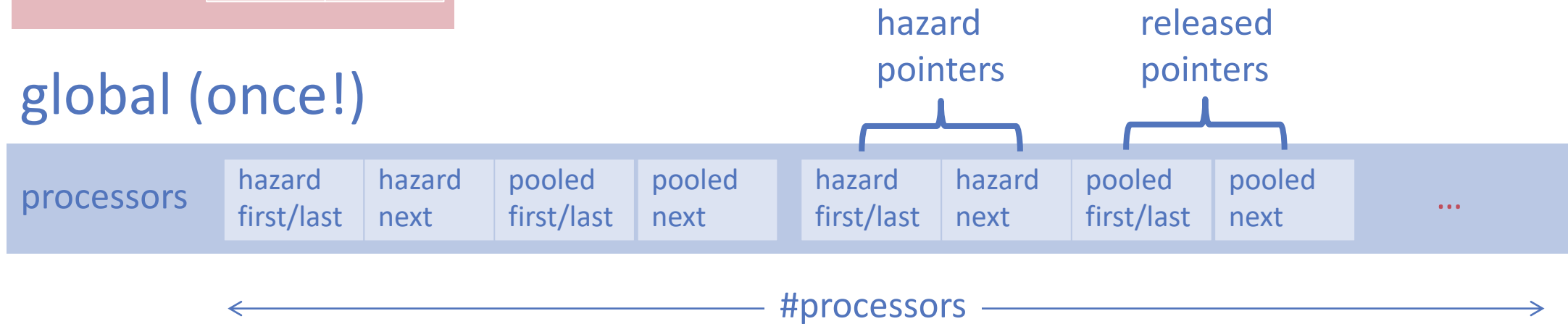Key idea: model interrupt handlers as virtual processors

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

Queue Data Structures

for each queue

Item

Node → Node → Node

Queue | first | last

hazard pointers

released pointers

global (once!)

| 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: the node in reference was set hazardous before it was here available in reference

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
```

accessed via activity register

access to current processor

stack management

quantum and scheduling

active object

```
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;
```
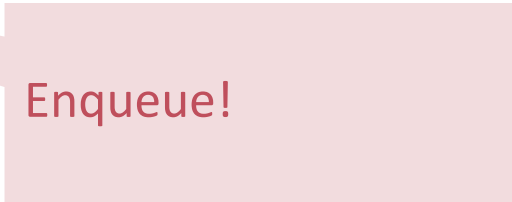
Enqueue runs on new thread

Calls finalizer of previous thread

```
(* 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.Enqueue (previous, queue^);
        IF ADDRESS OF queue^ = ADDRESS OF readyQueue[IdlePriority] THEN RETURN END;
        IF Counters.Read (working) < Processors.count THEN Processors.ResumeAllProcessors END;
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;
```
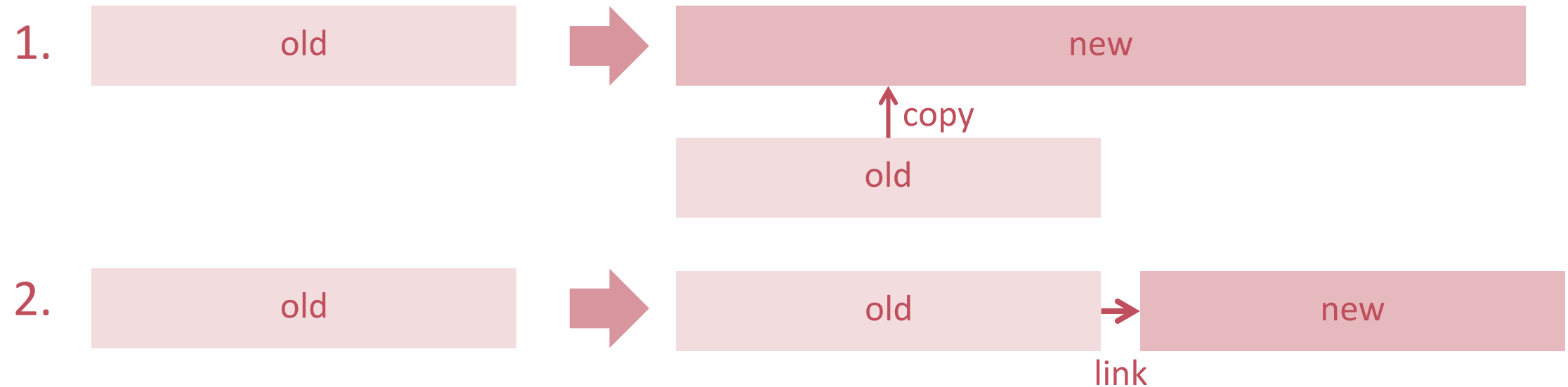
Enqueue!

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

caller of
A.B

A.B

parameters

pc

fp

proc desc

var

par (copy)

pc (**ReturnToStackSegment**)

fp

pdesc of A.B

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