

# To Preempt or Not To Preempt, That is the Question



Bernard Blackham, Vernon Tang and Gernot Heiser

NICTA and University of New South Wales  
Sydney, Australia



Australian Government  
Department of Broadband, Communications  
and the Digital Economy  
Australian Research Council

## NICTA Funding and Supporting Members and Partners



Australian  
National  
University

UNSW  
THE UNIVERSITY OF NEW SOUTH WALES



Griffith  
UNIVERSITY



Queensland  
Government

State Government  
Victoria

QUT  
Queensland University of Technology



THE UNIVERSITY OF  
QUEENSLAND  
AUSTRALIA

# The hard real-time myth

Protected RTOSes need a  
*fully-preemptible* kernel



Painting by Antonio di Correggio  
Image © Zenodot Verlagsgesellschaft mbH  
Licenced under the GNU Free Documentation Licence

# The hard real-time myth

Protected RTOSes need a  
*fully-preemptible* kernel

?

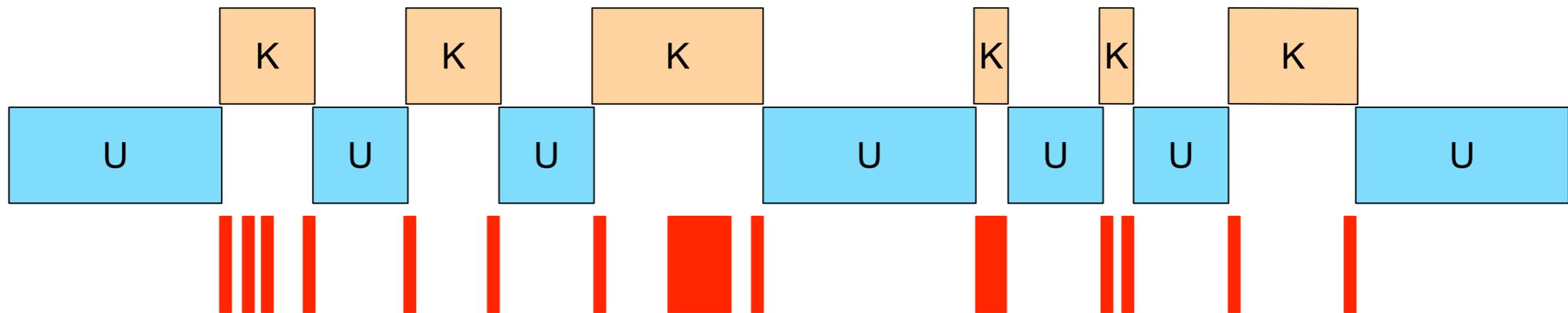
Everybody does it!

- ▶ QNX Neutrino
- ▶ Wind River VxWorks
- ▶ Green Hills Software INTEGRITY
- ▶ ENEA OSE
- ▶ Mentor Graphics VRTXsa
- ▶ Symbian EKA2
- ▶ ...



# What does that even mean?

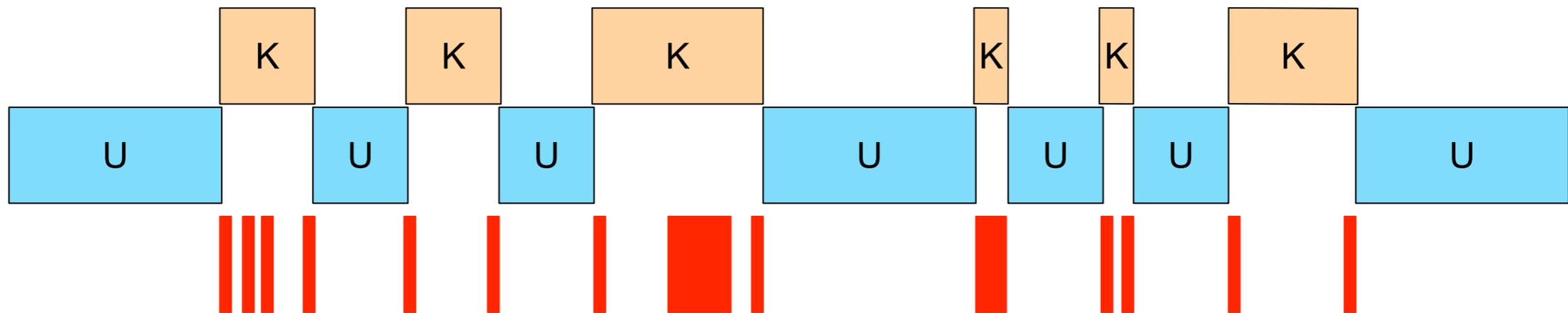
- **Fully-preemptible** kernels:
  - Interrupts allowed (almost) anywhere inside the kernel
  - Interrupts masked briefly during critical sections



# What does that even mean?

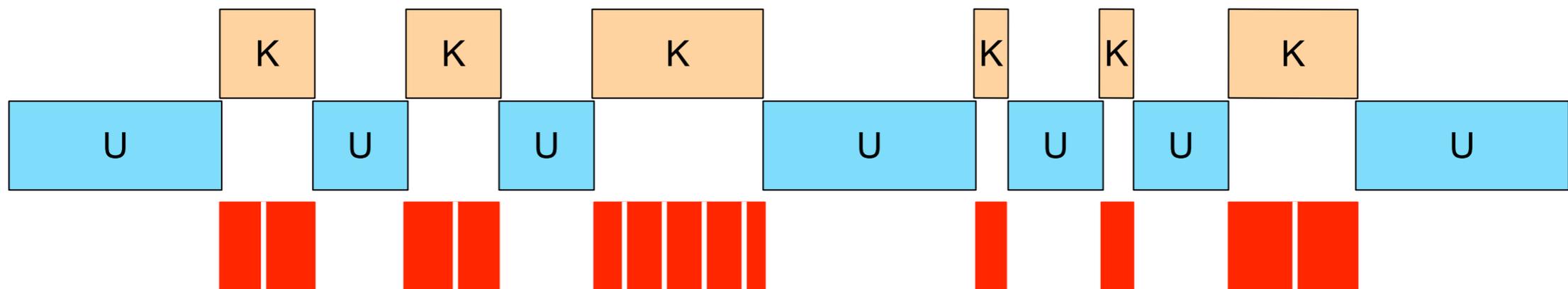
- **Fully-preemptible** kernels:

- Interrupts allowed (almost) anywhere inside the kernel
- Interrupts masked briefly during critical sections



- **Non-preemptible** kernels:

- Interrupts almost always masked while kernel is executing
- Pending interrupts polled for at preemption points



# What really matters?

- Hard real-time systems **don't** care about average-case interrupt latency
- Hard real-time systems **do** care about:
  - ▶ Worst-case interrupt latency
  - ▶ Functional correctness
  - ▶ Average-case performance (secondary concern)

# On functional correctness

- **Fully-preemptible** kernels:
  - cannot be formally proven correct (yet)
  - have exponentially more possible states than non-preemptible kernels
  - are akin to multi-threaded programming

# On functional correctness

- **Fully-preemptible** kernels:
  - cannot be formally proven correct (yet)
  - have exponentially more possible states than non-preemptible kernels
  - are akin to multi-threaded programming

“... a folk definition of insanity is to do the same thing over and over again and to expect the results to be different.

By this definition, we in fact require that programmers of multithreaded systems be insane.

Were they sane, they could not understand their programs.”

- Edward A. Lee

# On functional correctness

- **Fully-preemptible** kernels:
  - cannot be formally proven correct (yet)
  - have exponentially more possible states than non-preemptible kernels
  - are akin to multi-threaded programming
- **Non-preemptible** kernels:
  - can be formally proven correct (as demonstrated by seL4)
  - have less concurrency, making testing and assurance easier

# On average-case performance

- **Fully-preemptible** kernels must use one of:
  - locks (mutexes or spinlocks)
    - Cost: minimum is disable/enable interrupts, memory access per lock/unlock
  - lock-free algorithms (e.g. speculative lock elision)
    - Cost: generally more complex code (at least larger i-cache footprint)
- **Non-preemptible** kernels, using preemption points must:
  - Poll regularly for interrupts
    - Cost: efficient implementation: one memory access per preemption period

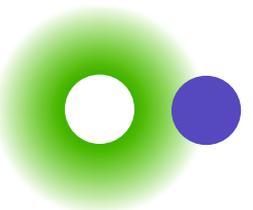
# What really matters?

- We only consider protected RTOSes
  - Multi-criticality systems require isolation
    - (assume execution of malicious code)
  - Without hardware-enforced protection, all bets are off
  - Interrupts are delivered to **userspace handlers**
- Monolithic kernels are likely fully-preemptible
  - e.g. Linux, Windows, Mac OS X
- Most RTOSes build upon a {micro,nano,exo}kernel

# Our intuition

- Microkernel operations **should be kept brief**
  - Managing address spaces, hardware access, threads and IPC
  - Why should anything take a long time?
- Kernel entry/exit and context switch times on modern hardware are high
  - x86: 100s-1000s of cycles per context switch
  - ARM Cortex-A8: minimum of 150 cycles per context switch

Execution time of **kernel operations** should be **within an order of magnitude** of hardware-imposed **context switching times**.



# A look at interrupt latency

Can **non-preemptible** kernels match *worst-case* interrupt latencies of **fully-preemptible** kernels?

# A look at interrupt latency

Can **non-preemptible** kernels match *worst-case* interrupt latencies of **fully-preemptible** kernels?

- Representative of fully-preemptible: **QNX**
  - Mature, popular commercial RTOS
  - Uses QNX Neutrino microkernel
- Representative of non-preemptible: **seL4**
  - Formally verified with machine-checked proof of correctness
  - Worst-case execution time analysis previously completed (RTSS'11)

# The fully-preemptible: QNX

We analysed QNX to compute its worst-case interrupt latency

- Using static analysis to give a safe approximation
- Investigated all “disable-interrupt” regions
- Selected a representative subset of Neutrino microkernel
  - Ignored some complex code (e.g. sporadic scheduling)
- Chosen subsets include interrupt delivery paths, plus anything else which could be analysed

Results give a **lower-bound**  
on worst-case interrupt latency

# The fully-preemptible: QNX

Not without some “fun” technical challenges:

- Managing a very large control-flow graph
  - Subsets chosen carefully to reduce complexity
- Some code generated at boot-time
  - Executable extracted from running image
- Following function pointers
  - Resolved manually by user

# The fully-preemptible: QNX

- ▶ Longest disable-interrupt region: **4,441 cycles**
- ▶ WCET of interrupt delivery to userspace: **17,413 cycles**

\* (These are lower bounds)

# The non-preemptible: seL4

- Previous analysis and improvements give interrupt latency of ~200,000 cycles (EuroSys'12)
- Improvements constrained by ability of formal verification

Ignoring formal verification\*, how much better can we do?

\* Fallback to industry standard methods of testing: code coverage tests, code inspection, model checking, etc.

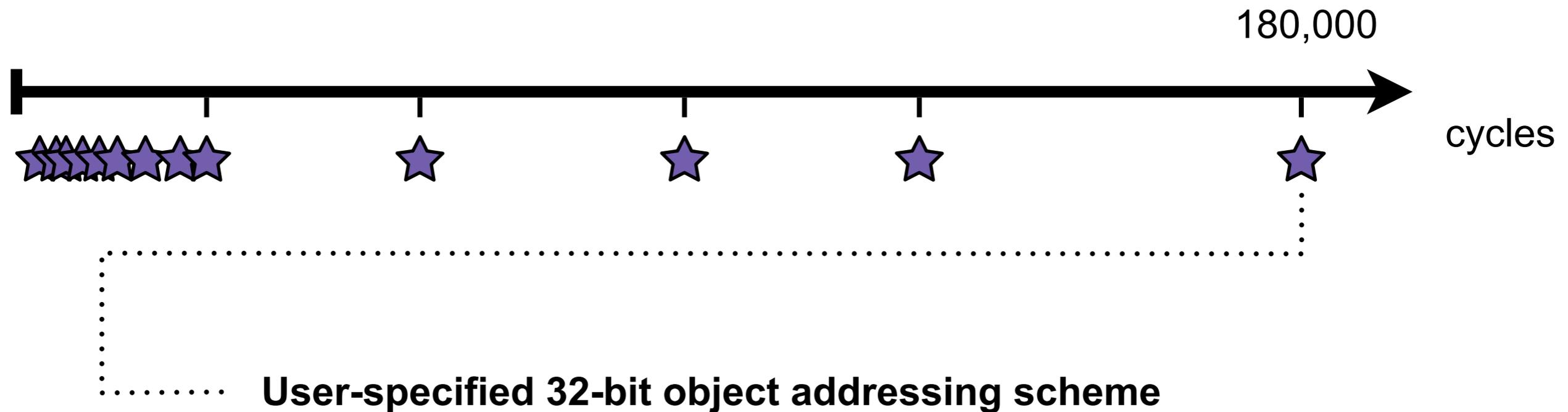
# The non-preemptible: seL4

- What causes the worst-case?



# The non-preemptible: seL4

- What causes the worst-case?

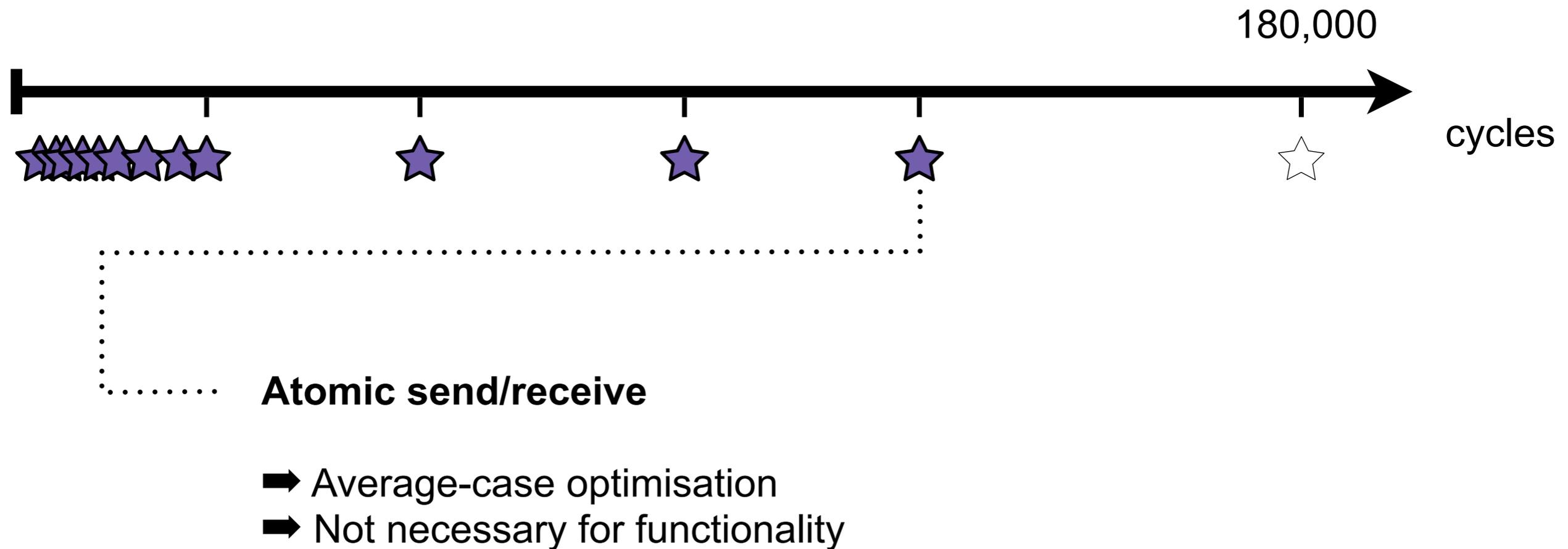


- ➡ Up to 32 cache misses per lookup
- ➡ Up to 11 lookups per system call

**Solution: limit object addressing to 1 level (like other RTOSes)**

# The non-preemptible: seL4

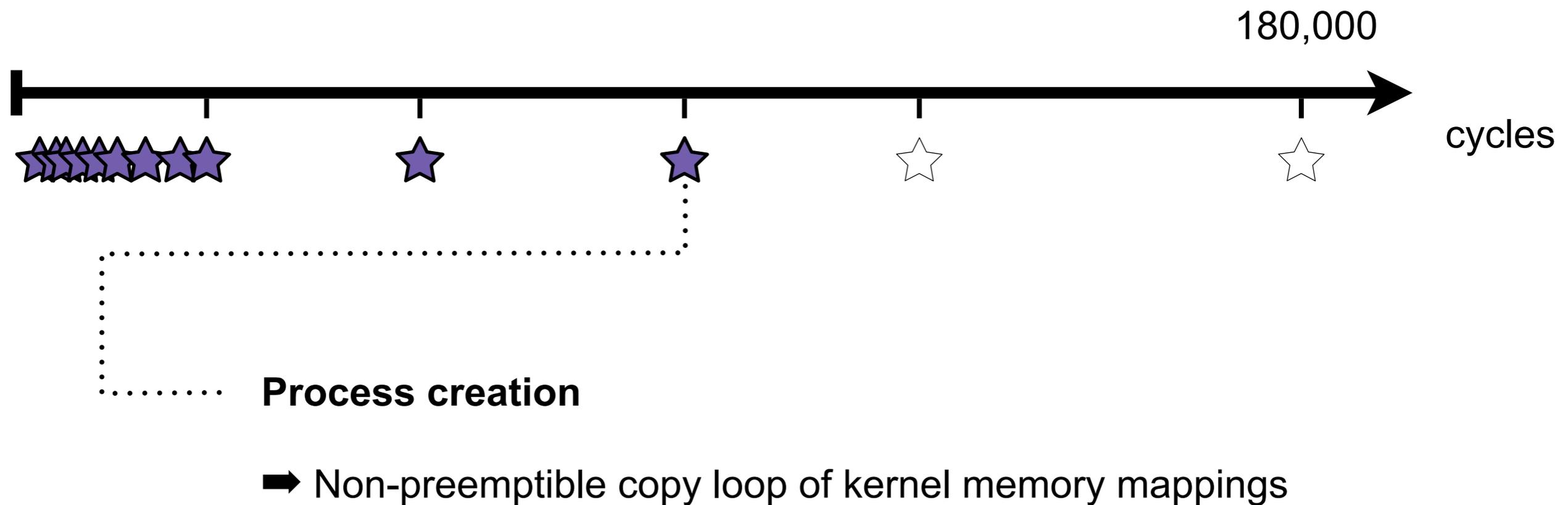
- What causes the worst-case?



**Solution: add preemption point between send/receive in slow path**

# The non-preemptible: seL4

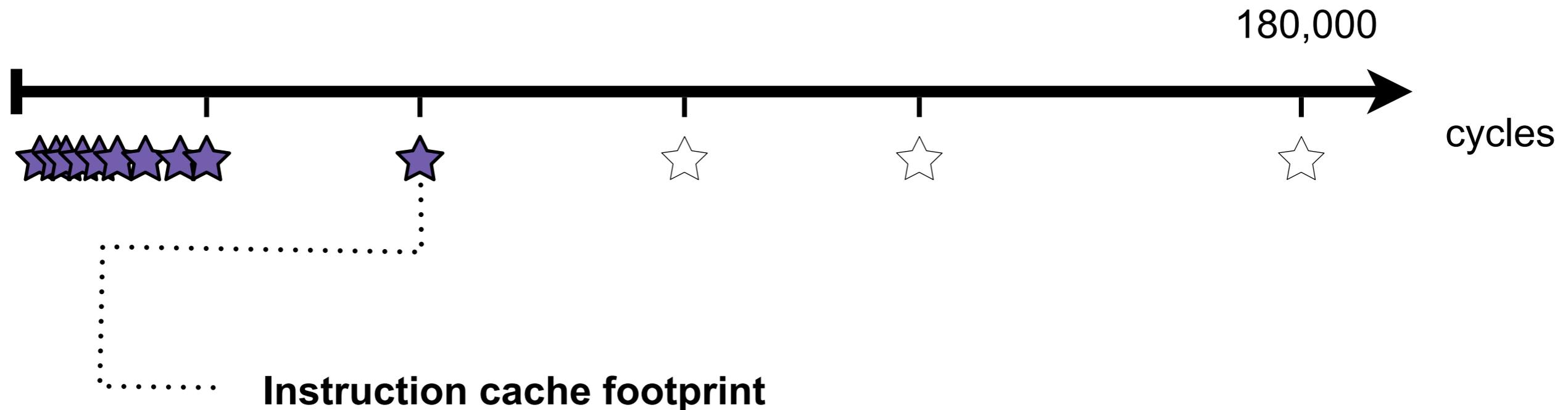
- What causes the worst-case?



**Solution: add preemption point, or use a separate kernel address space**

# The non-preemptible: seL4

- What causes the worst-case?

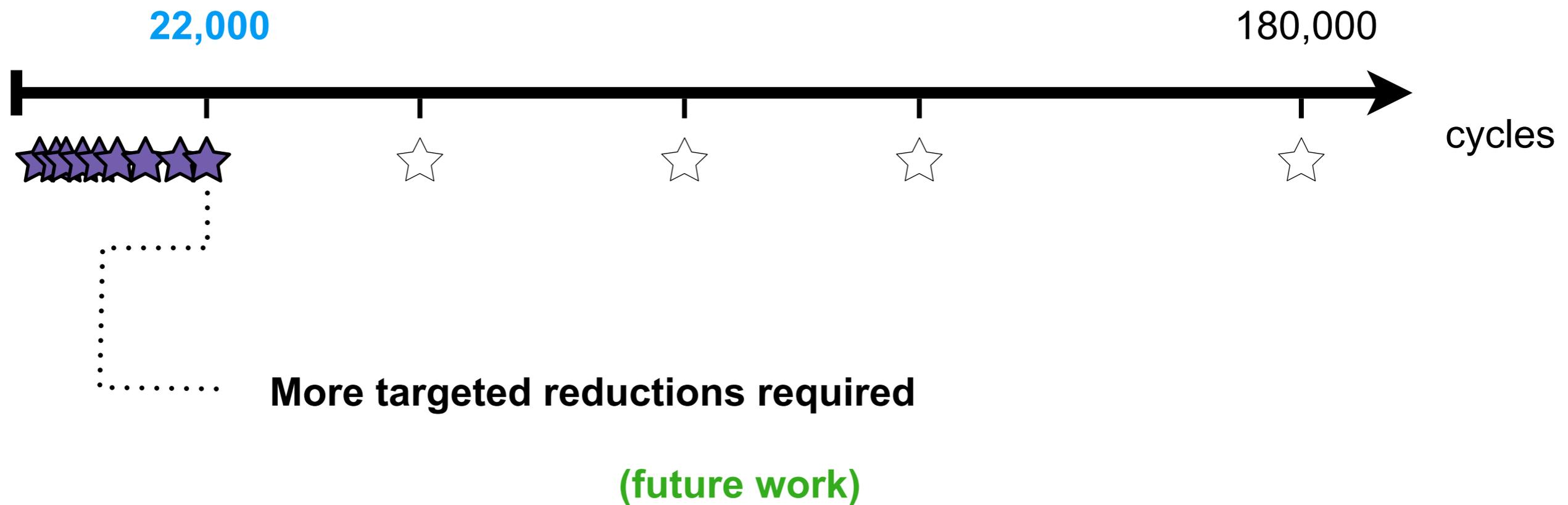


- ➡ Most loops are preemptible
- ➡ Worst-case execution time caused largely by instruction cache misses

**Solution: pin microkernel into faster L2 cache (only 36 KiB of kernel text!)**

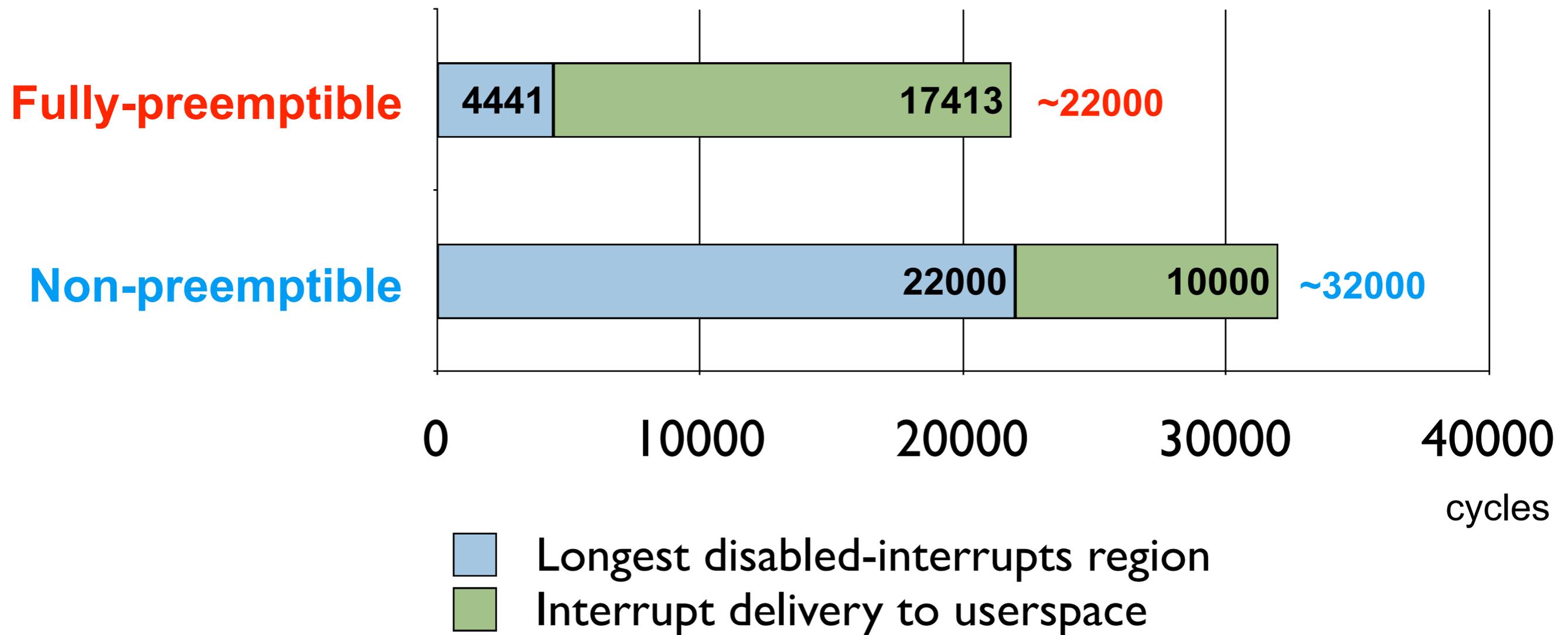
# The non-preemptible: seL4

- What causes the worst-case?



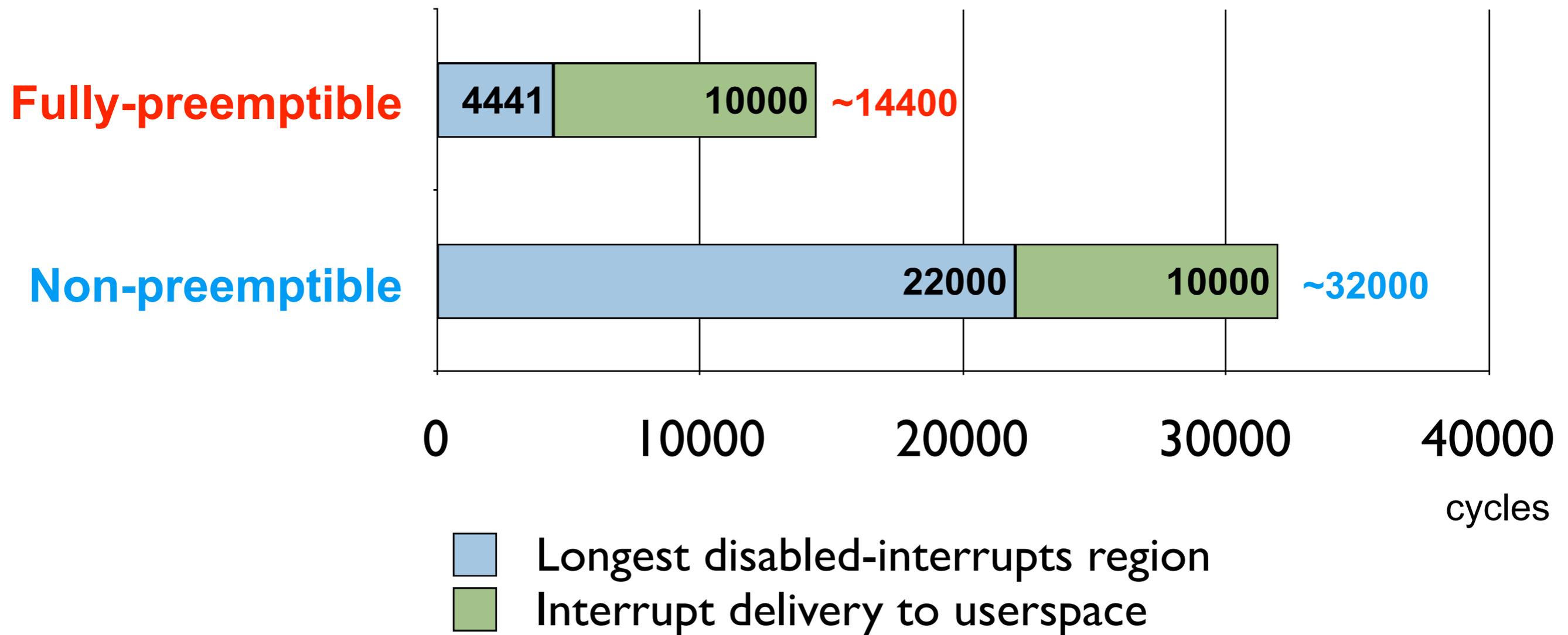
# How do they stack up?

## Worst-case interrupt response time



# How do they stack up?

## Worst-case interrupt response time



## Back to the mythology...

Do protected RTOSes need a *fully-preemptible* kernel?

- Our analysis suggests not – i.e., that **non-preemptible** kernels can be competitive with **fully-preemptible**
- **Non-preemptible** kernels can offer greater assurance
- **Non-preemptible** kernels can offer better average-case performance
- So why choose to go insane?

