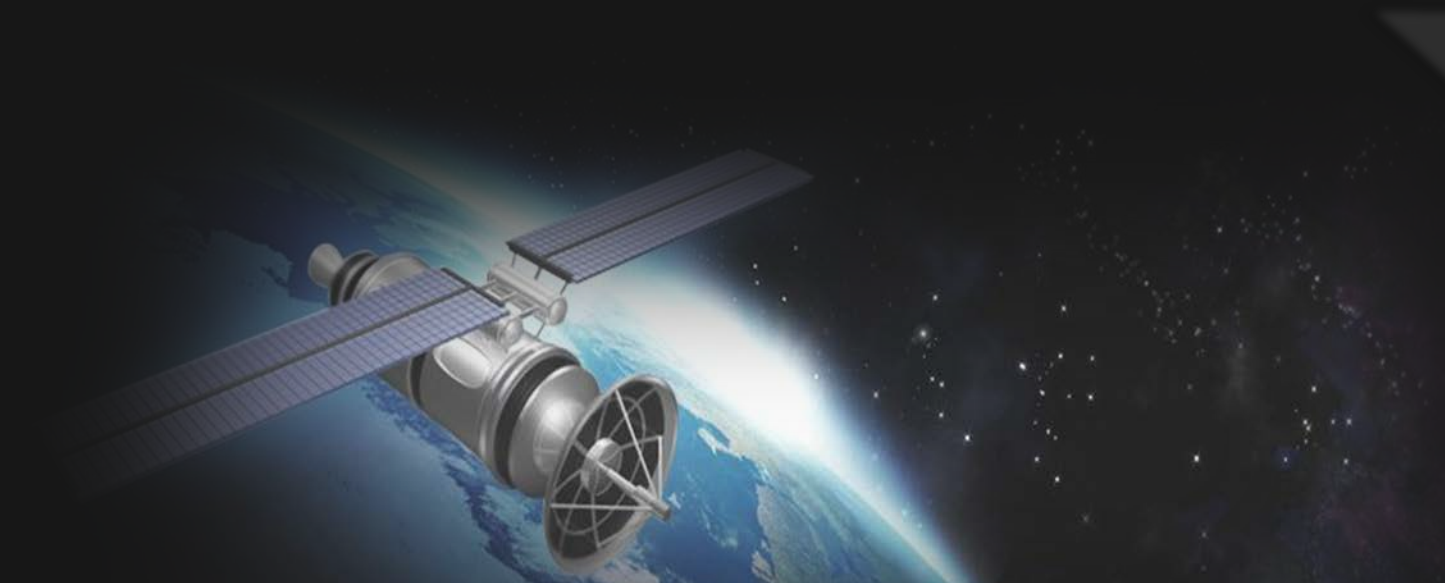


Reaching into space
TOGETHER

Development Environment for Future Leon MultiCore

FINAL PRESENTATION

01/06/2015

UNIVERSITÀ DEGLI STUDI DI PADOVA

embedded brains

SPACEBEL

Agenda

- Project Team
- Conclusions Spacebel
  - On Multi-core
  - RTEMS SMP Outlook
  - Parallel libraries
- RTEMS product status – embedded brains
- State of the art of real-time multi-core systems – University of Padua
- Questions

Project Team

- Spacebel, University of Padua, embedded brains
  - Very complementary and cooperative
    - Building RTOS is a special profession
    - Predictable multi-core systems is a special profession
    - Space embedded systems is a special case
- Parallel contract with team Cobham (Aeroflex) Gaisler AB
  - Very complementary and cooperative
    - Good cooperation with RTEMS team (OAR Corporation)
    - Outstanding integration/validation by Cobham Gaisler AB

Spacebel Conclusions on Multi-Core

- Processors
  - Single core NGMP @200 MHz ≈ 4-4,5 times faster than single core GR712 @50 MHz
- RTEMS SMP
  - Reference: Proba DHS + image processing
  - Enabling all cores, core 0 runs only Proba DHS
    - GR712:
      - DHS slows down by 3 %
      - gains 90 % of processing capacity
    - NGMP:
      - DHS slows down by 4 %
      - gains 270 % of processing capacity

Outlook RTEMS SMP

- Exploiting RTEMS SMP
  - Core allocation not in OBSW source code
  - Traditional designs moved over several cores
    - tightly coupled
    - more overhead
    - determinism needs further analysis
  - But keeping existing OBSW design on one core
    - Adding new designs on other cores possible
      - But need loosely coupling (i.e. decoupled message queues)
      - Sufficient processing resources available
        - for payloads or instruments
- Exploiting all cores potential requires dedicated design
- RTEMS SMP needs further optimisation to minimise Giant Lock

Parallel Libraries

- Giant lock and object API make parallel libraries costly for CPU
- Existing parallel libraries (Cilk Plus, OpenMP, …)
  - Are optimised for target processor architectures
  - Try to bypass OS as much as possible
  - Tend to use active polling
  - Difficult to validate
  - Maturity lacking for use in embedded systems
  - Current state does not provide real advantages over discrete programming
- Proper parallel libraries are the way to go when
  - RTEMS SMP is optimised for that
  - Optimal Parallel library run-times are designed on top of that

Product status

RTEMS: After 20^{+} years in operation,
from single-core to multi-core

Strategy: Single-Core → Multi-Core

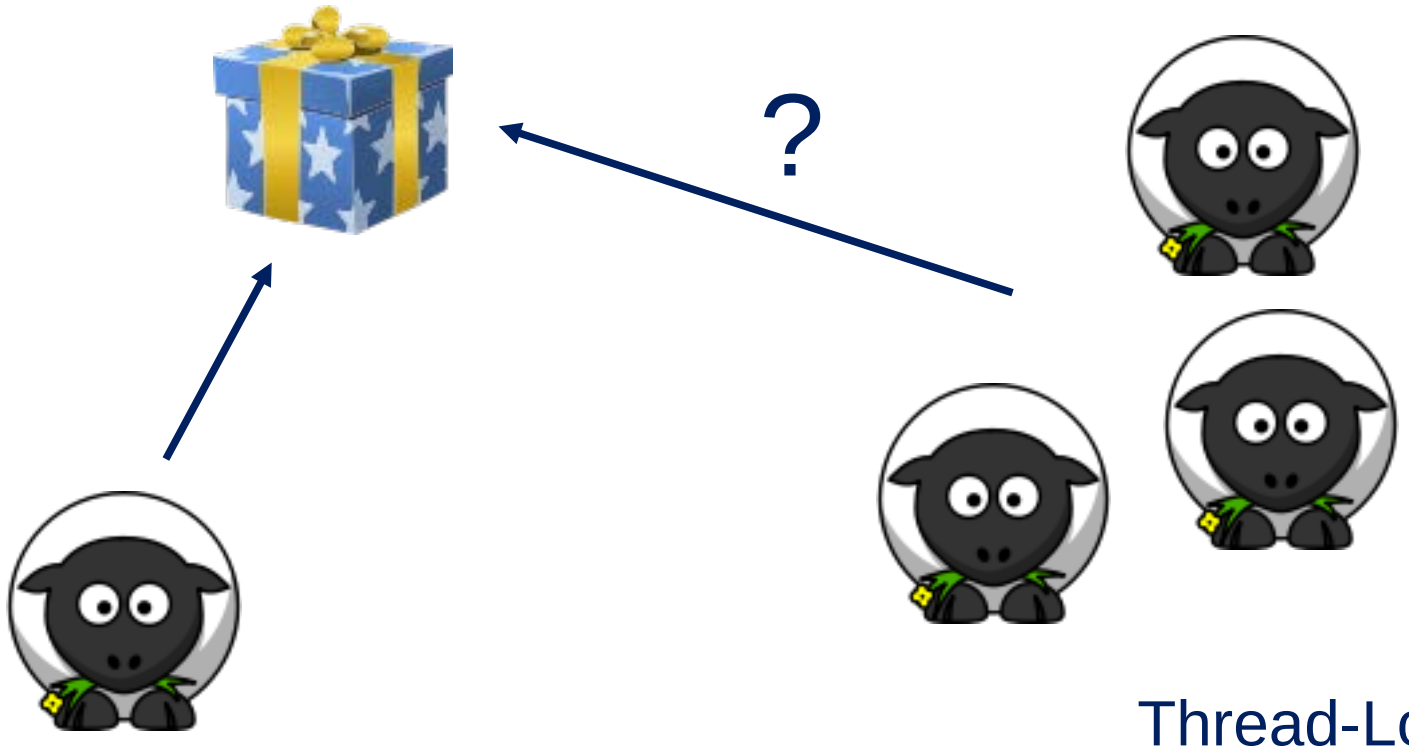
- Evaluate high-level APIs
- Evaluate and choose low-level synchronization primitives
- Get it running on multi-core with minimal effort
- New APIs (e.g. partitioned/clustered scheduling)
- Add profiling to identify bottlenecks
- Get rid of bottlenecks step by step

More than one executing thread

- Interrupt service routines? Timer routines?
- Stop timers?
- Thread priority? Mutual exclusion?
- Disabled pre-emption? Mutual exclusion?

vs.

Task variables



POSIX Keys

Thread-Local Storage

Task Variable = Global Variable
Changed during Context Switch

Low-Level Synchronization

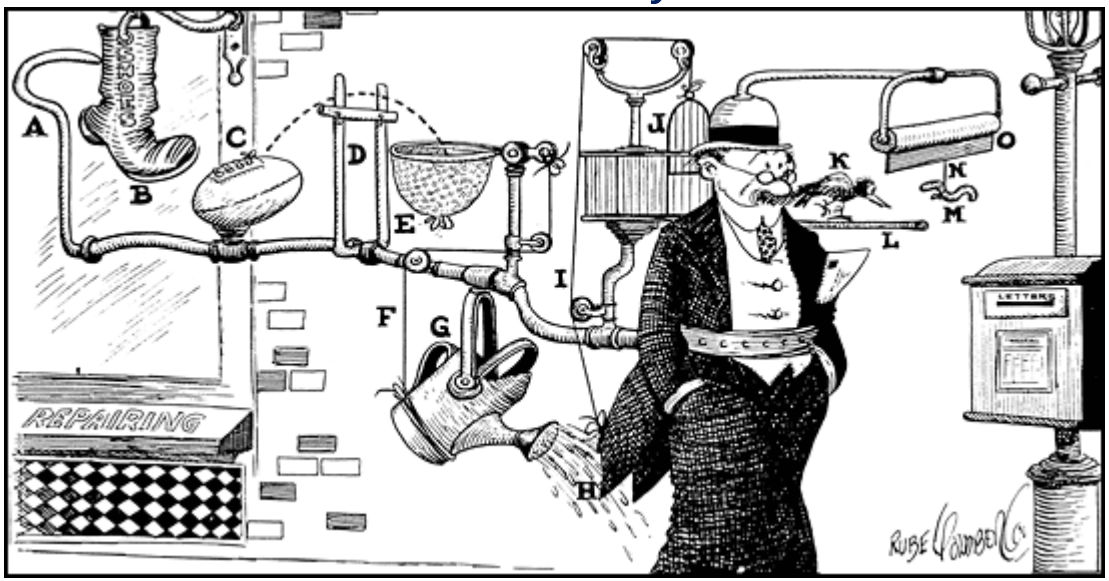
Thundering Herd

Lock-Free Algorithms

Spin Locks

Lock Convoy

Patents



VS.

Everyone can use
a hammer (disable interrupts)

Atomic Operations

C11

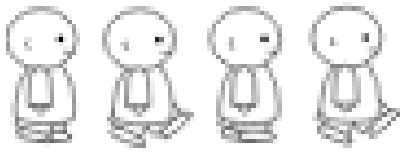
C++11

Livelock

Memory Models

Fairness

Deadlock



Keep it simple

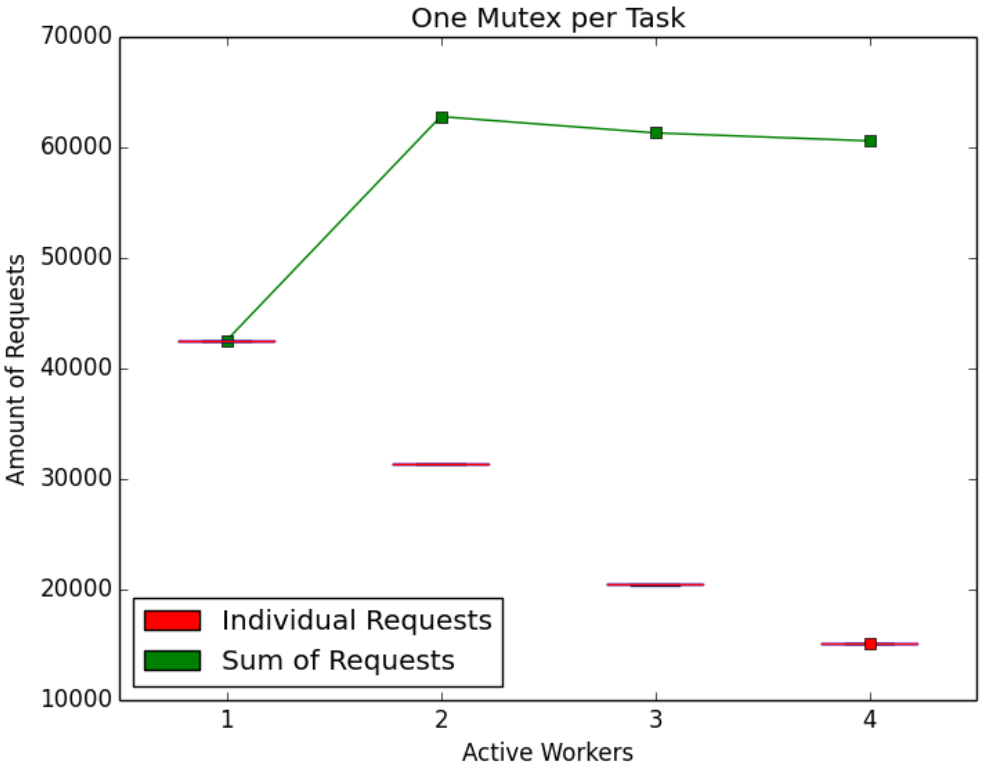
- FIFO fairness is required for low-level mutual exclusion (we want a predictable system, not maximum throughput)
- Use a ticket lock implemented via C11 atomic operations
  - Simple and space efficient implementation
  - Write to single location (next serving) during release
  - Write to single location (ticket) during acquire
  - Only reads during busy wait
- API capable of using Mellor-Crummey Scott (MCS) locks

The Giant Lock

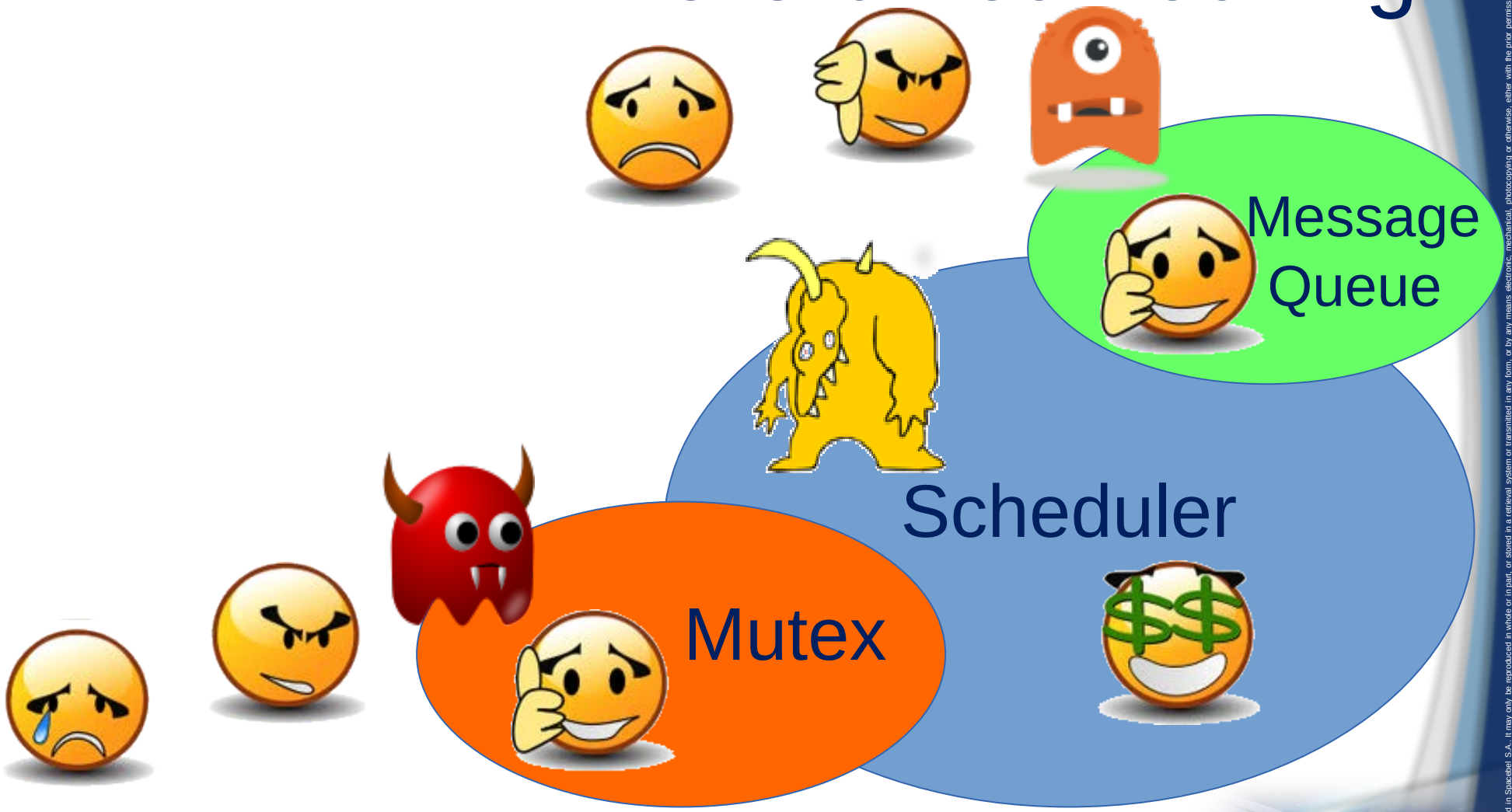
Operating System

The Giant Lock in Action

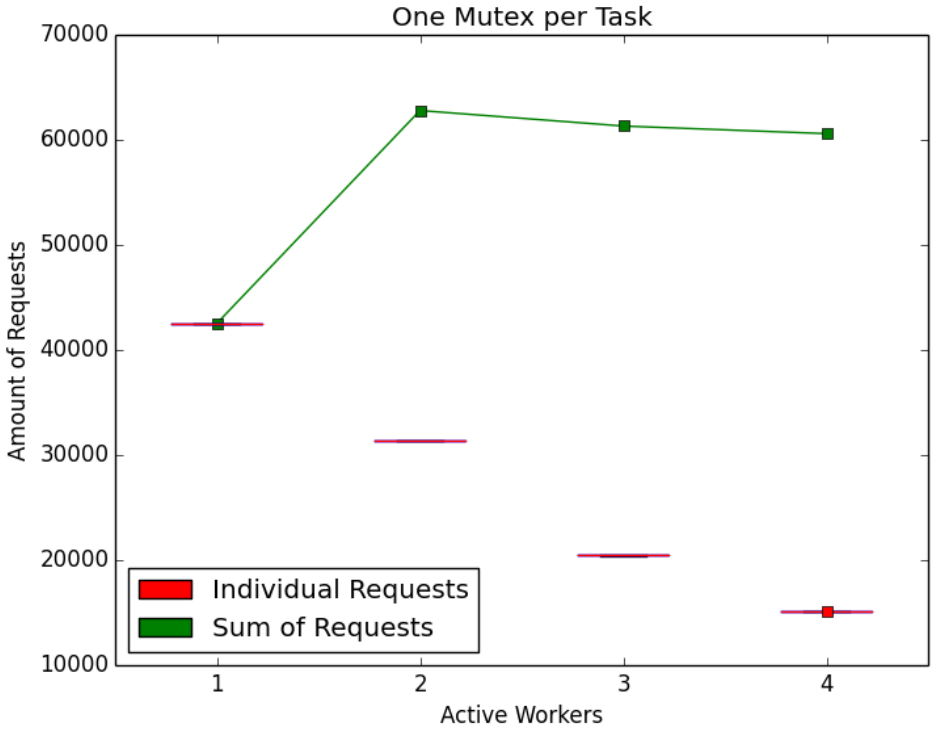
```
mtx ← new mutex
while true:
      mtx.obtain
      mtx.release
```

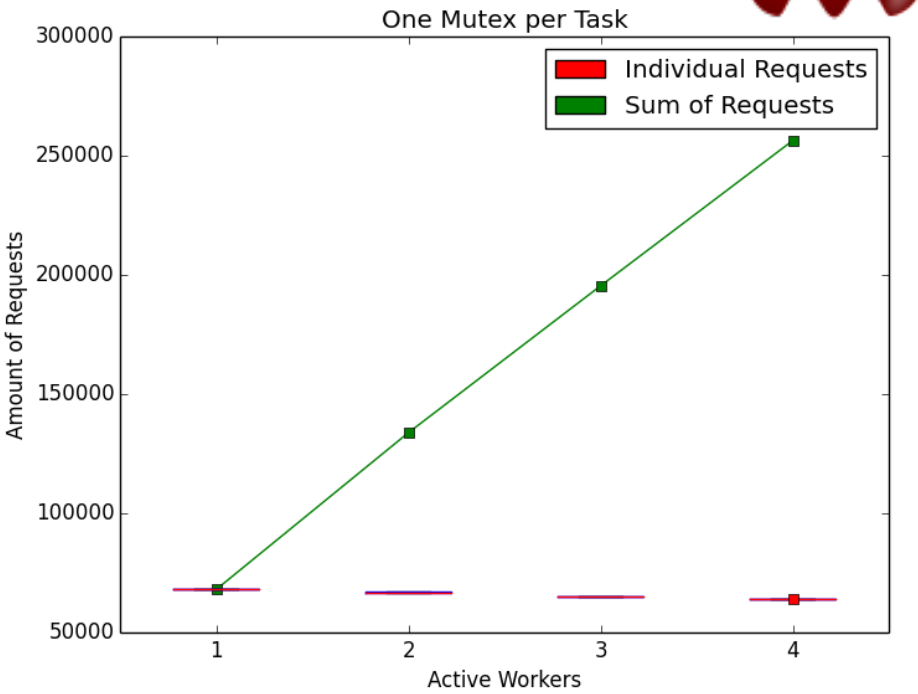


Fine Grained Locking



Fine Grained Locking for Mutexes





Profiling

- Get run-time statistics of low-level synchronization primitives
  - Spin-locks
  - Interrupt processing
  - Thread dispatch critical sections (per core)
- Low-overhead measurement of short time intervals (problematic on NGMP prototype, maybe fixed on GR740)
- Acceptable overhead for production systems
- XML reports for test programs (more than 500)

Per Core Profiling Example

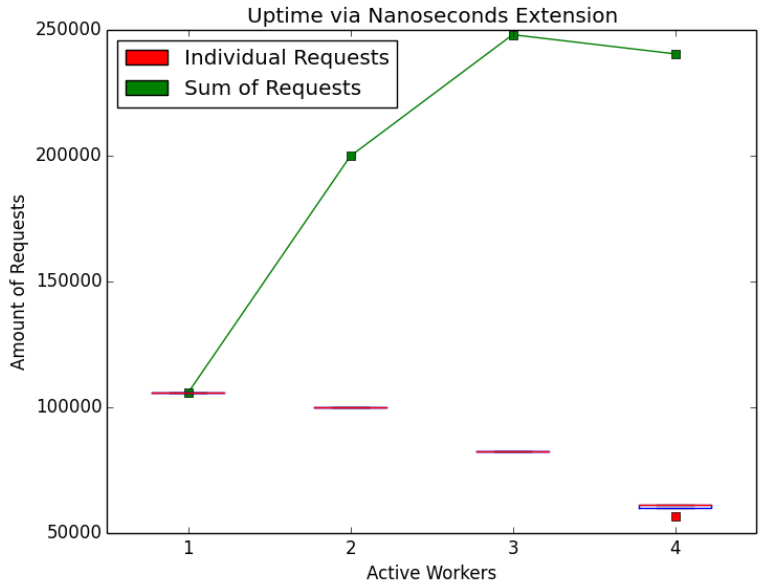
```
<PerCPUProfilingReport processorIndex="0">
  <MaxThreadDispatchDisabledTime unit="ns">3807457</...>
  <MeanThreadDispatchDisabledTime unit="ns">124091</...>
  <TotalThreadDispatchDisabledTime unit="ns">1706880473</...>
  <ThreadDispatchDisabledCount>13755</ThreadDispatchDisabledCount>
  <MaxInterruptDelay unit="ns">0</MaxInterruptDelay>
  <MaxInterruptTime unit="ns">24661</MaxInterruptTime>
  <MeanInterruptTime unit="ns">10148</MeanInterruptTime>
  <TotalInterruptTime unit="ns">127682501</TotalInterruptTime>
  <InterruptCount>12582</InterruptCount>
</PerCPUProfilingReport>
```

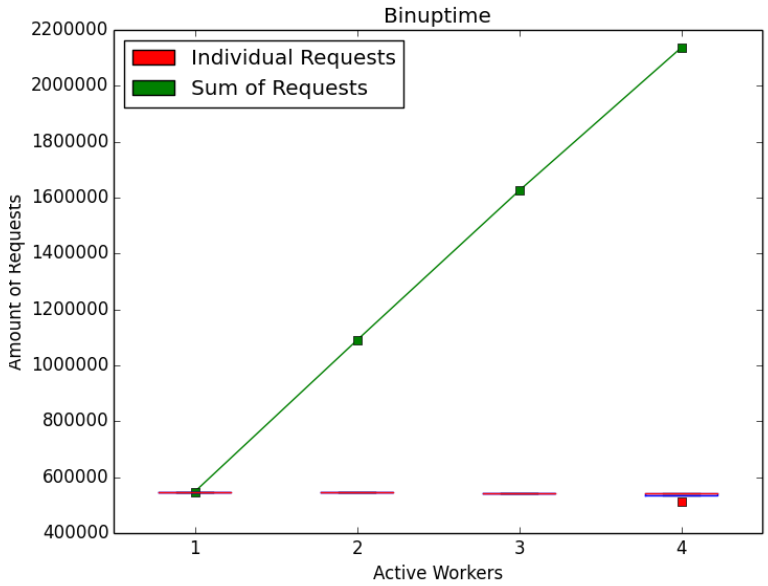
SMP Lock Profiling Example

```
<SMPLockProfilingReport name="Watchdog">
  <MaxAcquireTime unit="ns">47020</MaxAcquireTime>
  <MaxSectionTime unit="ns">2709</MaxSectionTime>
  <MeanAcquireTime unit="ns">31</MeanAcquireTime>
  <MeanSectionTime unit="ns">52</MeanSectionTime>
  <TotalAcquireTime unit="ns">990203330</TotalAcquireTime>
  <TotalSectionTime unit="ns">1674926849</TotalSectionTime>
  <UsageCount>31604848</UsageCount>
  <ContentionCount initialQueueLength="0">10574</ContentionCount>
  <ContentionCount initialQueueLength="1">8168</ContentionCount>
  <ContentionCount initialQueueLength="2">8578</ContentionCount>
  <ContentionCount initialQueueLength="3">31577528</ContentionCount>
</SMPLockProfilingReport>
```

Time Keeping

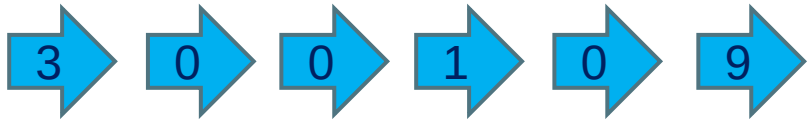
- Nanoseconds extension broken by design on SMP
- Global lock for timestamps





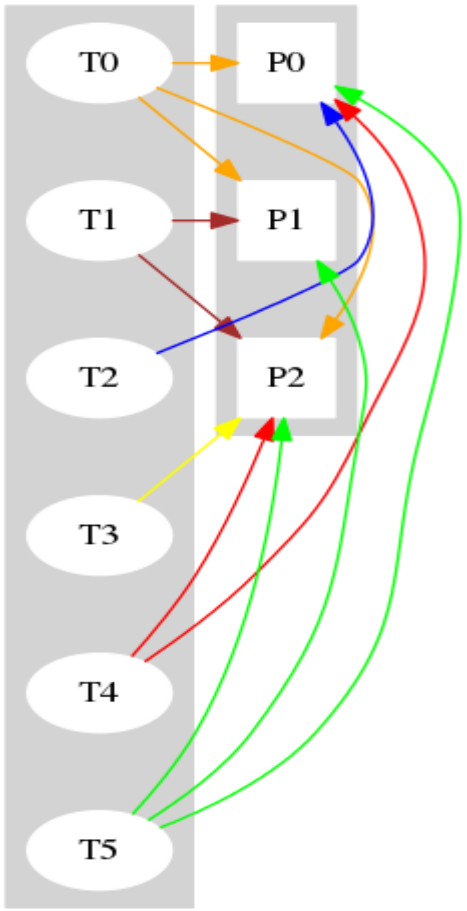
- Replace it with FreeBSD Timecounters

Watchdogs (Timers)

- Used global variables (very ugly)
  - Requires Giant lock
- Replaced with new implementation (less ugly)
- Enables per scheduler tick support
  - Needs clock driver support
  - Not yet implemented
- Uses delta chains 

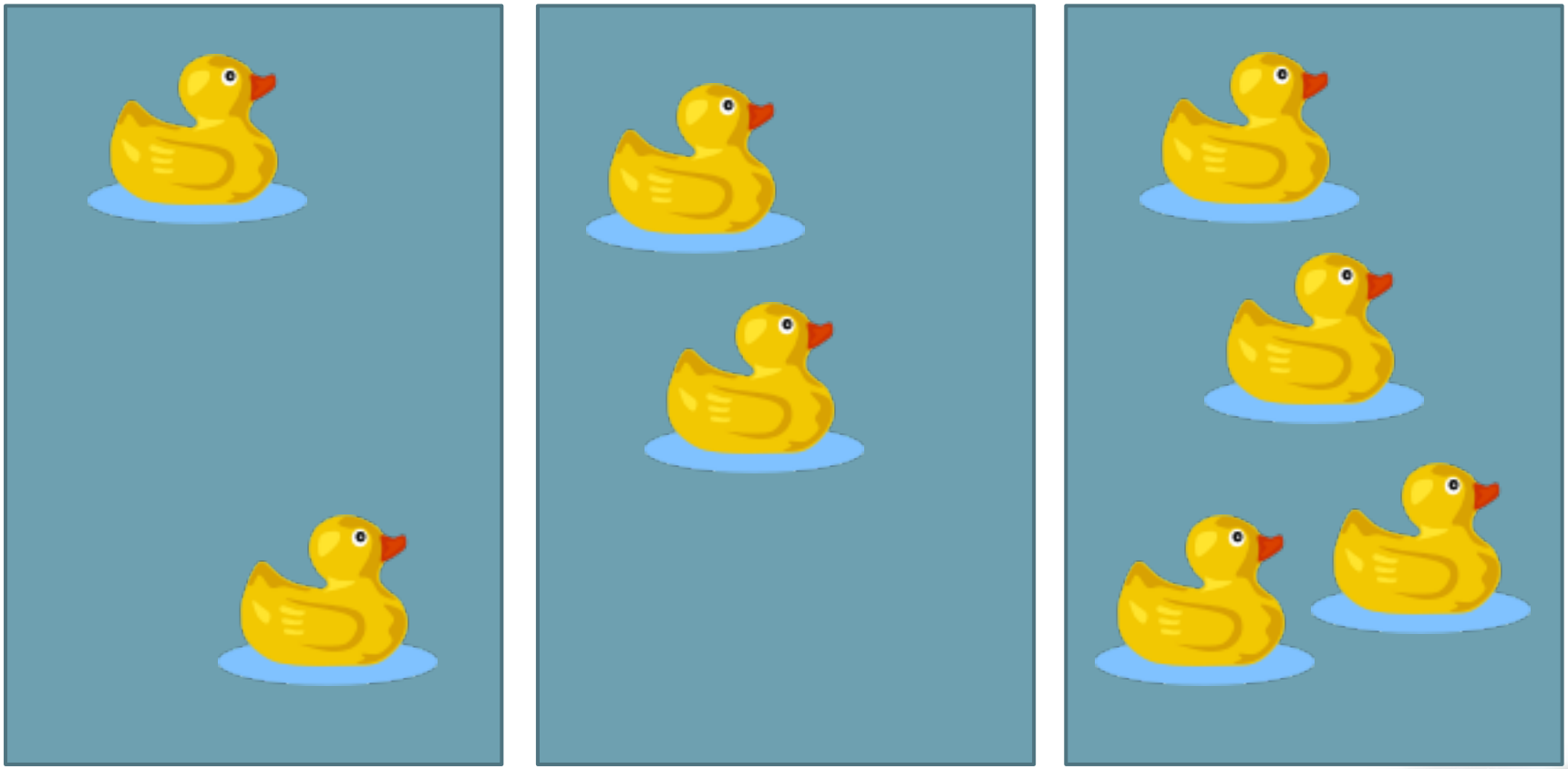
  - O(n) insert operation time complexity, n = count of watchdog in the chain
  - Maybe look for alternatives

Arbitrary Thread Processor Affinity



- General and flexible, covers
  - partitioned scheduling
  - clustered scheduling
- Easy to use API
  - Linux/BSD compatible pthread_setaffinity_np()
- Hard to predict runtime behaviour
- Efficient implementation unknown
- Coarse locking
- Matching problem in bi-partite graph

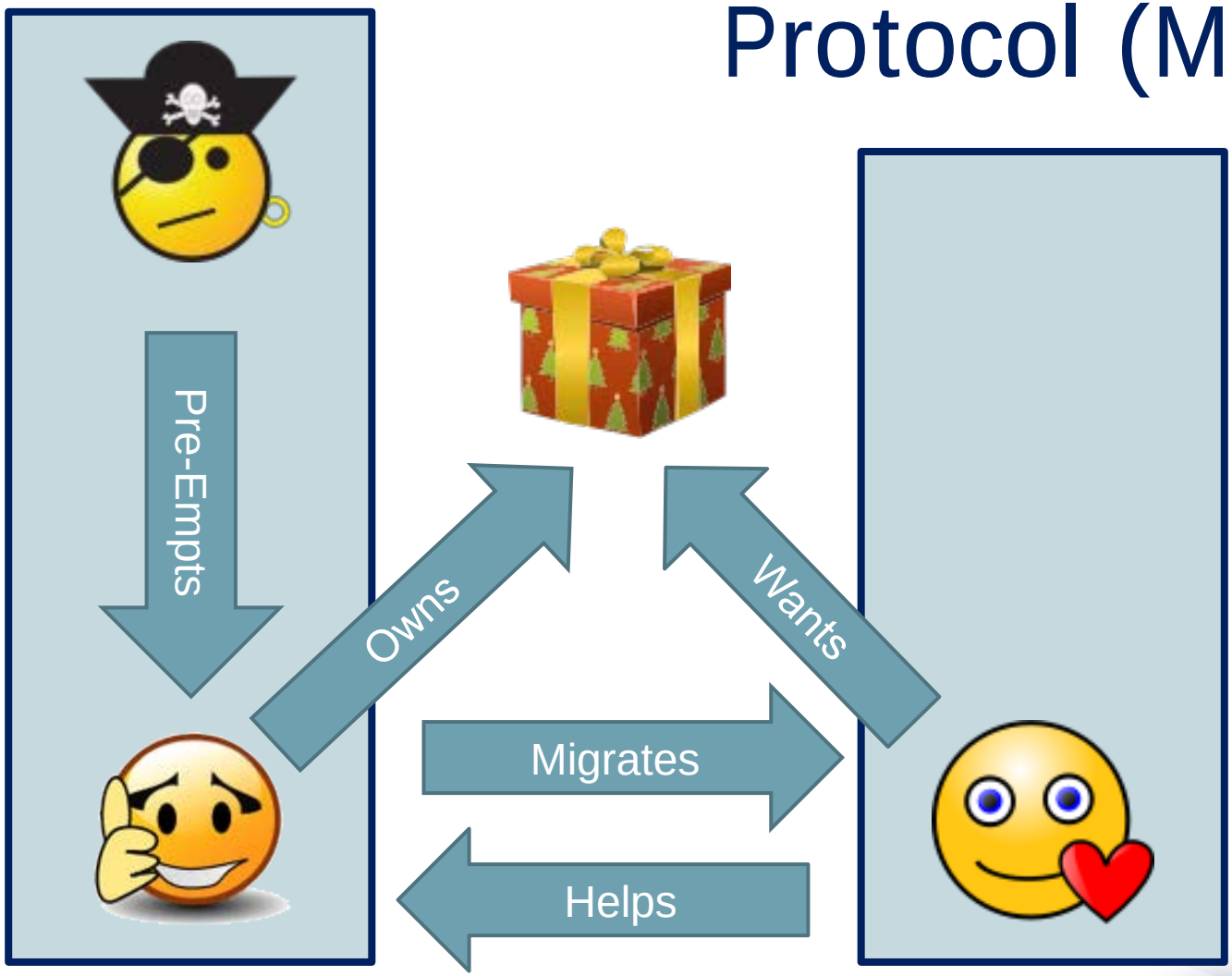
Partitioned/Clustered Scheduling



Partitioned/Clustered Scheduling

- Easy and efficient implementation
- Decoupled system (fine grained locking) ✗
- Thread partitioning during system design phase
- Intra-partition synchronization
  - Events ✓
  - Message queues ✓
  - Priority inheritance with priority boost ✗
  - O(m) independence-preserving protocol (OMIP) ✗
  - Multi-processor resource sharing protocol (MrsP) ✓

Multi-Processor Resource Sharing Protocol (MrsP)



Open Source

- RTEMS is available to everyone without registration or other obstacles
  - git clone git://git.rtems.org/rtems.git
- Work sponsored by other users during the ESA project
  - Basic SMP scheduler framework
  - SMP support for ARM and PowerPC
    - Helps to speed up development due to better debug support (e.g. Lauterbach PowerTrace)
    - Test runs on more targets reveal more bugs
  - Network stack port from FreeBSD
    - Prototype implementation for fine grained locking

Status Quo

- Partially ready for production systems
- Solid low-level implementation
  - Low-level synchronization
  - Thread migration and processor assignment
  - SMP scheduler framework
  - Partitioned/clustered scheduling
  - Thread queues (building block for objects which may block a thread)
- Low-overhead guest system for Time and Space Partitioning (IMA)

Quo Vadis?

- Eliminate Giant lock entirely
- Proper priority queues for partitioned/clustered scheduling (combination of FIFO and per scheduler priority queues)
- Support for priority boosting
- OMIP support
- Per scheduler locks
- New APIs for objects without an identifier to object translation and workspace usage

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

A look into the state of the art

Placing the study choices in a broader context

Premise /1

- Your current OBSW is designed for single-core processors
  - A reflection of culture and habit more than of need
- This causes the OBSW to scale poorly to a multi-core processor
- The obvious thought is to assign the whole OBSW to one core and use «the rest» for the payload SW
  - On the assumption that the payload SW is more «parallelizable»
  - And that nice segregation can be had between them

UNIVERSITÀ DEGLI STUDI DI PADOVA

Premise /2

- Segregation does not contemplate scaling however
- So you had better prepare the ground for more scalable solutions
  - In order that «when the future comes» you are prepared for it
- This study investigated sustainable solutions for scaling RTEMS to multi-core processing
  - Making it amenable to host OBSW that may need more than 1 core as well as a parallel P/L SW that can execute in a time predictable fashion

UNIVERSITÀ DEGLI STUDI DI PADOVA

Dilemmas /1

- To achieve high *schedulable utilization* (i.e., to sustain workload as high as the total capacity → *optimality*) scheduling must be *work conserving*
  - Any CPU cycle not used in any single core may cause some task to miss its deadline
- To be work conserving you need *global* scheduling
  - But global scheduling needs inordinate task migration which causes massive overhead from cache disruption and increase of traffic on the memory bus

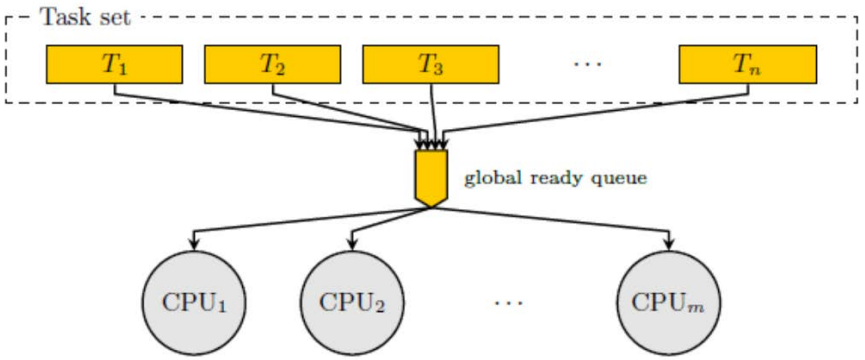
UNIVERSITÀ DEGLI STUDI DI PADOVA

Dilemmas /2

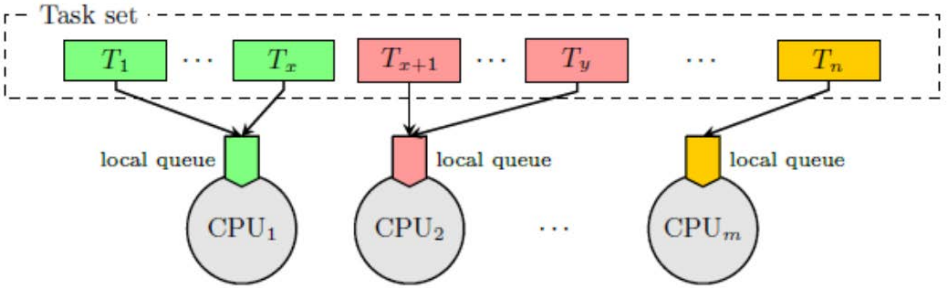
- You say: we don't need optimality
  - I add: perhaps only for now
- But then you do need *time predictability*
  - So that you can reason about best and worst cases to ascertain feasibility soundly
- Bad news is that the worst case on multicore processors is extremely difficult to determine
  - Because of parallelism

Solution space

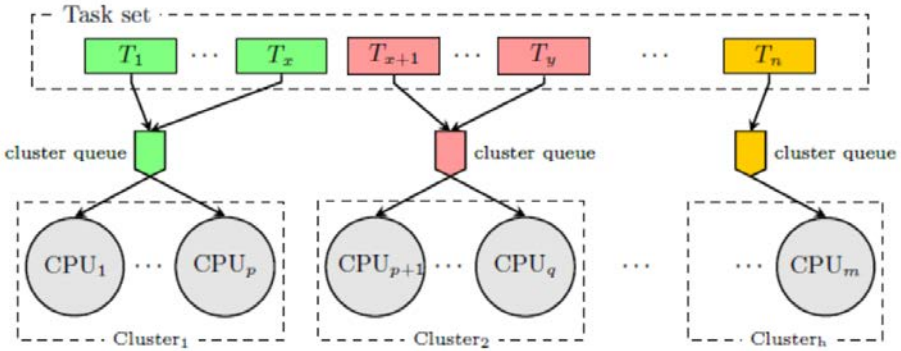
Global



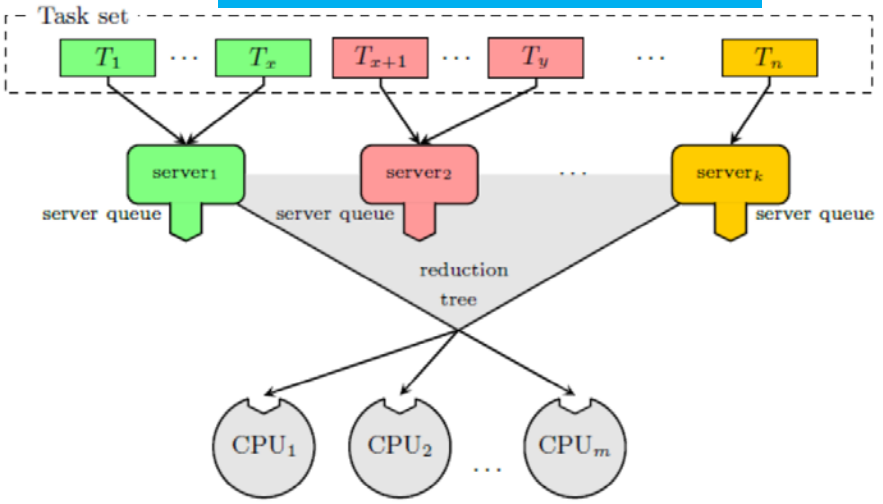
Partitioned



Clustered

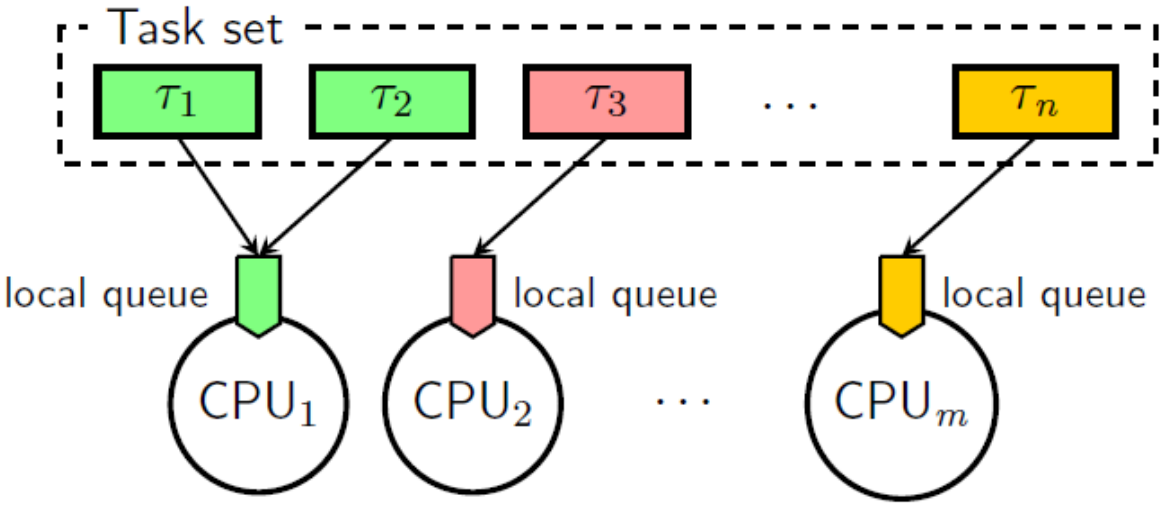


Hybrid (semi-partitioned)



Our choice /1

- We partition



- But so doing, we buy into *bin packing*
  - Which is a nasty problem, antagonistic to proper design

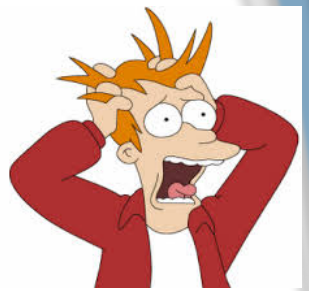
Observation

- Partitioned scheduling uses *processor affinities*
  - Constant through system lifetime
- When processor affinities are *arbitrary* (hence clusters may overlap) we have «*APA scheduling*»
  - More powerful than all other choices, but also completely indeterminate unless invariants are defined
  - *Weak APA invariant*: no running task is migrated (obviously sub-optimal)
  - *Strong APA invariant*: running tasks may be shifted to run elsewhere so that a ready task may run in its place (high runtime complexity)

Our choice /2

- Extending a single-core RTOS to a multi-core processor is complex (short of full re-design)
  - Because the RTOS data structures are centralized and memory is shared and not partitioned
- You immediately trip into the Giant Lock
  - To rid the RTOS of it is hard but needed
- For partitioned scheduling to scale you need to support *global resource sharing*
  - *Spin locking* is needed to preserve local prerogatives

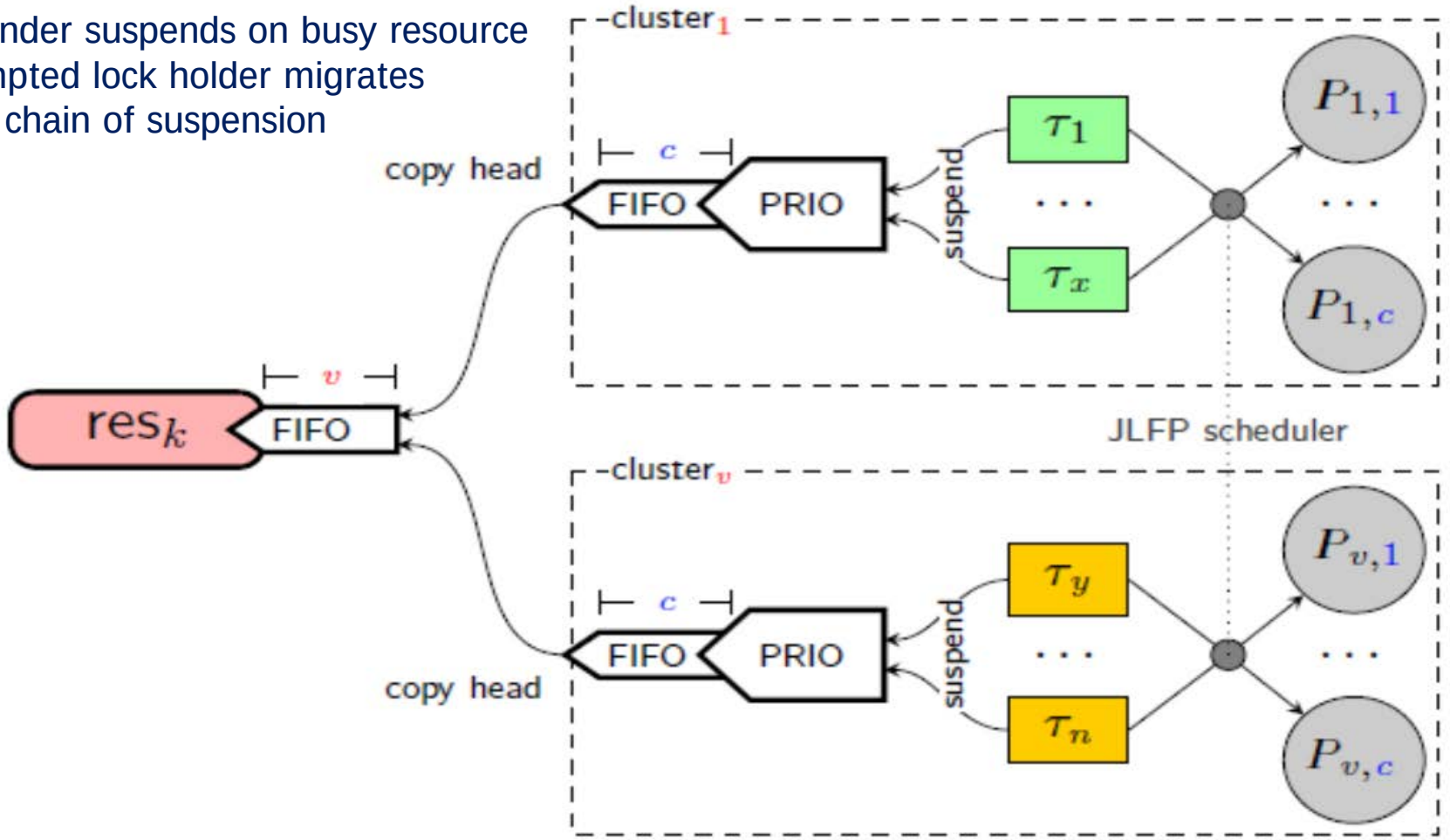
Our choice /3



- When global shared resources are used, *task migration* is needed
  - To prevent unbounded priority-inversion blocking
  - To preserve independence (those which do not share do not suffer)
- Two optimal solutions
  - O(m) Independence-preserving Protocol (**OMIP**)
  - **MrsP** (a pun on M-SRP) ☑

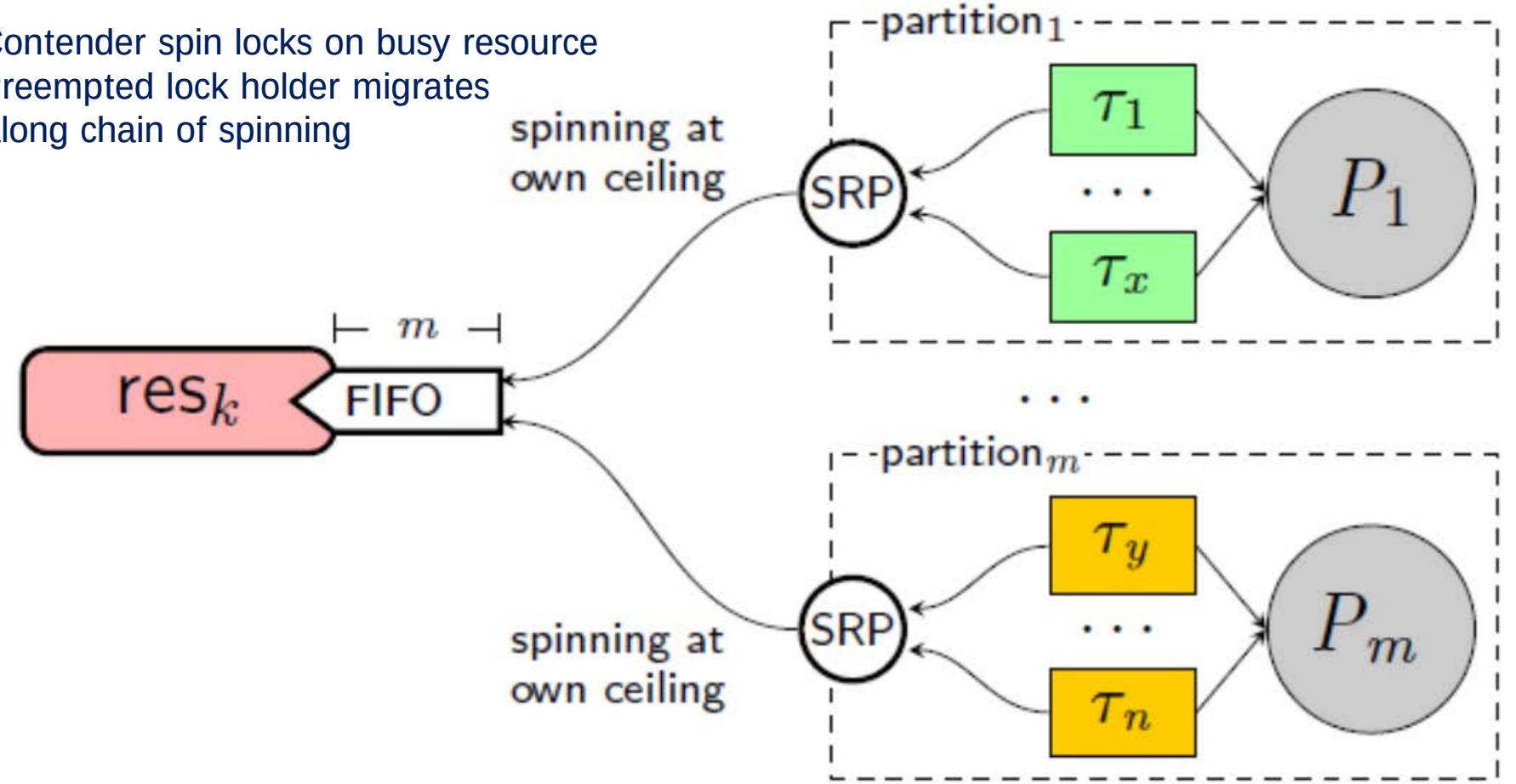
OMIP

Contender suspends on busy resource
Preempted lock holder migrates along chain of suspension



MrsP

Contender spin locks on busy resource
Preempted lock holder migrates along chain of spinning



Parallel libraries /1

- Time predictability is not a concern of mainstream solutions for parallel programming
- Their mind-set is generally wasteful
  - A distinction is to be drawn between worker threads and blocks candidate for parallel execution
    - Cumbersome outside of the RTOS
  - The general models make sense when you have *lots of cores* and can amortize the ensuing distributed overhead
- Two approaches
  - By libraries (costly and complex to port)
  - Built-in the programming language (to come)

UNIVERSITÀ DEGLI STUDI DI PADOVA

Parallel libraries /2

- For a quad-core processor to host parallel computation you need *cluster scheduling*
  - With cluster size $3 \leq c \leq 4$
- And a sound parallel runtime that sits well on the RTOS model of concurrency
  - That would have to be a well-designed subset of RTEMS SMP
- Porting existing libraries (a-la OpenMP) is a vast endeavour
  - Many times the size of this study