ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling

Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis

kernel-ghost@google.com

Why does kernel scheduling matter?

- Performance
  - Are processes getting a fair share of CPU time?
  - Prioritize latency-sensitive apps (e.g., key-value store) over throughput-oriented apps (e.g., video rendering, analytics, etc.).
- Security
  - Do not run two different customers’ threads in parallel on the same physical core (Spectre)
- System Stability
  - Periodically run system daemons to keep the system healthy (slab allocator, garbage collector)
  - These daemons should not interfere with other workloads.

What are the problems with existing schedulers?

- Kernel programming is difficult
  - Low-level languages
  - Hard to debug
  - Complicated synchronization (atomics, RCU, etc.)
  - Slow development
  - Manually port to new kernels or upstream to Linux (hard)
  - Slow to upgrade production machines (**restart required!**)

What are the problems with existing schedulers?

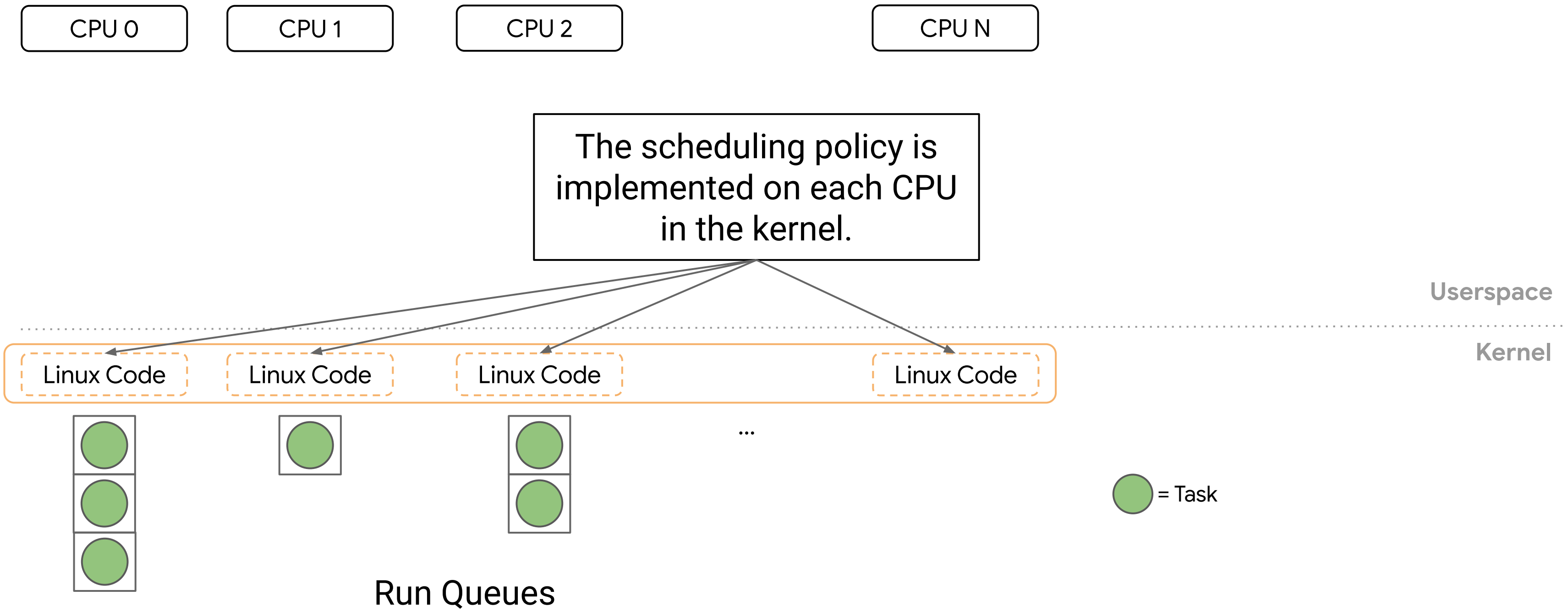
- Need to modify scheduling policies quickly
    - New classes of workloads
        - Low-latency/user-facing workloads
    - New hardware requires policy revamps
        - NUMA nodes, AMD CCX, hundreds of cores, GPUs, TPUs
    - Need to get the upgraded policies onto machines quickly
        - But doing a machine reboot makes this impossible...

What are the problems with existing schedulers?

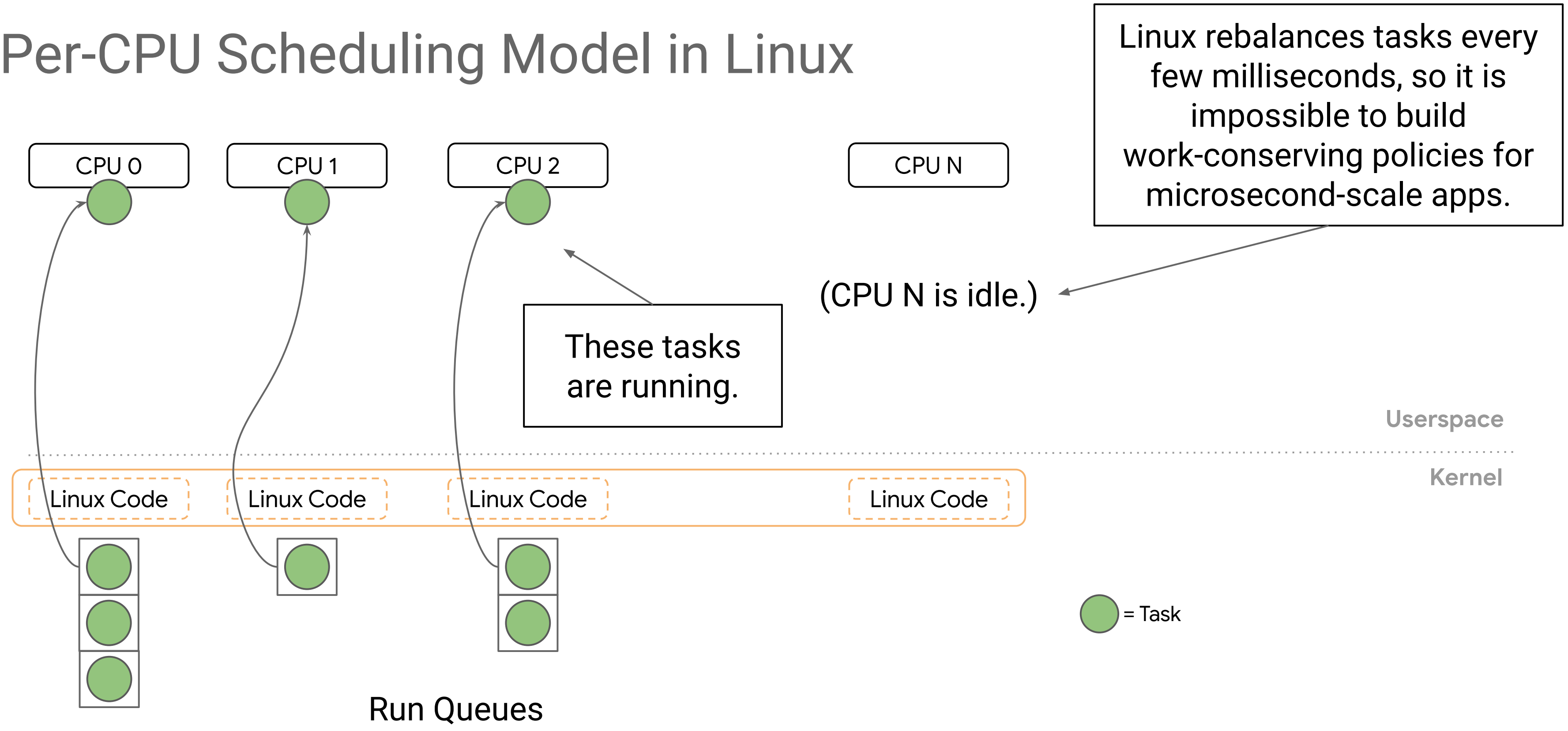
- Offers flexibility in policy only at a per-CPU level
  - Linux constrains policies to the **per-CPU model**
    - Does not support cross-CPU or cross-policy scheduling
  - Need **centralized model**
    - Work-conserving policies for microsecond-scale workloads
    - Shinjuku [NSDI’19], Shenango [NSDI’19], Caladan [OSDI’20]
  - Need **per-socket models (**Per-NUMA-node, Per-CCX)
  - Support multiple tenants on machines (hard to do efficiently with per-CPU models)

Google

Per-CPU Scheduling Model in Linux

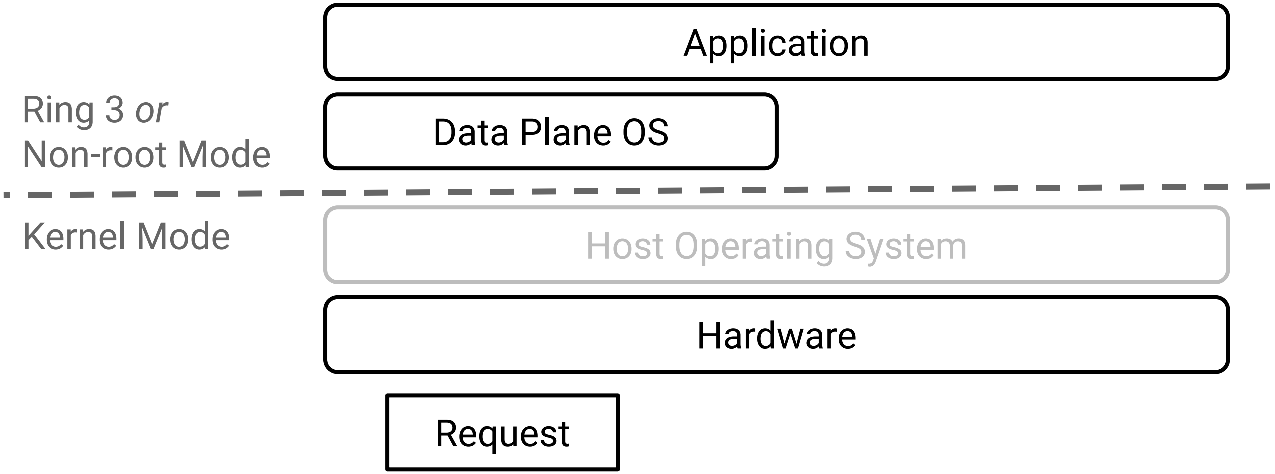


Per-CPU Scheduling Model in Linux



Data planes to the rescue?

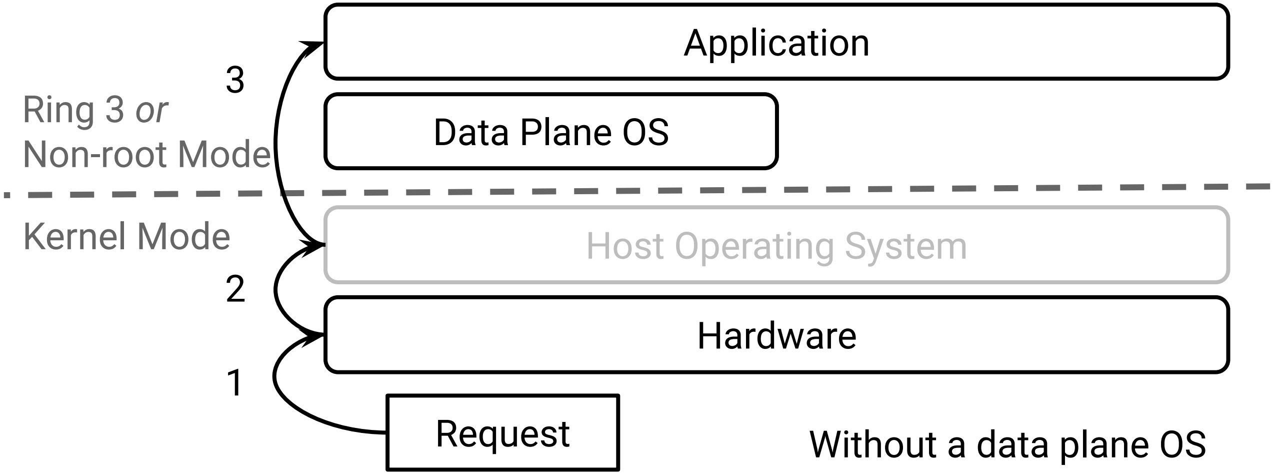
- Researchers move complexity into “container”-like data plane operating systems
  - IX [OSDI’14], ZygOS [SOSP’17], Shinjuku [NSDI’19], Shenango [NSDI’19], Caladan [OSDI’20], Google Snap [SOSP ‘19]
- Get the host kernel “out of the way”



Google

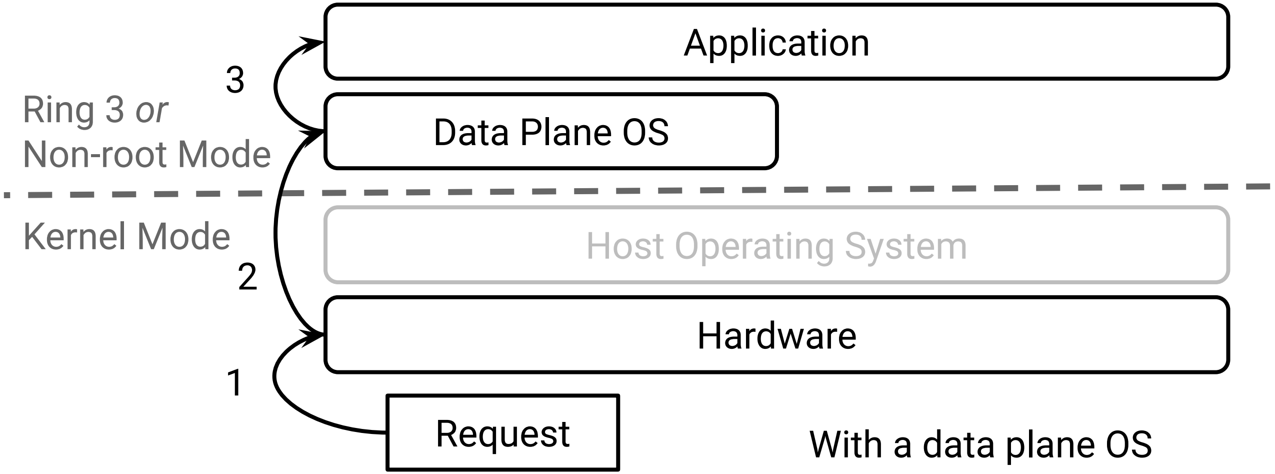
Data planes to the rescue?

- Researchers move complexity into “container”-like data plane operating systems
  - IX [OSDI’14], ZygOS [SOSP’17], Shinjuku [NSDI’19], Shenango [NSDI’19], Caladan [OSDI’20], Google Snap [SOSP ‘19]
- Get the host kernel “out of the way”



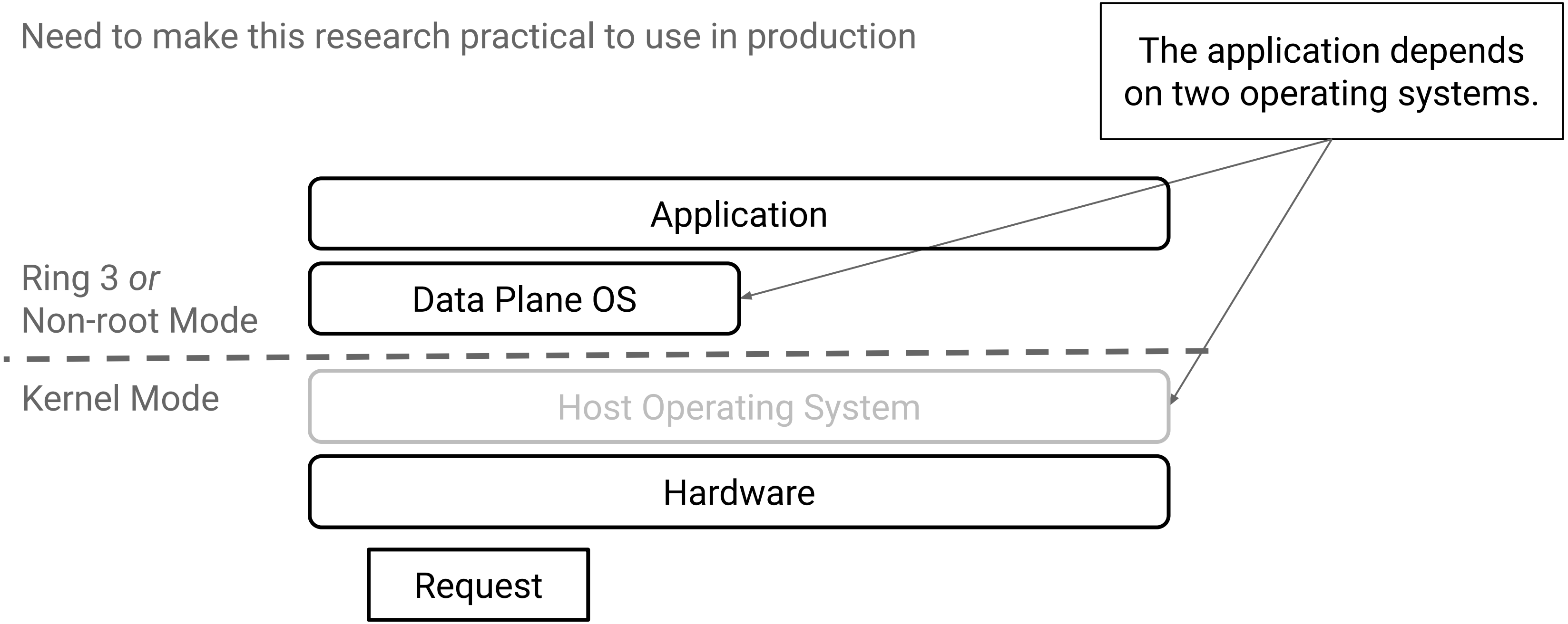
Data planes to the rescue?

- Researchers move complexity into “container”-like data plane operating systems
  - IX [OSDI’14], ZygOS [SOSP’17], Shinjuku [NSDI’19], Shenango [NSDI’19], Caladan [OSDI’20], Google Snap [SOSP ‘19]
- Get the host kernel “out of the way”



Data planes to the rescue?

- Need a data plane OS for every app and scheduling policy
  - Not feasible in a shared cloud environment
  - Need to make this research practical to use in production



Google

What should an ideal scheduler have?

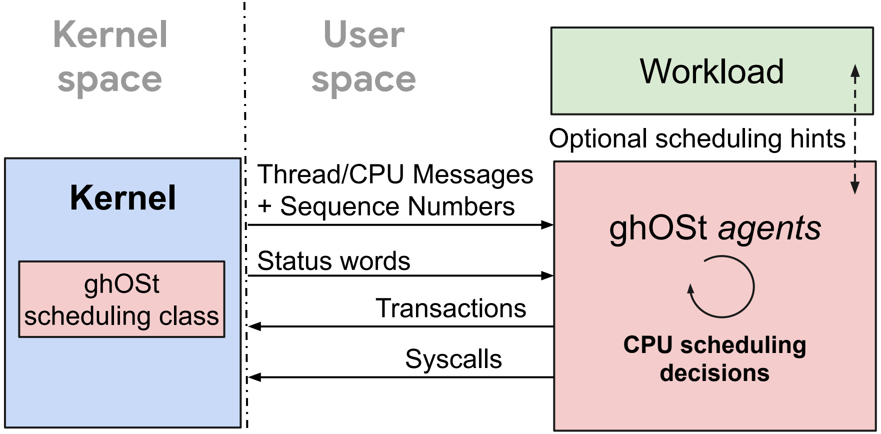
- Easy to implement policies and port across machines
- Optimize policies for a wide variety of targets
- Scheduling decision delegation
- Composition and partitioning
- Non-disruptive updates

Solution: ghOSt

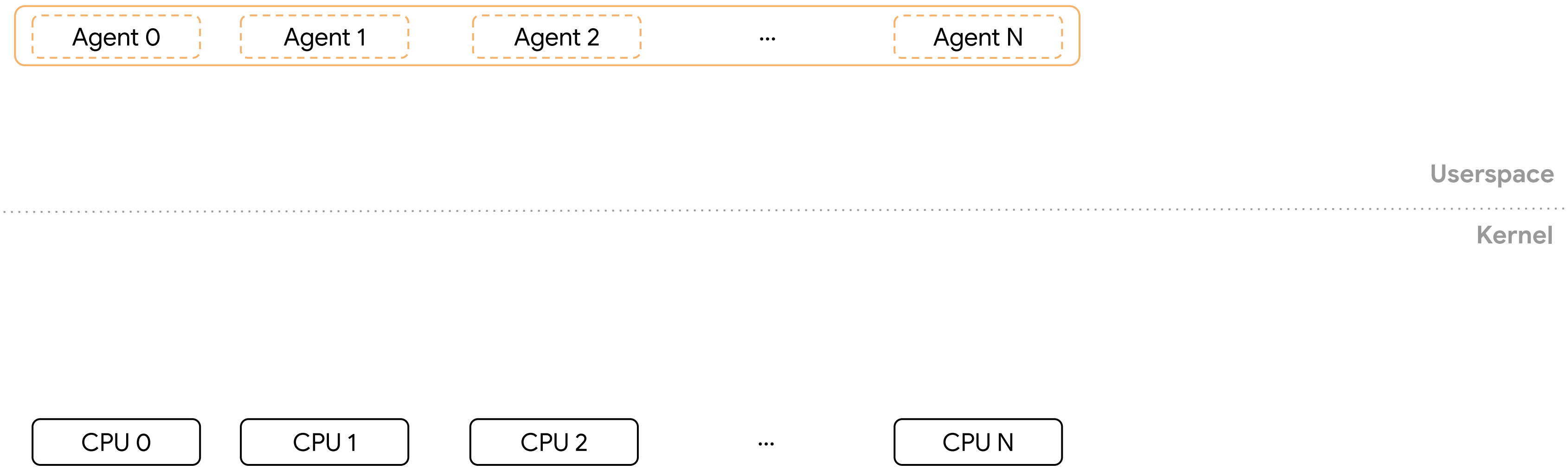
- New Linux kernel scheduler
- Runs scheduling policies in a userspace process
- Fast and flexible abstractions
- Supports a variety of scheduling policies
    - µs-scale workloads
    - Co-locate latency-sensitive apps with batch apps
    - Multi-tenant workloads
    - Centralized, partitioned, and per-CPU policies
- Upgrades are quick -- only a process restart required

ghOSt in a Nutshell

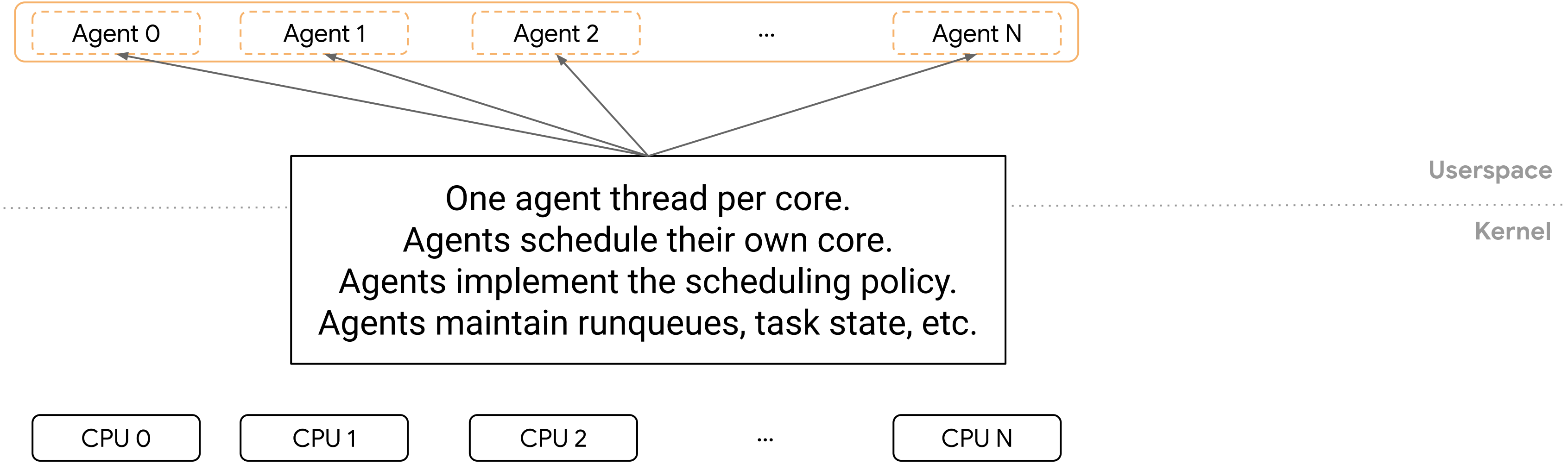
- Linux kernel scheduling class
- All scheduling policy runs in a userspace process
  - ghOSt is really just the scaffolding necessary to offload policies to userspace!
- The userspace process receives notifications about key events
  - E.g., task block, task yield, CPU timer tick, etc.
- Scheduling decisions committed to the kernel via transactions



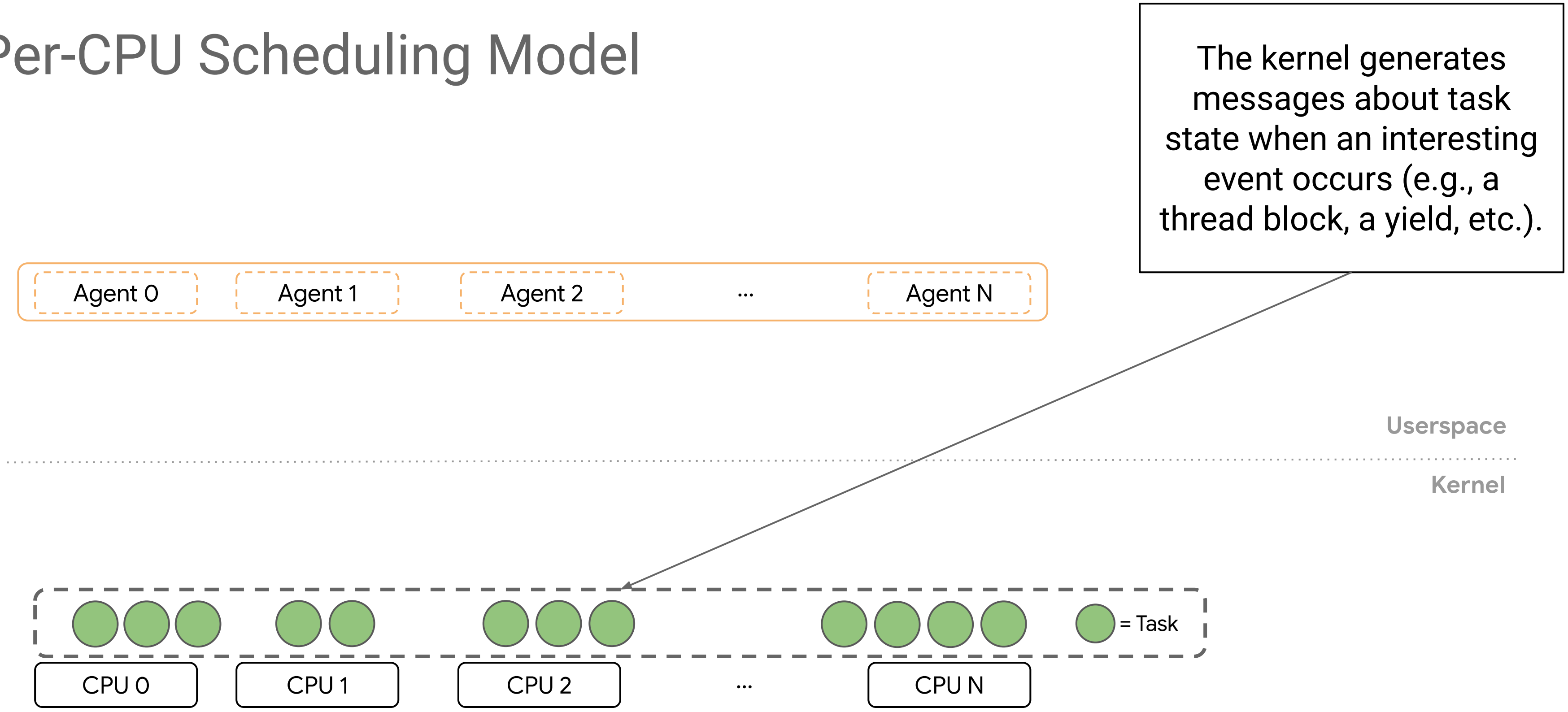
Per-CPU Scheduling Model



Per-CPU Scheduling Model

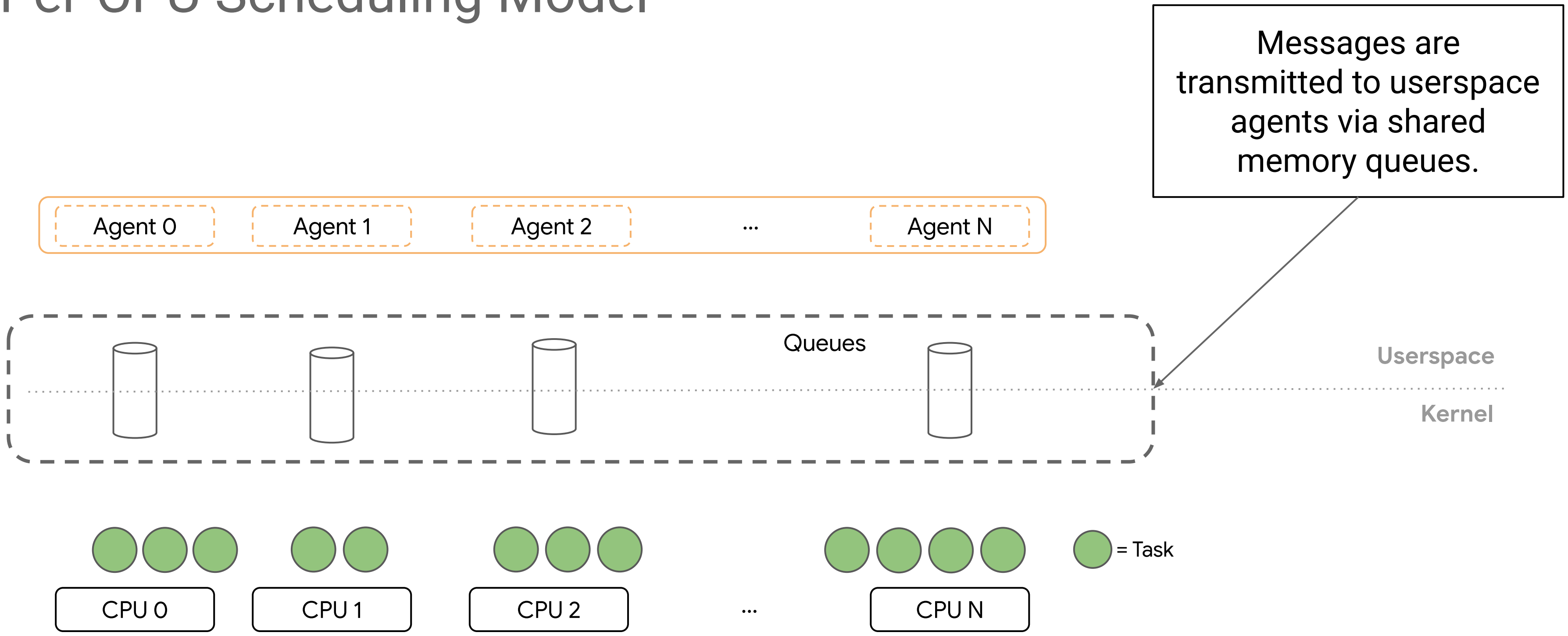


Per-CPU Scheduling Model

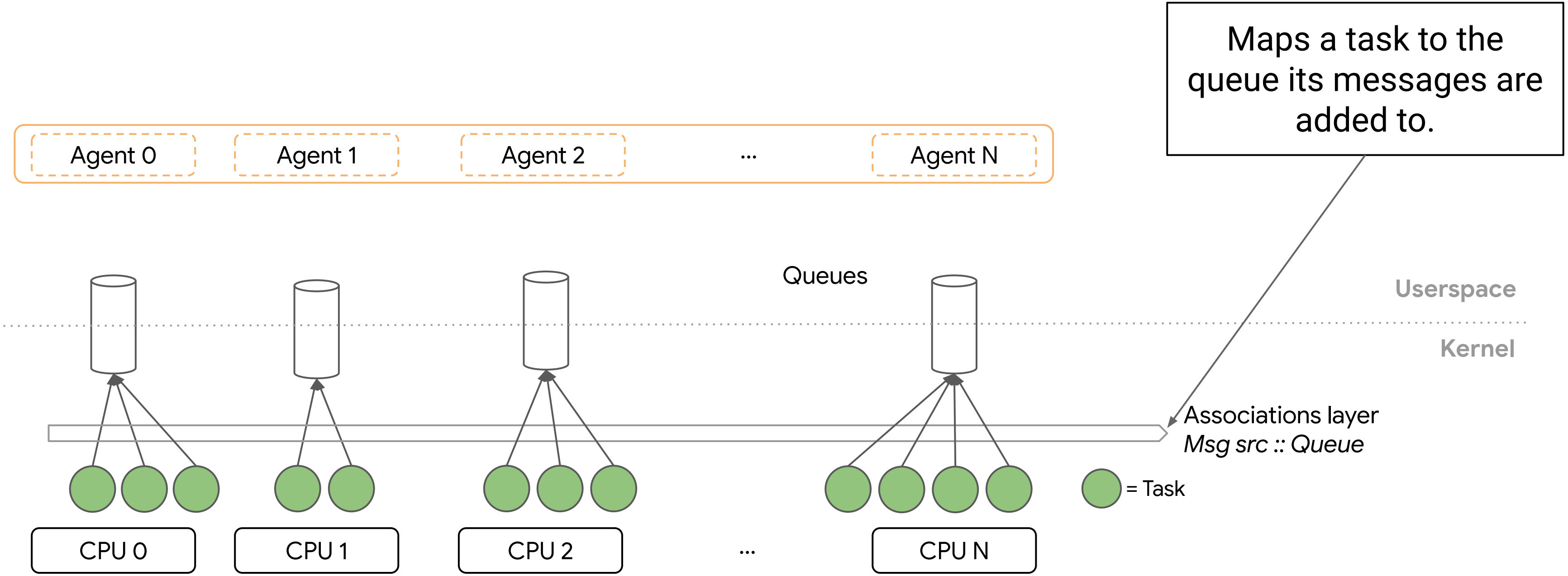


Task Messages: New, Blocked, Wakeup, Yield, Preempt, Departed, Dead

Per-CPU Scheduling Model

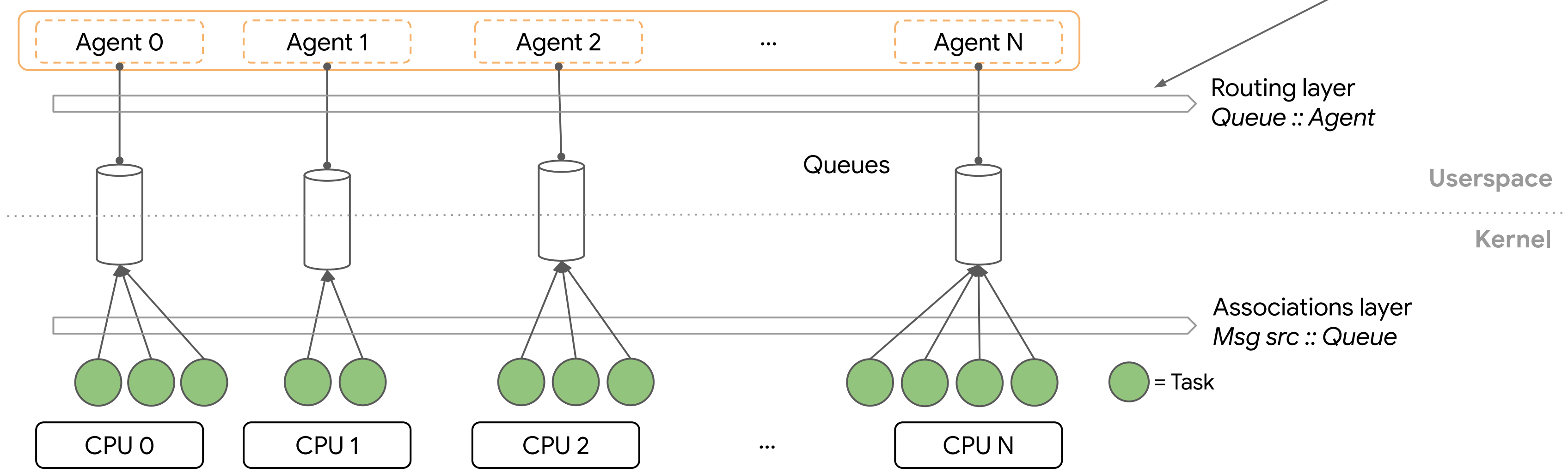


Per-CPU Scheduling Model



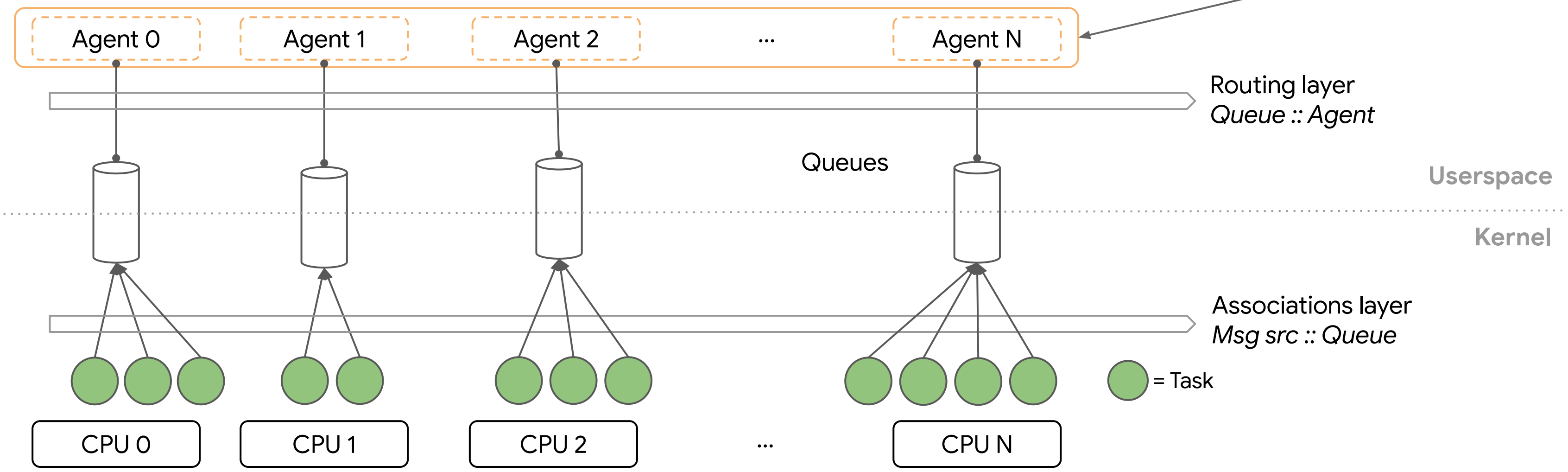
Per-CPU Scheduling Model

Determines which agent is woken up when a message is added to a queue.



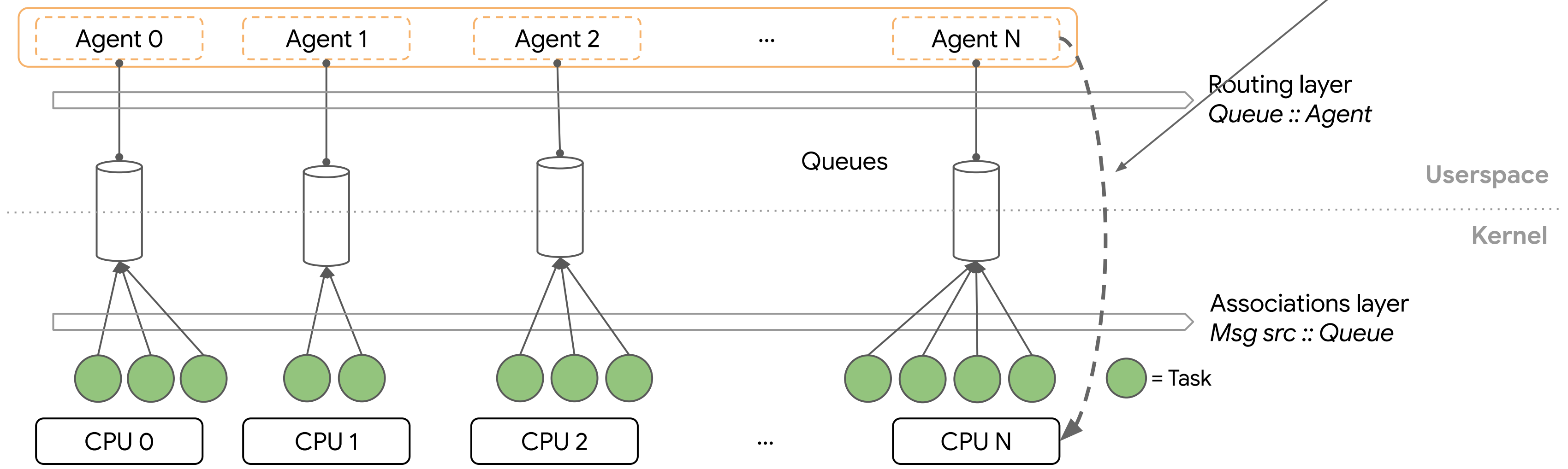
Per-CPU Scheduling Model

Reminder: The agents implement the scheduling policy.
They maintain runqueues, task state, etc.



Per-CPU Scheduling Model

The agent commits its scheduling decision to the kernel with a transaction.

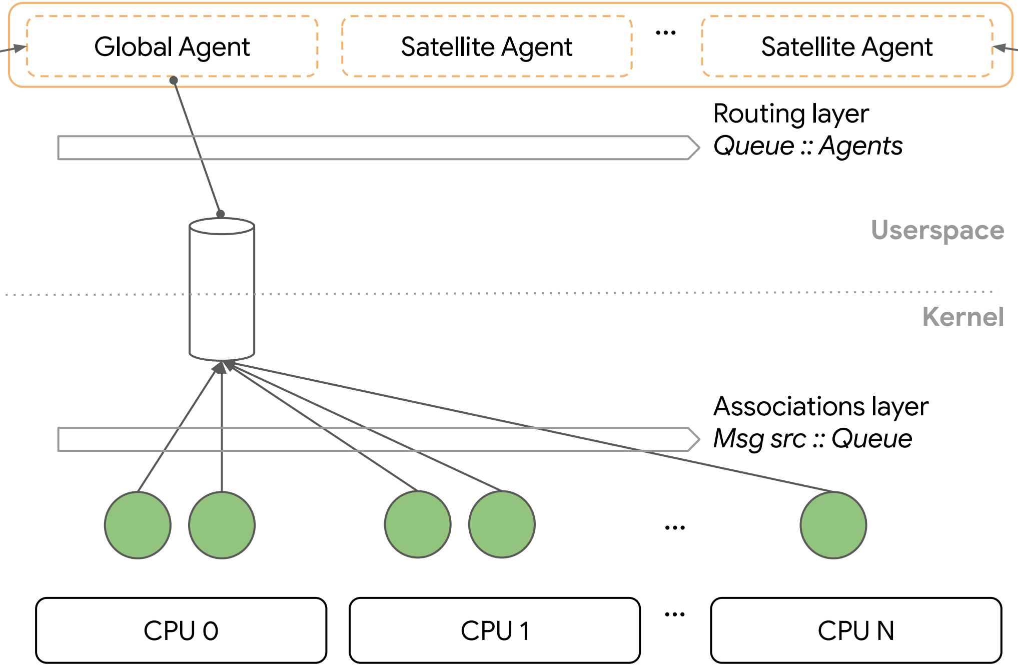


Centralized Scheduling Model

Only the Global Agent implements the scheduling policy.

Always awake and spins.

Schedules all CPUs in the machine.



The Satellite Agents are asleep and never wake up.

Transactions

- Agents commit scheduling decisions to the kernel via *transactions*
  - In each transaction, specify **TID** of thread being scheduled and **target CPU**
  - *Atomic* and *retractable*
    - Atomic: Multiple agents may be making decisions, state cleanup of transaction failure
    - Retractable: A decision could become invalid as OS state changes

TXN_CREATE()	• Create a transaction
TXNS_COMMIT()	• Commit one or more transactions • When >1 txns committed, use batch inter-processor interrupts (IPIs)
TXNS_RETRACT()	• Retract one or more transactions • May fail

Google

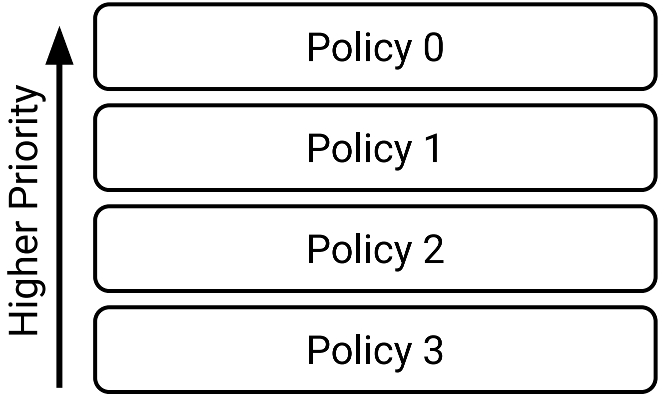
Synchronizing Agents with the Kernel

- The kernel is the **source of truth** and our **atomic store**
  - All task state lives in the kernel
  - This state is ephemeral and lives between agent restarts
  - Many ghOSt scheduler interactions happen in a context where we do not synchronously invoke an agent
- How can we keep agents in sync with the kernel?

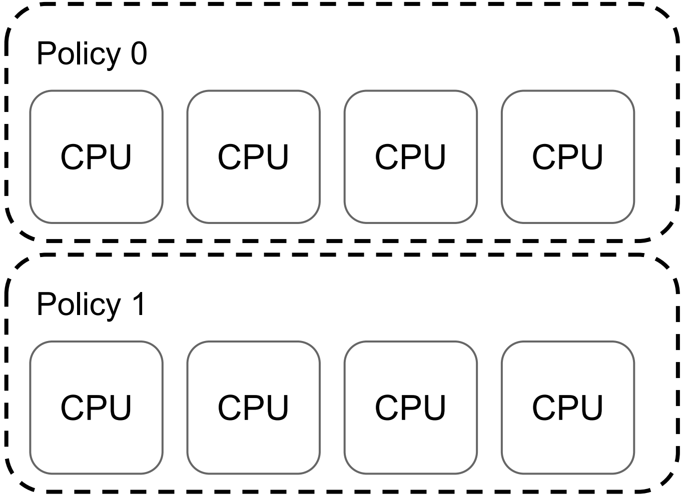
- Sequence numbers
  - Each task has a sequence number
  - When a task message is generated, the sequence number is incremented and included in the message.
  - When an agent opens a transaction, it includes the most recent sequence number.
  - If the sequence number in the transaction is outdated, the transaction fails.

Co-locating Policies

- Layering within an agent
    - Just like Linux
- Enclaves
    - Split CPUs up into groups and let each agent schedule its own group



Layering



Enclaves

Google

Quick Upgrades and Fault Tolerance

- Upgrades are fast (< 1 second)
  - Kill and restart agent in milliseconds
  - No machine reboot
- Schedule tasks *seamlessly* while the agent is down
  - Use simple in-kernel FIFO policy to keep apps alive
  - Could also kick tasks to CFS
- Recover state when the agent restarts

ghOSt is the scheduling solution for large cloud providers.

- Easy to implement policies and port across machines
- Optimize policies for a wide variety of targets
- Scheduling decision delegation
- Composition and partitioning
- Non-disruptive updates

Evaluation

Microbenchmarks

Syscall Overhead	72 ns
Message Delivery Overhead	265 ns
Local Commit	888 ns
Context Switch Overhead with Trivial Single-Task Kernel Scheduler	410 ns
CFS Context Switch Overhead	599 ns

Conclusion: ghOSt's API has similar overheads to other Linux schedulers, so it is practical to use for production scheduling, including for workloads with microsecond-scale requests.

Microbenchmarks

Remote Transaction Commit	
Committer Overhead	668 ns
Target CPU Overhead	1064 ns
End-to-End Latency	1772 ns

Remote Transactions Batch Commit (10 transactions)	
Committer Overhead	3964 ns (= 396 ns/transaction)
Target CPU Overhead	1821 ns
End-to-End Latency	5688 ns

Conclusion: Similarly, ghOSt has fairly low overheads for remote scheduling, making it practical for workloads with microsecond-scale requests.

2-socket Intel Xeon Platinum 8173M @ 2 GH, 28 cores per socket, 2 logical cores each, Linux 4.15

Lines of Code

CFS (kernel/sched/fair.c)	6,217 LOC
Kernel ghOSt Scheduling Class	3,777 LOC
Userspace Support Library	3,115 LOC
Google Snap Policy	855 LOC

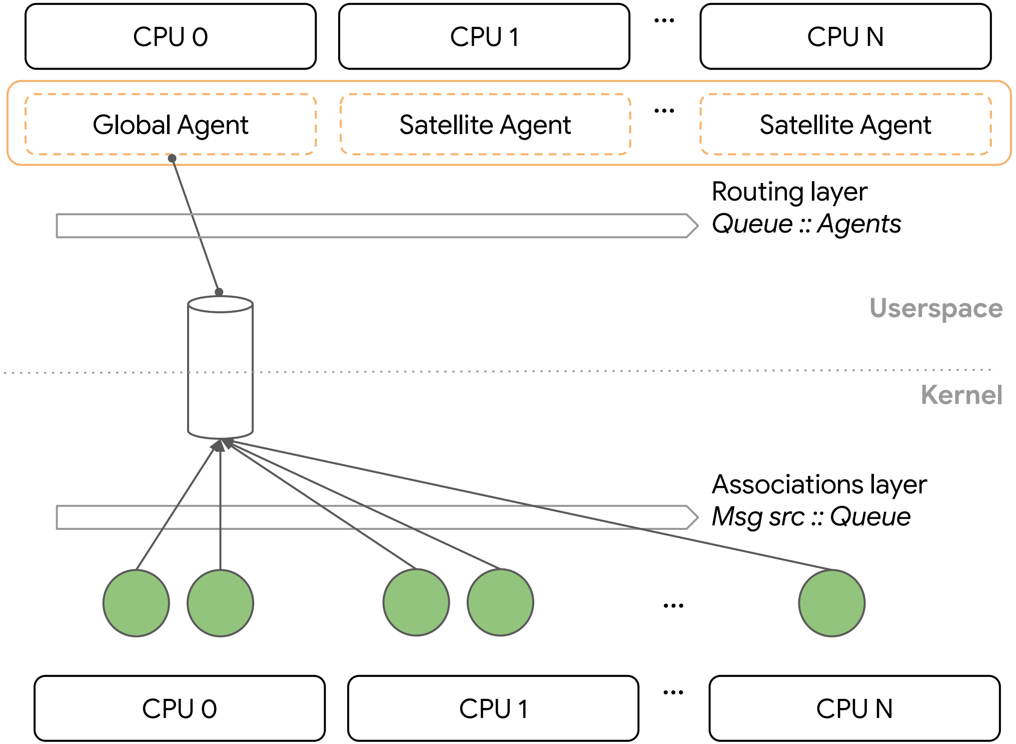
Google Snap

- Snap is our internal low-latency packet processing framework
- One main polling thread that processes network traffic
- Additional worker threads are spawned as needed when traffic increases

- We currently schedule Snap with MicroQuanta, a microsecond-scale real-time Linux kernel scheduler
- We compare with a centralized FIFO policy implemented in ghOSt

- We have one server and six clients
- Five clients send 64 kB messages, one client sends 64B messages

Centralized Scheduling Model

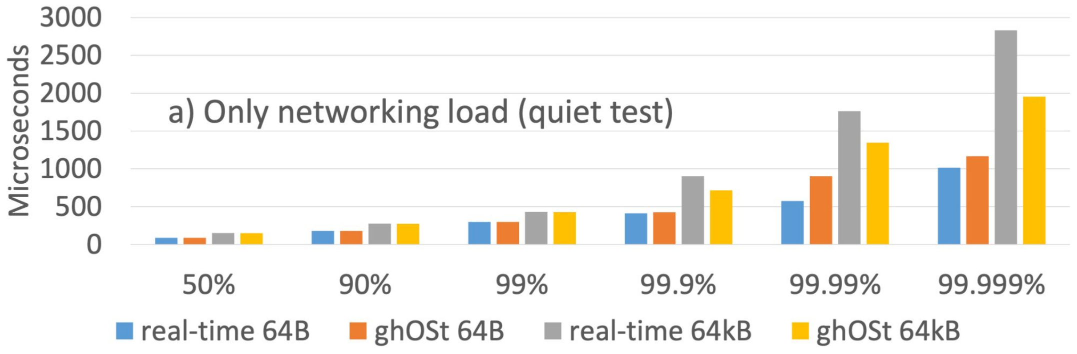


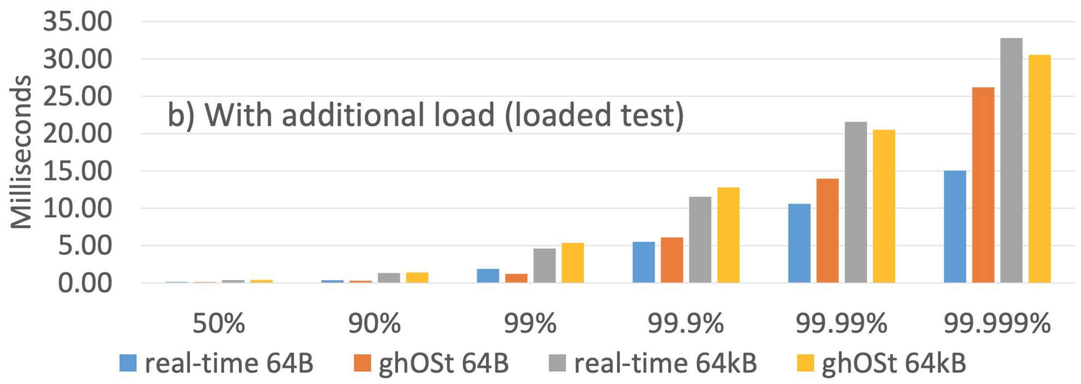
Centralized Model Accelerates Network Workloads

- **We do not have a centralized model in Linux schedulers today**

- The centralized model is much more responsive to network load changes
- Faster rebalancing across cores (μs-scale rather than ms-scale)
- Highly effective for μs-scale workloads (Shinjuku [NSDI'19], Shenango [NSDI'19], Caladan [OSDI'20], Google Snap [SOSP '19])

Google Snap Results





 2-socket Intel Xeon Platinum 8173M @ 2 GH, 28 cores per socket, 2 logical cores each, Linux 4.15

Future Work: NAPI Integration (+ elsewhere in the network stack)

- NAPI improves network performance by avoiding interrupts and dropping packets under high load
- Offloads packet processing to ksoftirqd
- Overall system performance is sensitive to the scheduler
    - Latency-sensitive applications + latency-sensitive networking are dependent on scheduling policy

- Use ghOSt to complement NAPI
- Implement ghOSt policies that are informed by application-specific and NAPI-specific context
    - Deadline scheduling, pipeline scheduling, better cache locality, etc.
    - ghOSt unlocks rapid prototyping since you do not need to recompile + restart kernel

Future Work

- Memory management
- New policies
- Tighter integration with other system stacks (e.g., networking stack).
- Formal Verification
  - Policy is isolated from complicated mechanisms. We can more easily prove formal properties on policies.
- eBPF Extensions
  - Accelerate more paths with eBPF. e.g., use eBFP to run custom policies when the agent process is down.
  - Use eBPF to generate better debugging and profiling tools.

Summary

- ghOSt is a new Linux kernel scheduler that is production-ready
- Runs scheduling policies in a userspace process
- Fast and flexible abstractions
- Supports a variety of scheduling policies with good performance on production workloads
- Upgrades are quick -- only a process restart required

ghOSt

- ghOSt is open source (both the kernel and userspace components)
- We hope you use it in your systems!

- Questions: kernel-ghost@google.com
- Come work with us! Reach out to the same email address.