CS 477:

Advanced Operating Systems

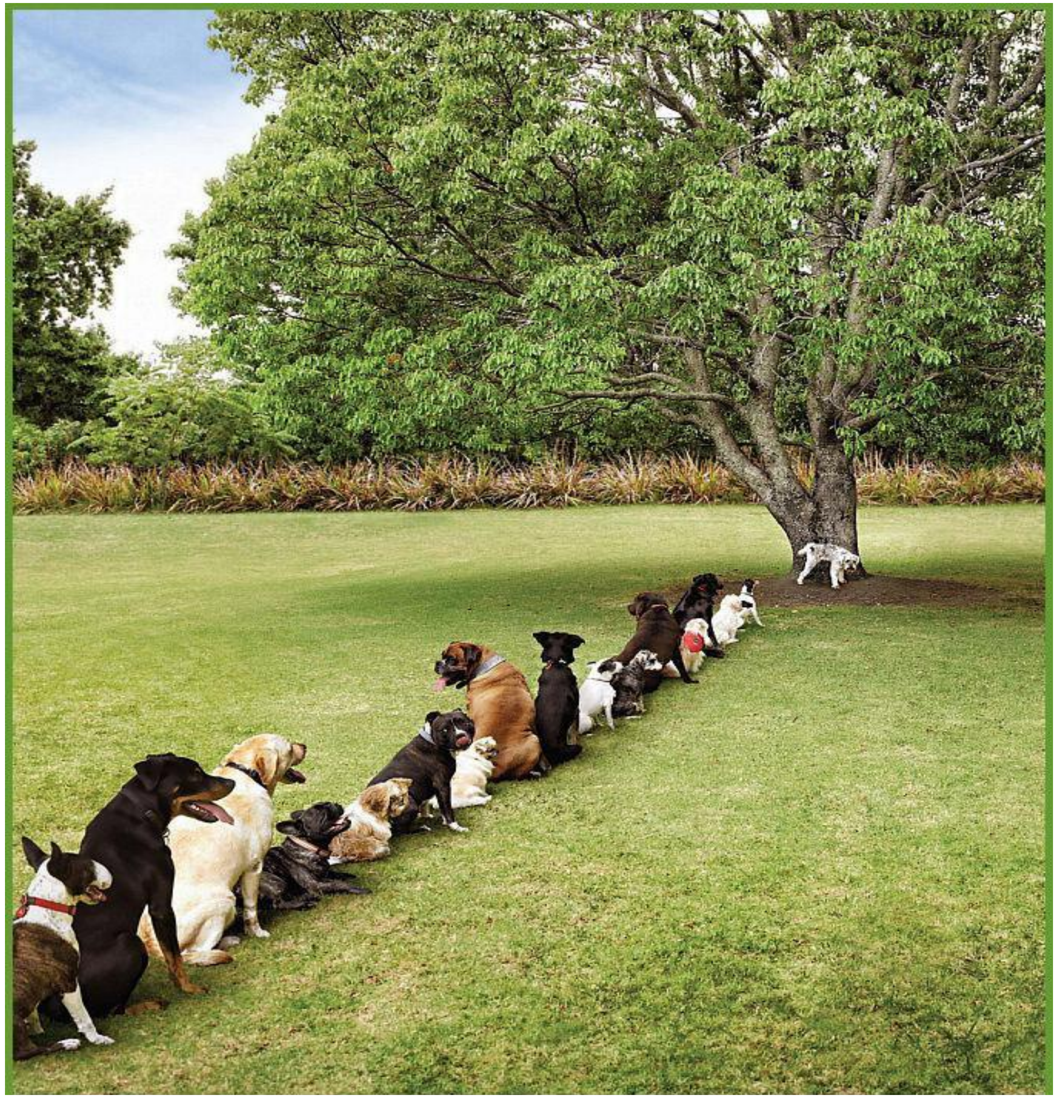
Scheduling II

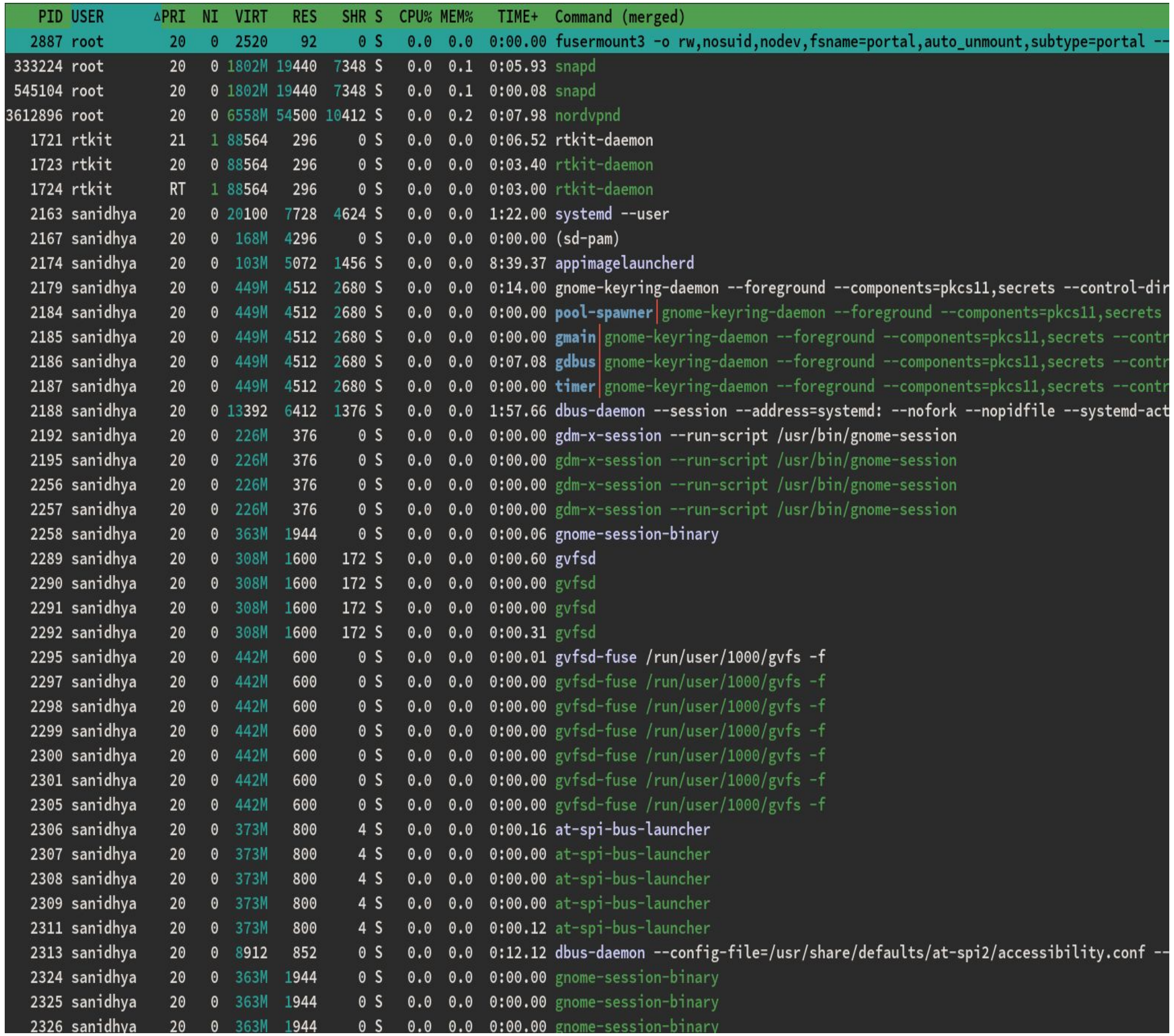
Administrivia

This week

- EEVDF scheduling
- Delegating scheduling to user space with GhOSt

Focus of today’s lecture: Scheduling …





PID	USER	ΔPRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command (merged)
2887	root	20	0	2520	92	0	S	0.0	0.0	0:00.00	fusermount3 -o rw,nosuid,nodev,fsname=portal,auto_unmount,subtype=portal --
333224	root	20	0	1802M	19440	7348	S	0.0	0.1	0:05.93	snapd
545104	root	20	0	1802M	19440	7348	S	0.0	0.1	0:00.08	snapd
3612896	root	20	0	6558M	54500	10412	S	0.0	0.2	0:07.98	nordvpnd
1721	rtkit	21	1	88564	296	0	S	0.0	0.0	0:06.52	rtkit-daemon
1723	rtkit	20	0	88564	296	0	S	0.0	0.0	0:03.40	rtkit-daemon
1724	rtkit	RT	1	88564	296	0	S	0.0	0.0	0:03.00	rtkit-daemon
2163	sanidhya	20	0	20100	7728	4624	S	0.0	0.0	1:22.00	systemd --user
2167	sanidhya	20	0	168M	4296	0	S	0.0	0.0	0:00.00	(sd-pam)
2174	sanidhya	20	0	103M	5072	1456	S	0.0	0.0	8:39.37	appimagelauncherd
2179	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:14.00	gnome-keyring-daemon --foreground --components=pkcs11,secrets --control-dir
2184	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:00.00	pool-spawner gnome-keyring-daemon --foreground --components=pkcs11,secrets
2185	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:00.00	gmain gnome-keyring-daemon --foreground --components=pkcs11,secrets --contr
2186	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:07.08	gdbus gnome-keyring-daemon --foreground --components=pkcs11,secrets --contr
2187	sanidhya	20	0	449M	4512	2680	S	0.0	0.0	0:00.00	timer gnome-keyring-daemon --foreground --components=pkcs11,secrets --contr
2188	sanidhya	20	0	13392	6412	1376	S	0.0	0.0	1:57.66	dbus-daemon --session --address=systemd: --nofork --nopidfile --systemd-act
2192	sanidhya	20	0	226M	376	0	S	0.0	0.0	0:00.00	gdm-x-session --run-script /usr/bin/gnome-session
2195	sanidhya	20	0	226M	376	0	S	0.0	0.0	0:00.00	gdm-x-session --run-script /usr/bin/gnome-session
2256	sanidhya	20	0	226M	376	0	S	0.0	0.0	0:00.00	gdm-x-session --run-script /usr/bin/gnome-session
2257	sanidhya	20	0	226M	376	0	S	0.0	0.0	0:00.00	gdm-x-session --run-script /usr/bin/gnome-session
2258	sanidhya	20	0	363M	1944	0	S	0.0	0.0	0:00.06	gnome-session-binary
2289	sanidhya	20	0	308M	1600	172	S	0.0	0.0	0:00.60	gvfsd
2290	sanidhya	20	0	308M	1600	172	S	0.0	0.0	0:00.00	gvfsd
2291	sanidhya	20	0	308M	1600	172	S	0.0	0.0	0:00.00	gvfsd
2292	sanidhya	20	0	308M	1600	172	S	0.0	0.0	0:00.31	gvfsd
2295	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.01	gvfsd-fuse /run/user/1000/gvfs -f
2297	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2298	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2299	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2300	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2301	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2305	sanidhya	20	0	442M	600	0	S	0.0	0.0	0:00.00	gvfsd-fuse /run/user/1000/gvfs -f
2306	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.16	at-spi-bus-launcher
2307	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.00	at-spi-bus-launcher
2308	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.00	at-spi-bus-launcher
2309	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.00	at-spi-bus-launcher
2311	sanidhya	20	0	373M	800	4	S	0.0	0.0	0:00.12	at-spi-bus-launcher
2313	sanidhya	20	0	8912	852	0	S	0.0	0.0	0:12.12	dbus-daemon --config-file=/usr/share/defaults/at-spi2/accessibility.conf --
2324	sanidhya	20	0	363M	1944	0	S	0.0	0.0	0:00.00	gnome-session-binary
2325	sanidhya	20	0	363M	1944	0	S	0.0	0.0	0:00.00	gnome-session-binary
2326	sanidhya	20	0	363M	1944	0	S	0.0	0.0	0:00.00	gnome-session-binary

Scheduler strives to achieve …

- **Fairness:** Everyone should get some CPU
- **Optimization:** Make optimal use of system resources, minimize critical sections
- **Low overhead:** Should run for as short as possible
- **Generalizable:** Should work on every architecture, for every workload etc.

Issues with CFS

- Difficult to experiment: recompile + reboot + rewarm caches
- Very complex, often takes O(years) for people to fully understand
- Generalizable scheduler
  - Often leaves some performance on the table for some workloads / architectures
  - Impossible to make everyone happy all the time
- Filled with heuristics

Issues with CFS

- CFS focuses mostly on fair allocation of CPUs time
- Does not properly considers:
  - Task migration cost among CPUs
  - Power management
  - Hybrid scheduling: fast and slow cores
  - Latency requirement: requiring CPU time as fast as possible
    - No possible way to express latency requirements

EEVDF scheduler

- Earliest virtual deadline first
  - Similar to earliest deadline first (EDF)
- Divides the available CPU time fairly among contending tasks
- **Lag**: expected virtual run time - actual running time of a task
  - Positive lag: A task is owed CPU time
  - Negative lag: A task has received more than its share
- **Eligible** task: lag >= 0
- Maintains an invariant: **sum of all the lag value in the system is zero**
- Scheduler chooses a task from a set of eligible tasks

EEVDF scheduler

- Before running a task, scheduler computes **virtual deadline**
  - Add the time remaining in the timeslice to the time it became eligible
  - Longer time slice: later virtual deadline (will run later)
  - Shorter time slice: Run first (denotes latency sensitive tasks)

EEVDF scheduler: Example

- 3 CPU-bound tasks start at the same time
- Over those 30 ms, each task should run 10 ms
- Scheduler picks A and runs with a 30 ms timeslice

	A	B	C
Lag:	0	0	0

EEVDF scheduler: Example

- 3 CPU-bound tasks start at the same time
- Over 30 ms, each task should run 10 ms
- Scheduler picks A and runs with a 30 ms timeslice
- A is no longer eligible, B is picked up and runs 30 ms

	A	B	C
Lag:	-20	10	10

EEVDF scheduler: Example

- 3 CPU-bound tasks start at the same time
- Over 30 ms, each task should run 10 ms
- Scheduler picks A and runs with a 30 ms timeslice
- A is no longer eligible, B is picked up and runs 30 ms
- Now, only C is eligible, which will run next

	A	B	C
Lag:	-10	-10	20

EEVDF scheduler: Handling sleeping tasks

- Scheduler calculates **lag** only for **runnable tasks**
- Lag decays over virtual run time
- Uses deferred lag decay approach:
  - Resetting immediately can affect fairness and introduce starvation
- Lag forgiveness for long sleeping tasks
  - Long-sleeping tasks have their -ve lag forgiven as virtual time passes
  - Sleeping tasks with -ve lag are first placed in deferred queue without scheduling them for execution, once zero/+ve, they are scheduled out
- Positive lag is retained indefinitely

EEVDF scheduler: Time-slice control

- Preemption based on virtual deadlines
  - Enables running short time-slice tasks sooner
  - Short time-slice task can now preempt an already running task
- Tasks can specify desired time slice (100us–100ms)
  - Use **sched_setattr()** system call

This week

- EEVDF scheduling
- Delegating scheduling to user space with GhOSt

Why does kernel scheduling matter?

- Performance
  - Are processes getting fair share of CPU time?
  - Prioritize latency-sensitive applications (eg., kv store) over throughput oriented apps (eg., video encoding, analytics, etc)
- Security
  - Do not run two diff customers' threads in parallel on the same physical core (Spectre attack)
- 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 upstreams to Linux (hard)
  - Slow to upgrade production machines (**restart required**)

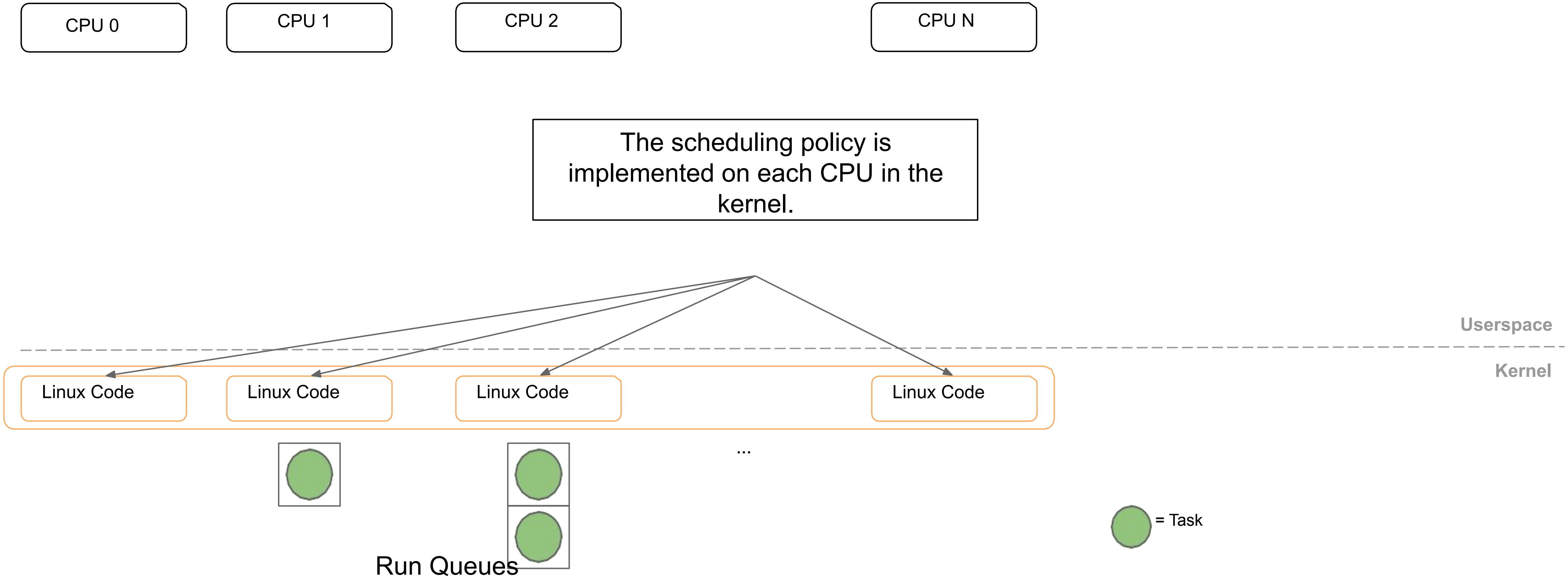
What are the problems with existing schedulers?

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

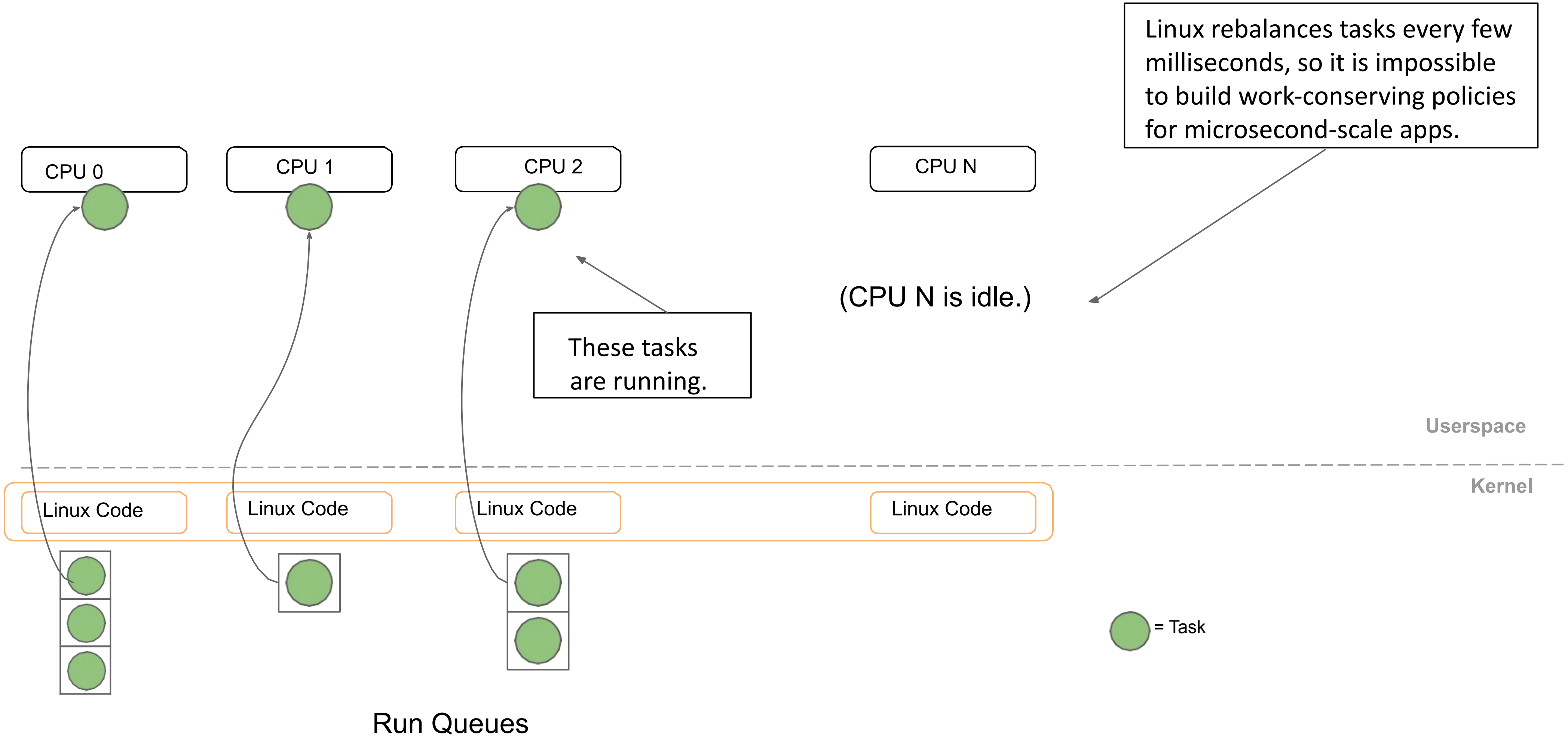
What are the problems with existing schedulers?

- 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
  - Need **per-socket models** (per-NUMA node, per-CCX)
  - Support multiple tenants on machines (hard to do efficiently with per-CPU models)

Per-CPU scheduling model in Linux

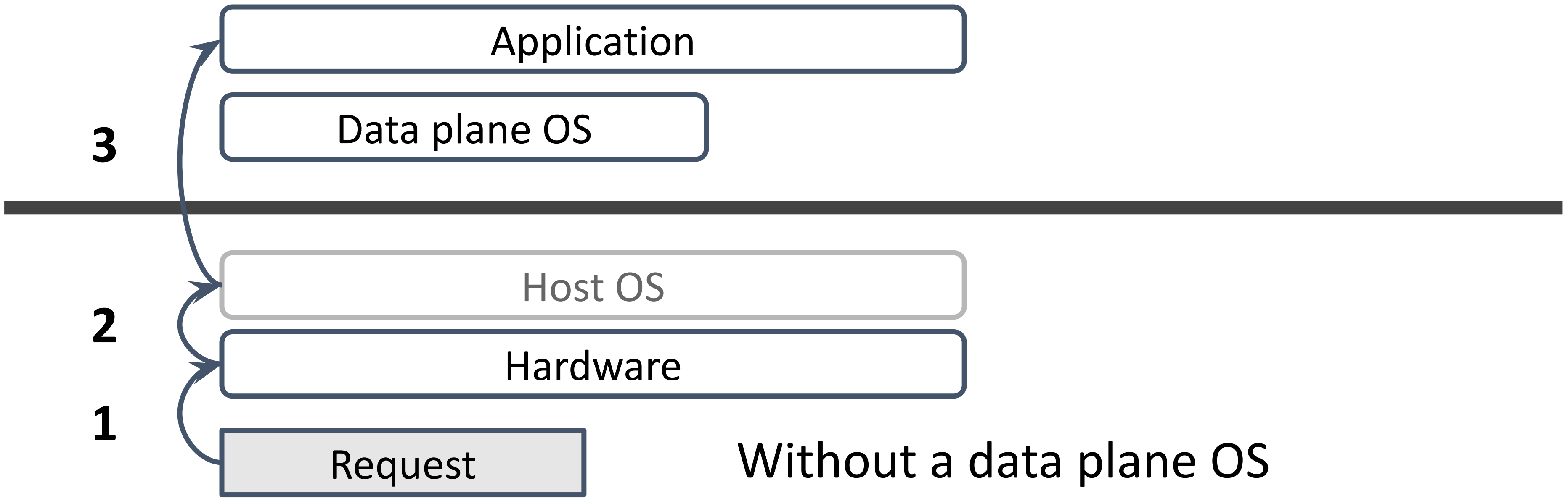


Per-CPU scheduling model in Linux



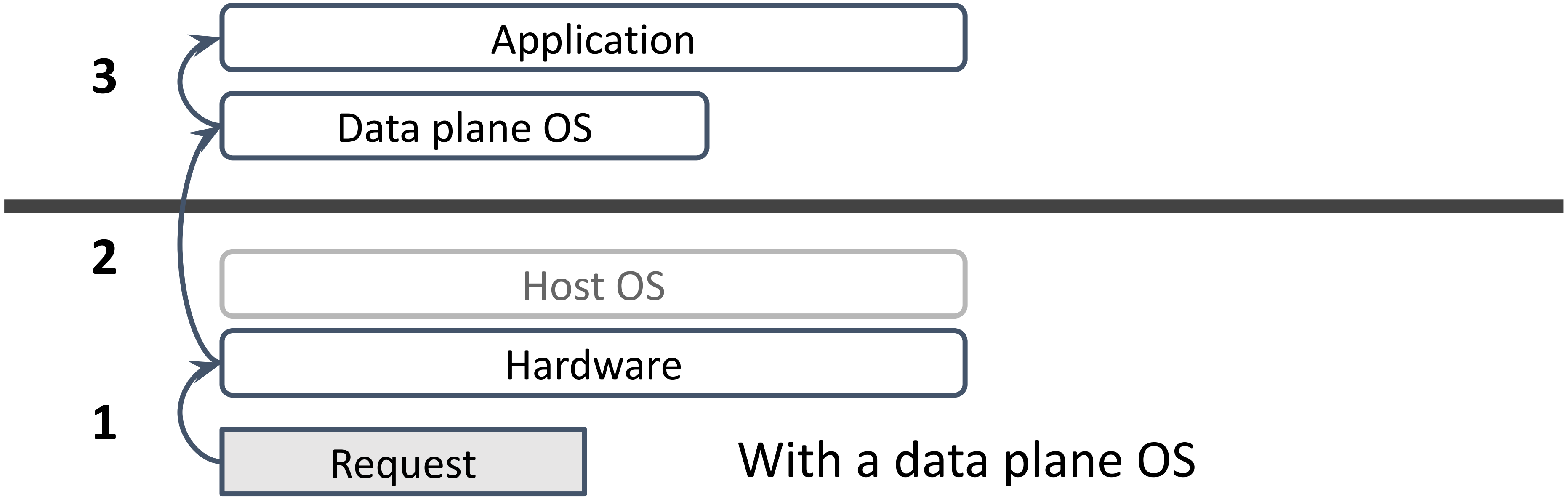
Data planes to the rescue?

- Researchers move complexity into “container”-like data plane OSes
- Get the host kernel “out of the way”



Data planes to the rescue?

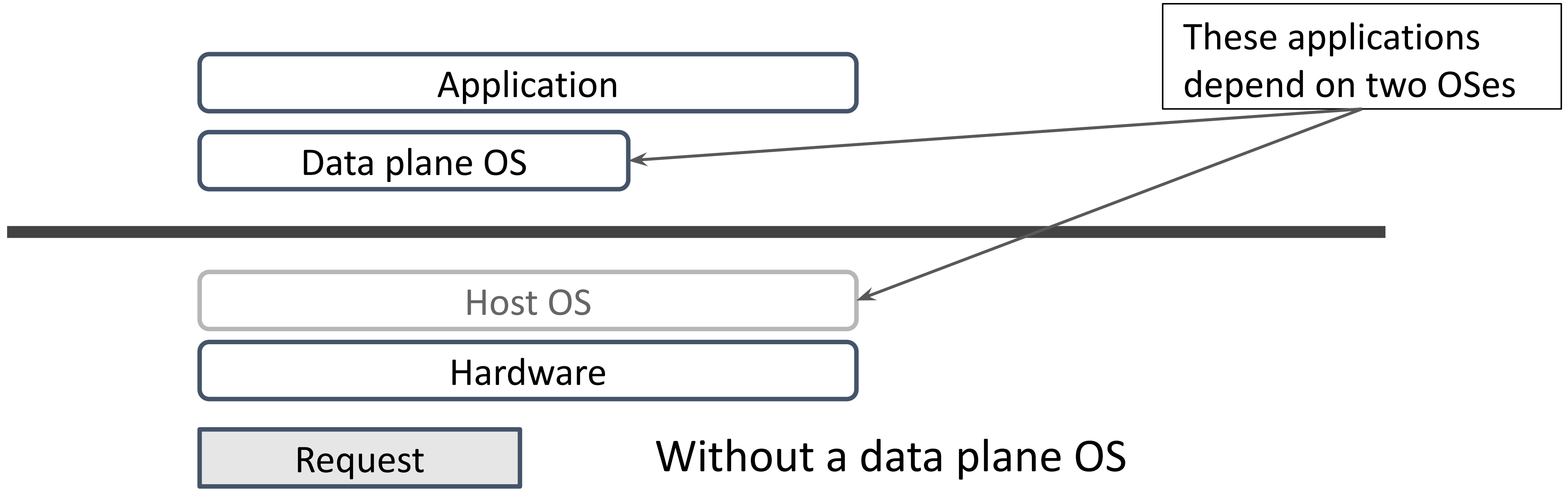
- Researchers move complexity into “container”-like data plane OSes
- Get the host kernel “out of the way”



With a data plane OS

Data planes to the rescue?

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



Requirements of an ideal scheduler

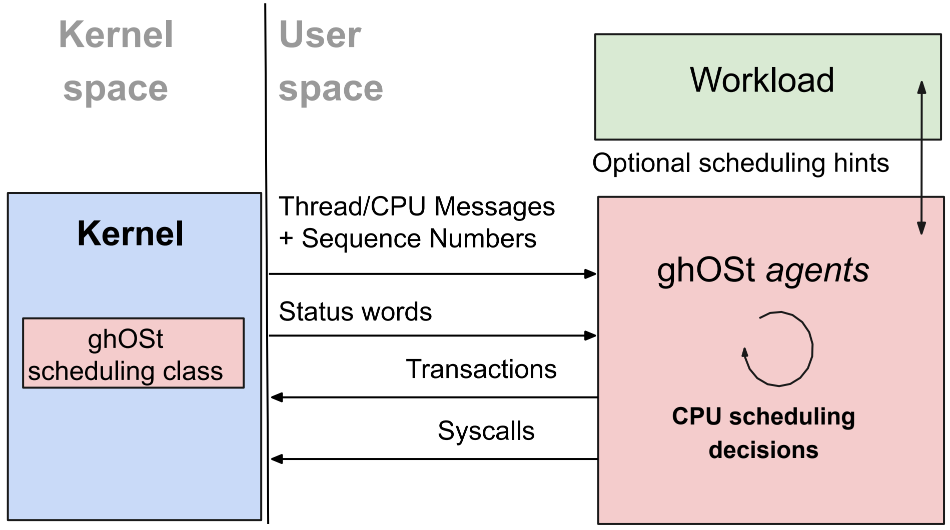
- Easy to implement policies and port across machines
- Optimize policies for a wide variety of targets
- Scheduling decision delegation
- Composition and partitioning
- Non-disruptive updates

Solution: ghOSt

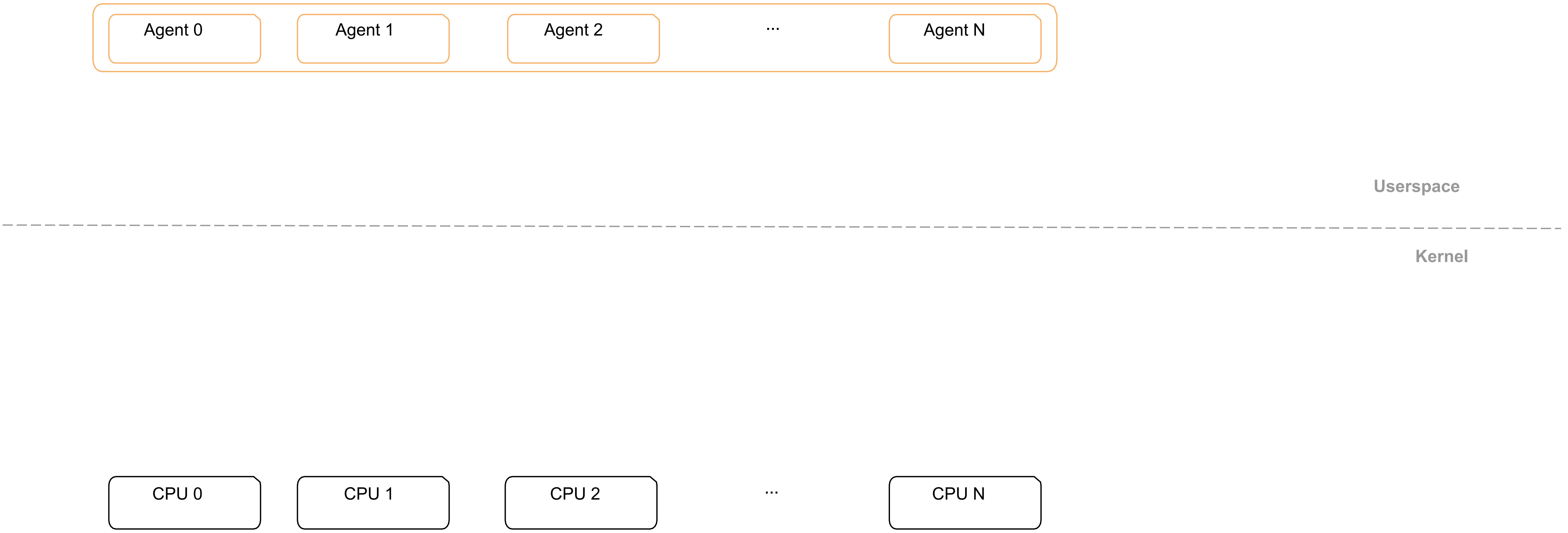
- New Linux kernel scheduler
- Runs scheduling policies in a user space process
- Fast and flexible abstractions
- Supports a variety of scheduling policies
  - Microsecond scale workloads
  - Co-locate latency-sensitive applications with batch ones
  - Multi-tenant workloads
  - Centralized, partitioned, and per-CPU policies
- Upgrades are quick: only a process restart

ghOSt in a nutshell

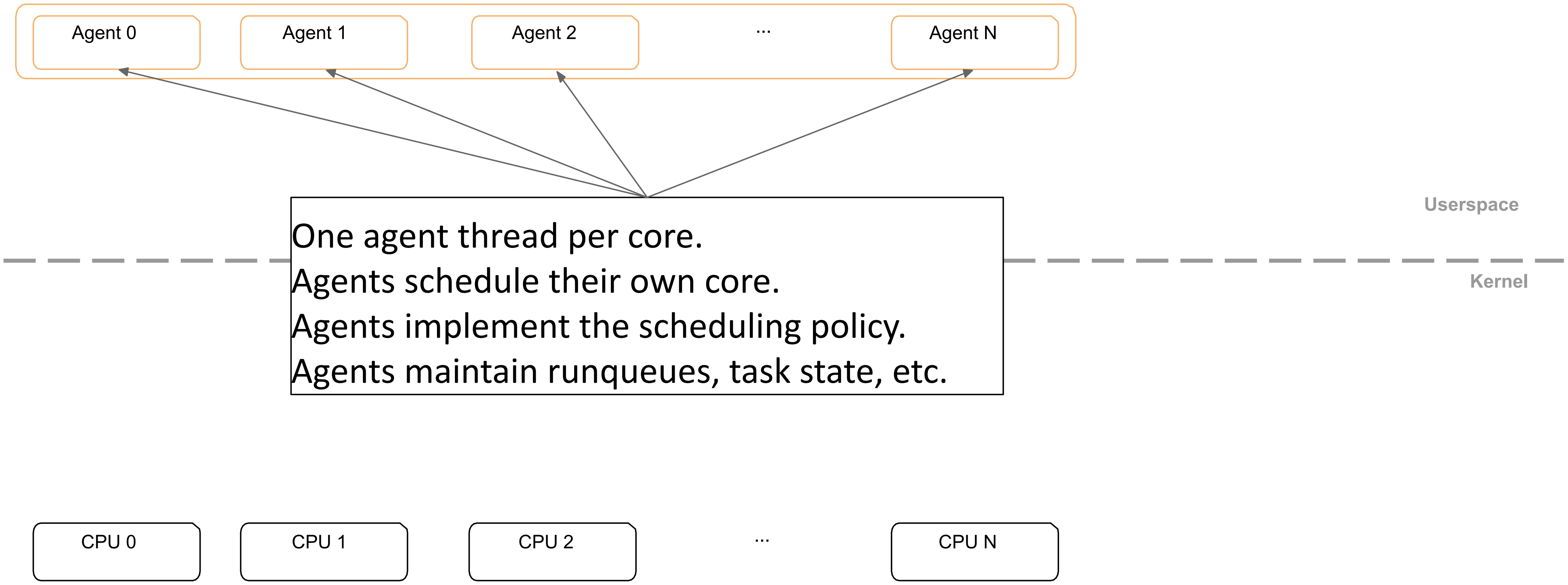
- Linux kernel scheduling class
- All scheduling policy runs in a user space process
  - ghOSt is a scaffolding necessary to offload policies to user space
- The user space process receives notifications about key events
  - E.g, task block, task yield, CPU timer, tick, etc.
- Scheduling decisions committed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



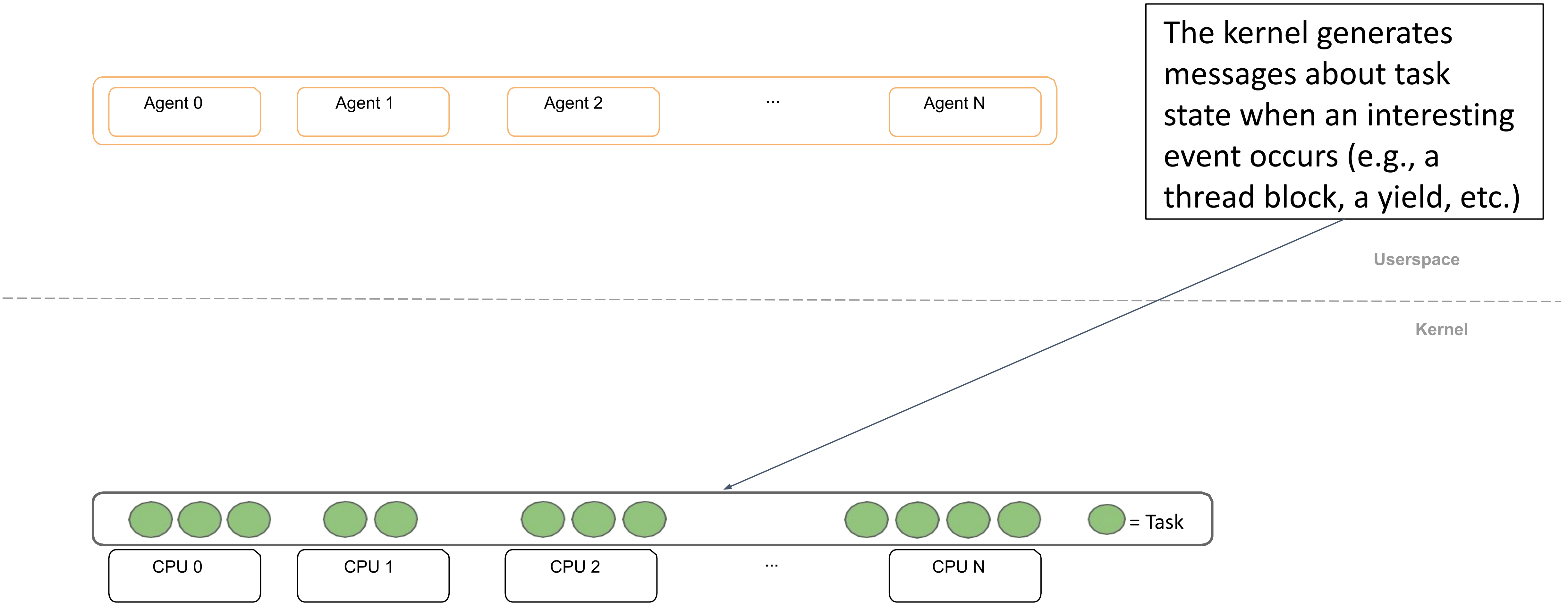
Per-CPU scheduling model



Per-CPU scheduling model

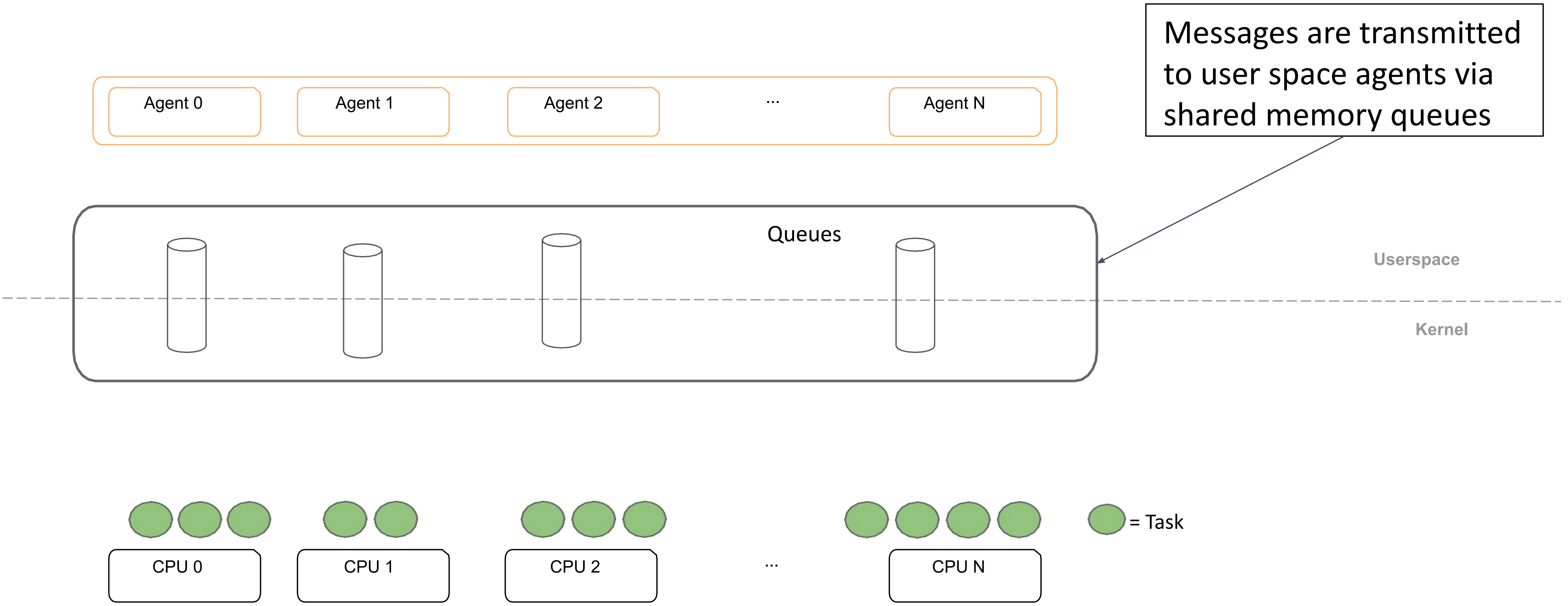


Per-CPU scheduling model

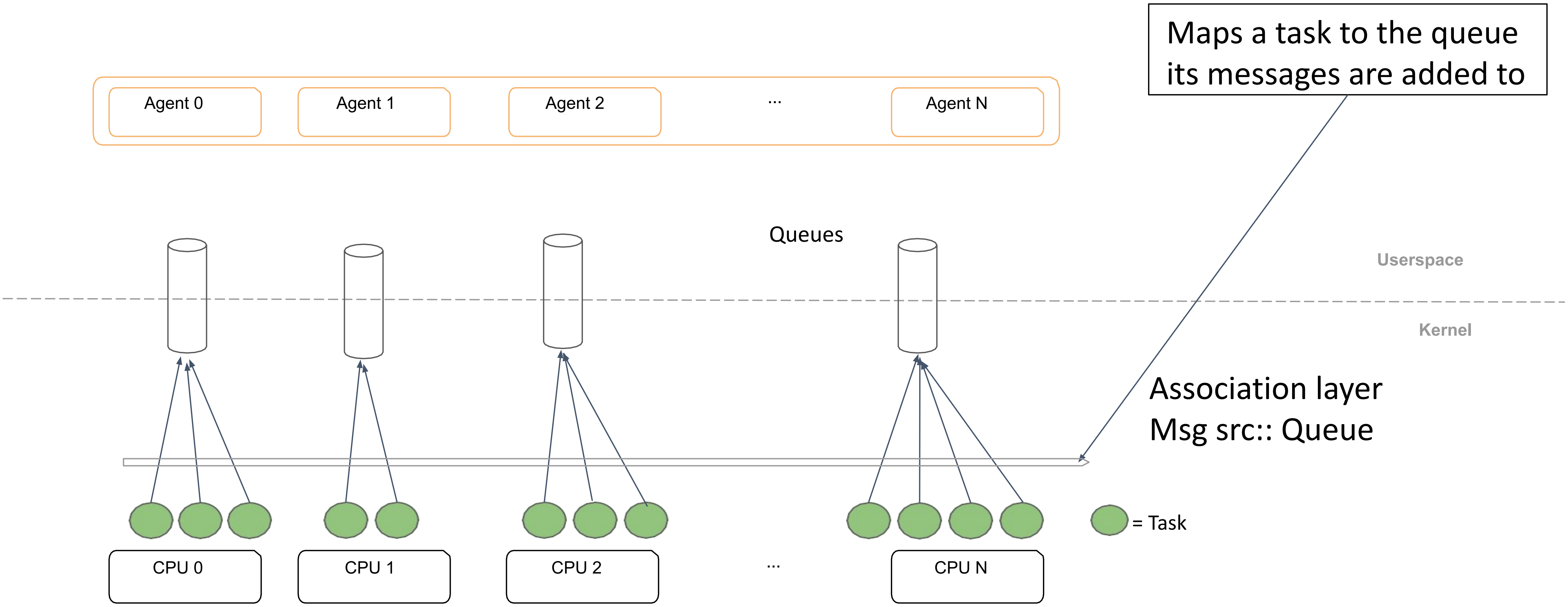


Task Messages: New, Blocked, Wakeup, Yield, Preempt, Departed, Dead

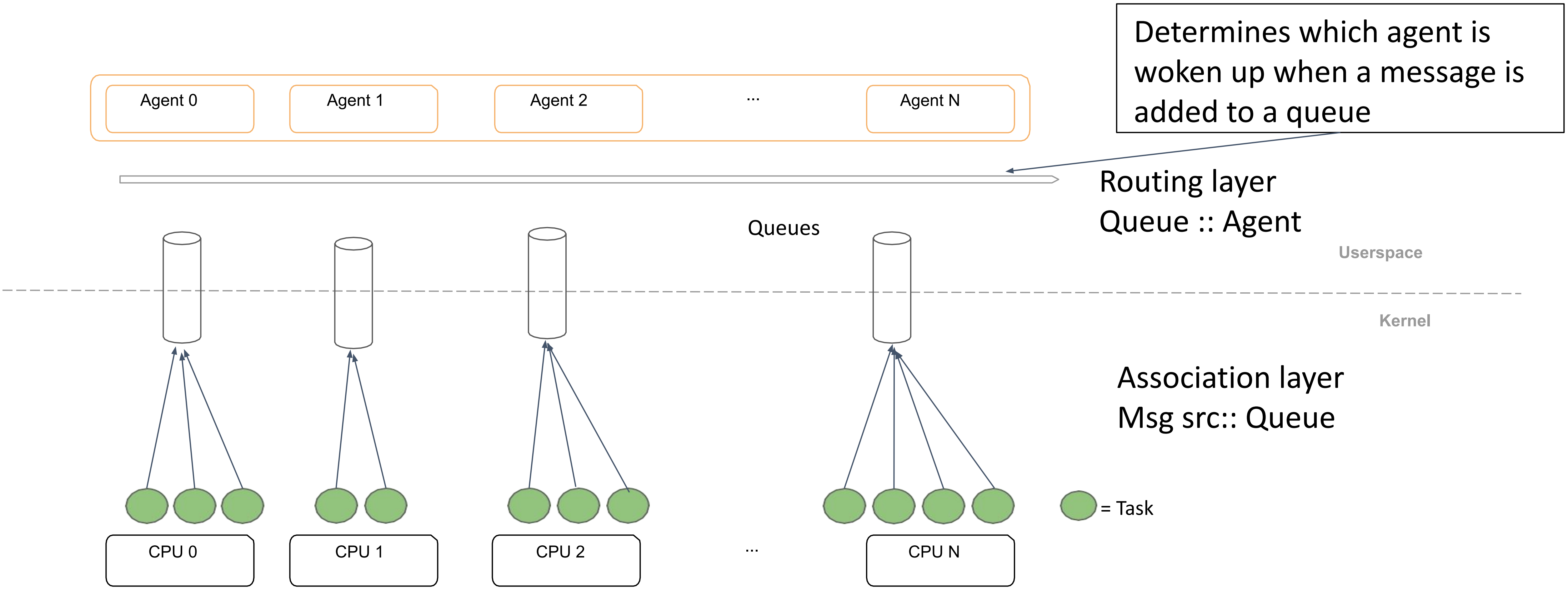
Per-CPU scheduling model



Per-CPU scheduling model

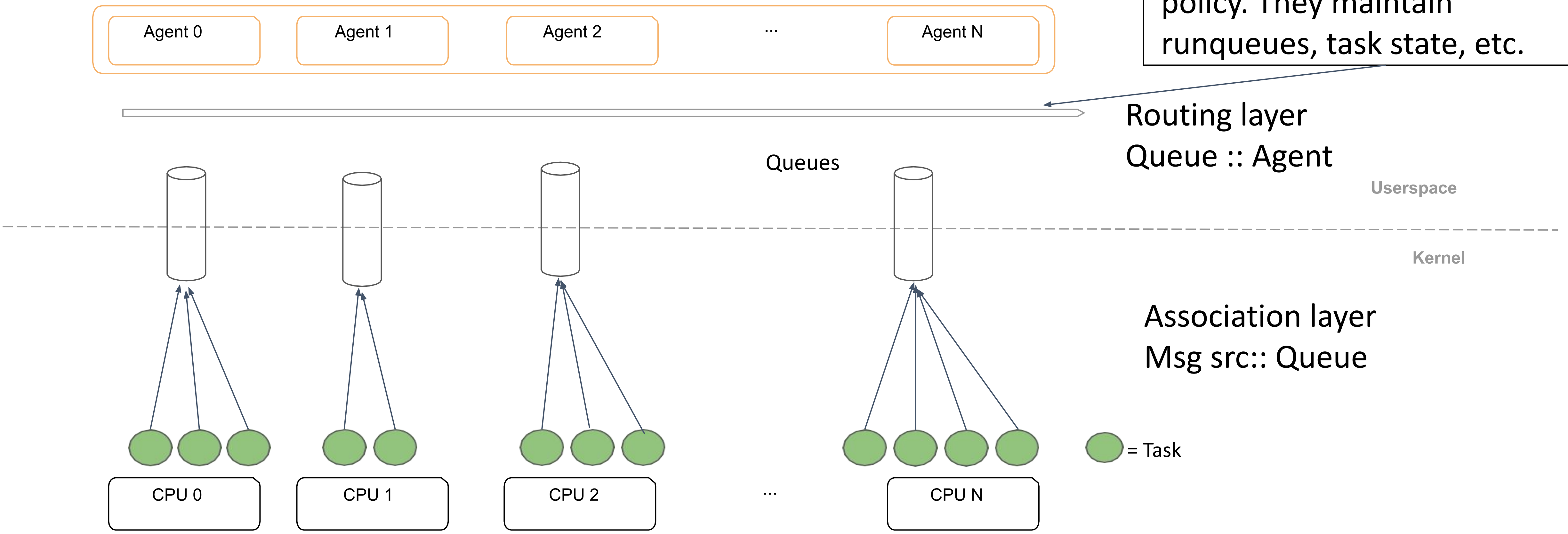


Per-CPU scheduling model

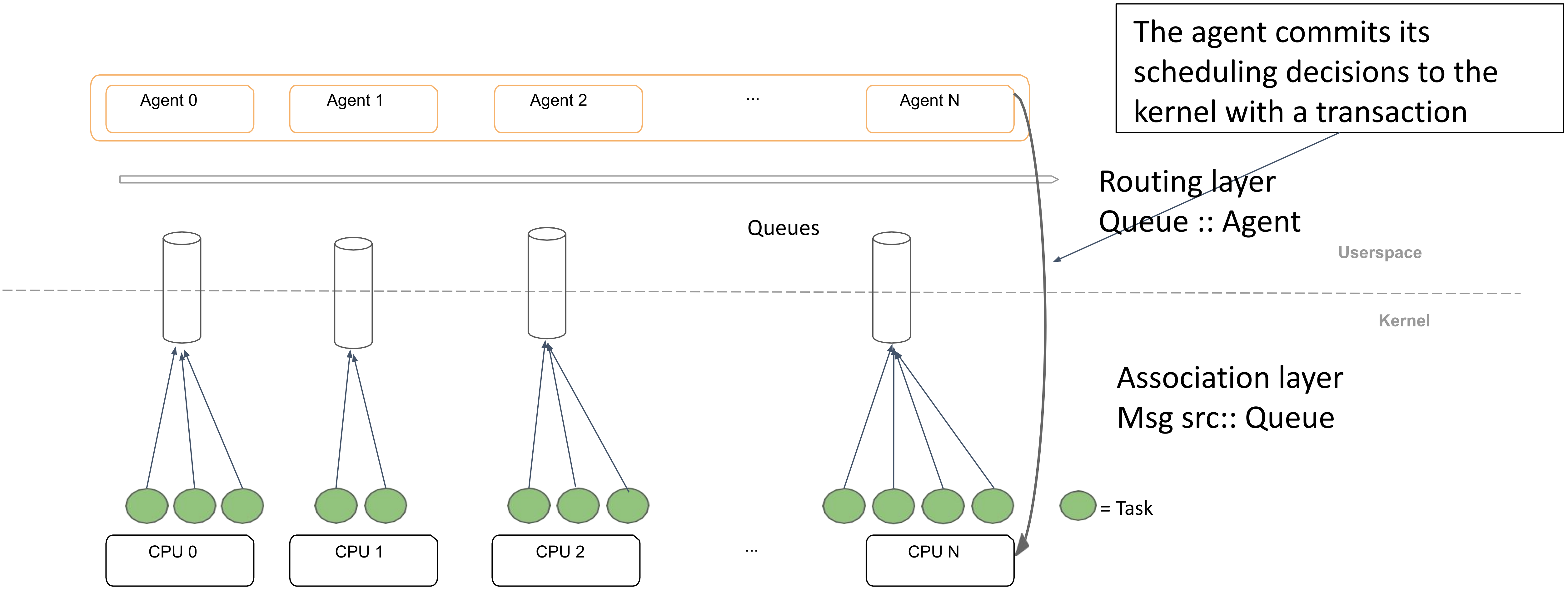


Per-CPU scheduling model

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

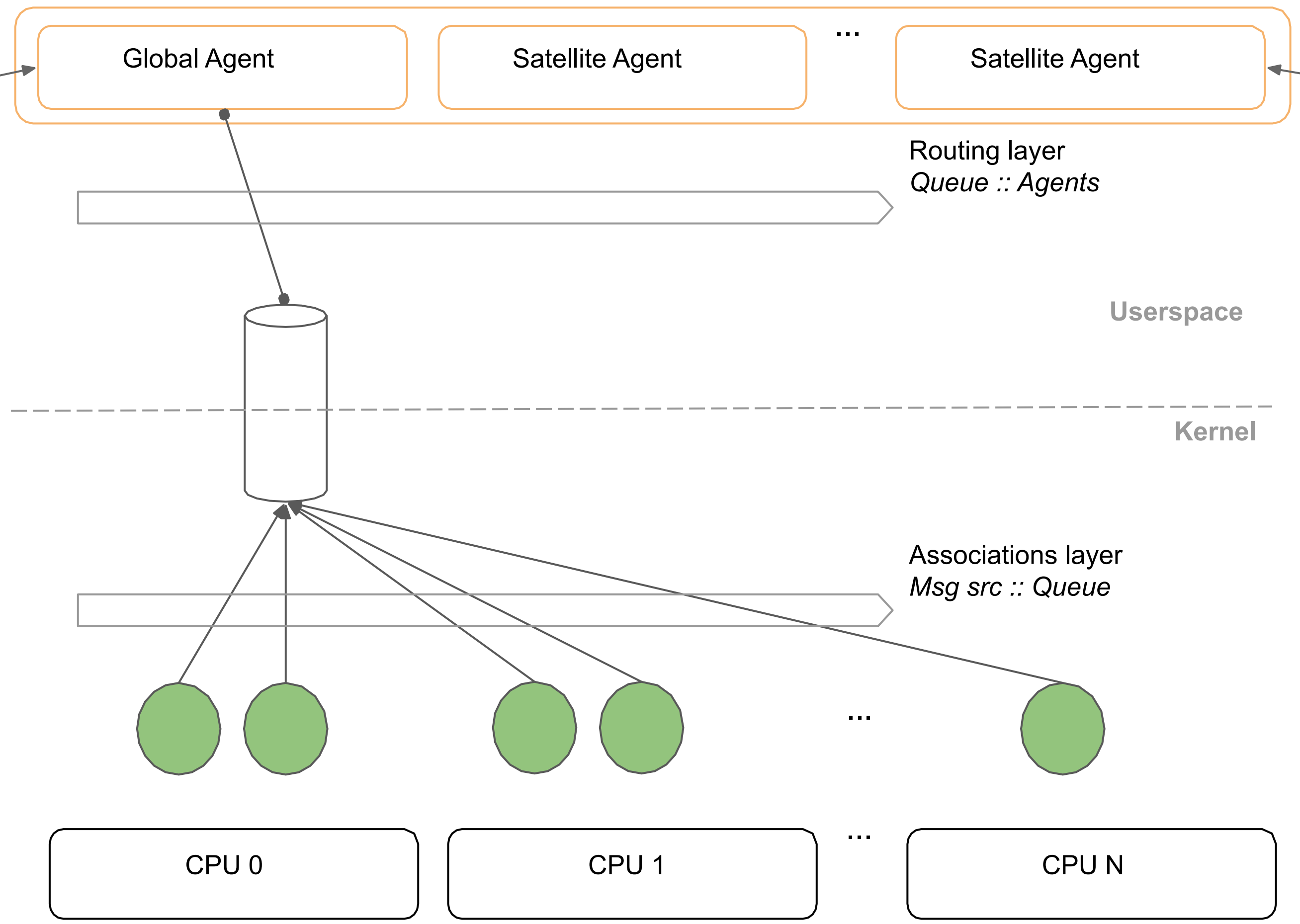


Per-CPU scheduling model



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

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

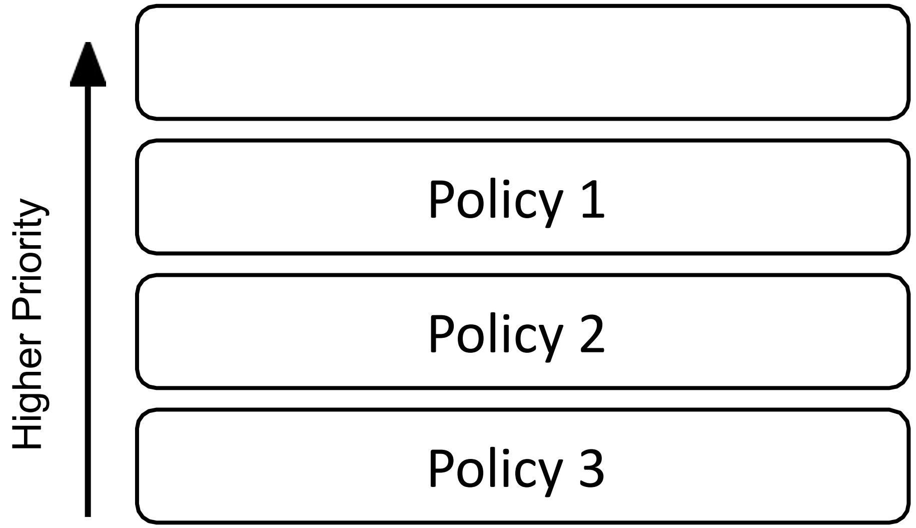
TXN_CREATE()	● Create a transaction
TXNS_COMMIT()	● Commit one or more transactions ● When >1 txns committed, use batch inter-processor interrupts (IPIs)
TXNS_RETRACT()	● Retract one or more transactions ● May fail

Synchronizing agents with the kernel

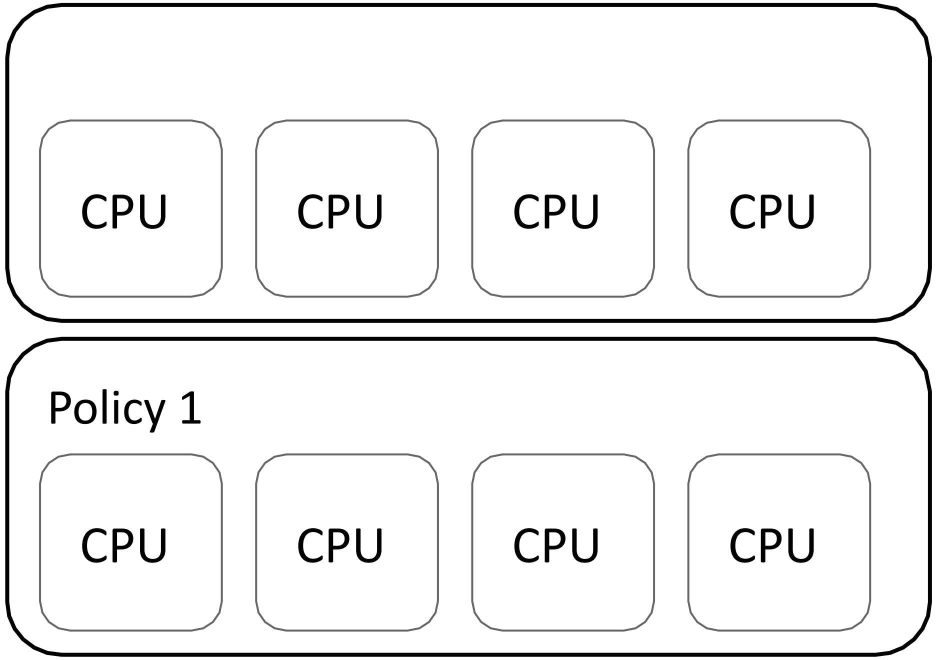
- The kernel is the **source of truth** and our **atomic store**
  - All task state live in the kernel
  - This state is ephemeral and lives between agent restarts
  - Many ghOSt scheduler interactions happen in a ctx where we do not synchronously invoke an agent
- How can we keep agents in sync with the kernel?
- Sequence numbers
  - Each task has a sequence number
  - When a task message is generated, increment the seq no and include in the message
  - When an agent opens a transaction, it includes the most recent seq no
  - If the seq no in the transaction is outdated, then that transaction fails

Co-locating policies

- Layering within an agent (same as Linux)
- Enclaves: Split CPUs into groups and let each agent schedule its own group



Layering



Enclaves

Quick upgrades and fault tolerance

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

ghOSt: A 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

Eval: Microbenchmark

- ghOSt API has similar overheads to other Linux schedulers
  - Practical for production, scheduling, even for microsecond scale requests

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

Eval: Microbenchmark

- ghOSt has fairly low overheads for remote scheduling, practical for workloads with microsecond-scale

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

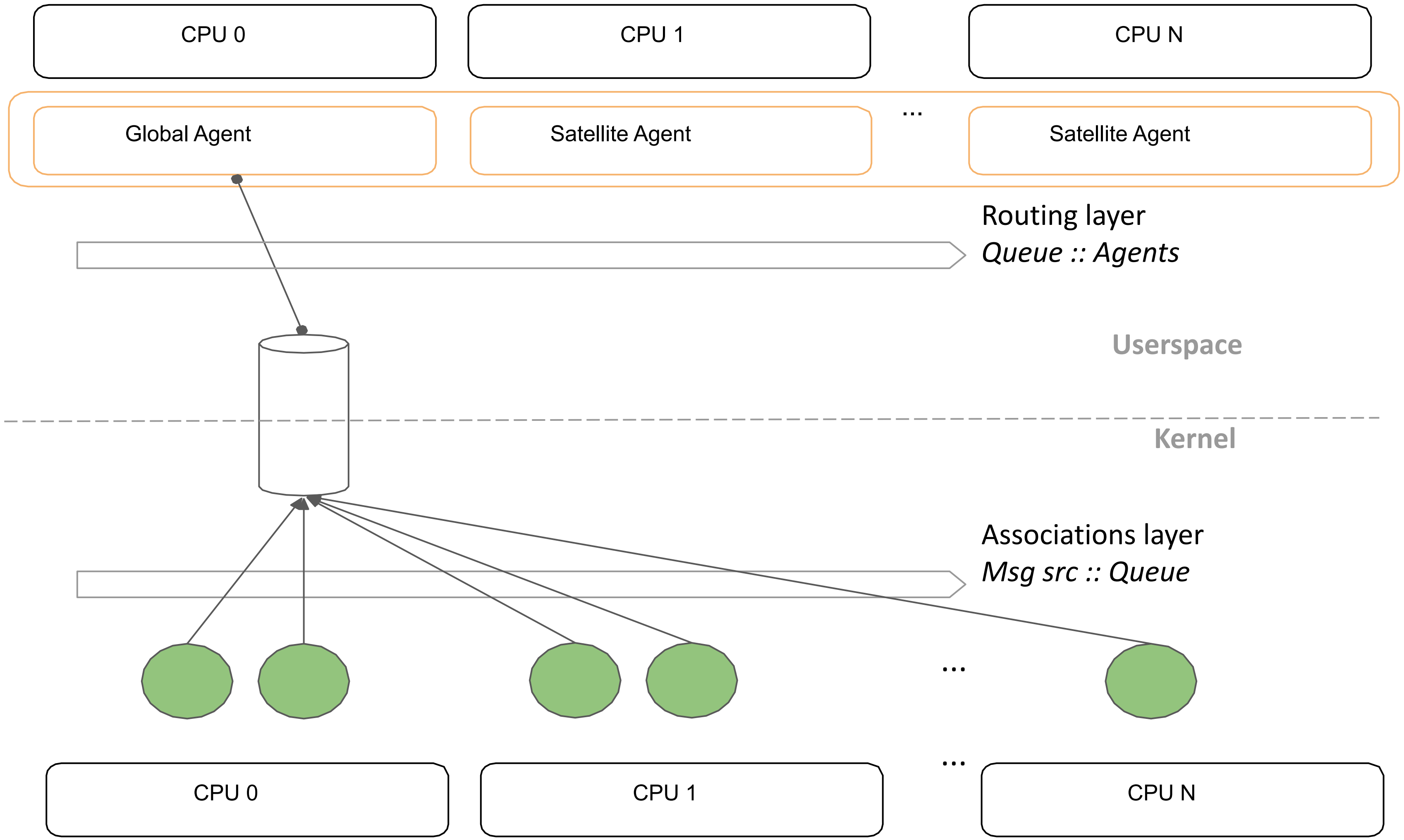
Remote Transactions Batch Commit (10 transactions)	
Committer Overhead	3964 ns (= 396 ns/transaction)
Target CPU Overhead	1821 ns
End-to-End Latency	5688 ns

Google Snap

- Snap is Google's internal low-latency packet processing framework
- One main polling thread that processes network traffic
- Additional worker threads are spawned as needed when traffic increases
- Snap uses a microsecond-scale real-time Linux kernel scheduler (MicroQuanta)
- Compared with centralized FIFO policy

- Eval setup:
  - One server and six clients
  - Five clients send 64 kB messages, one client sends 64 B messages

Centralized scheduling model

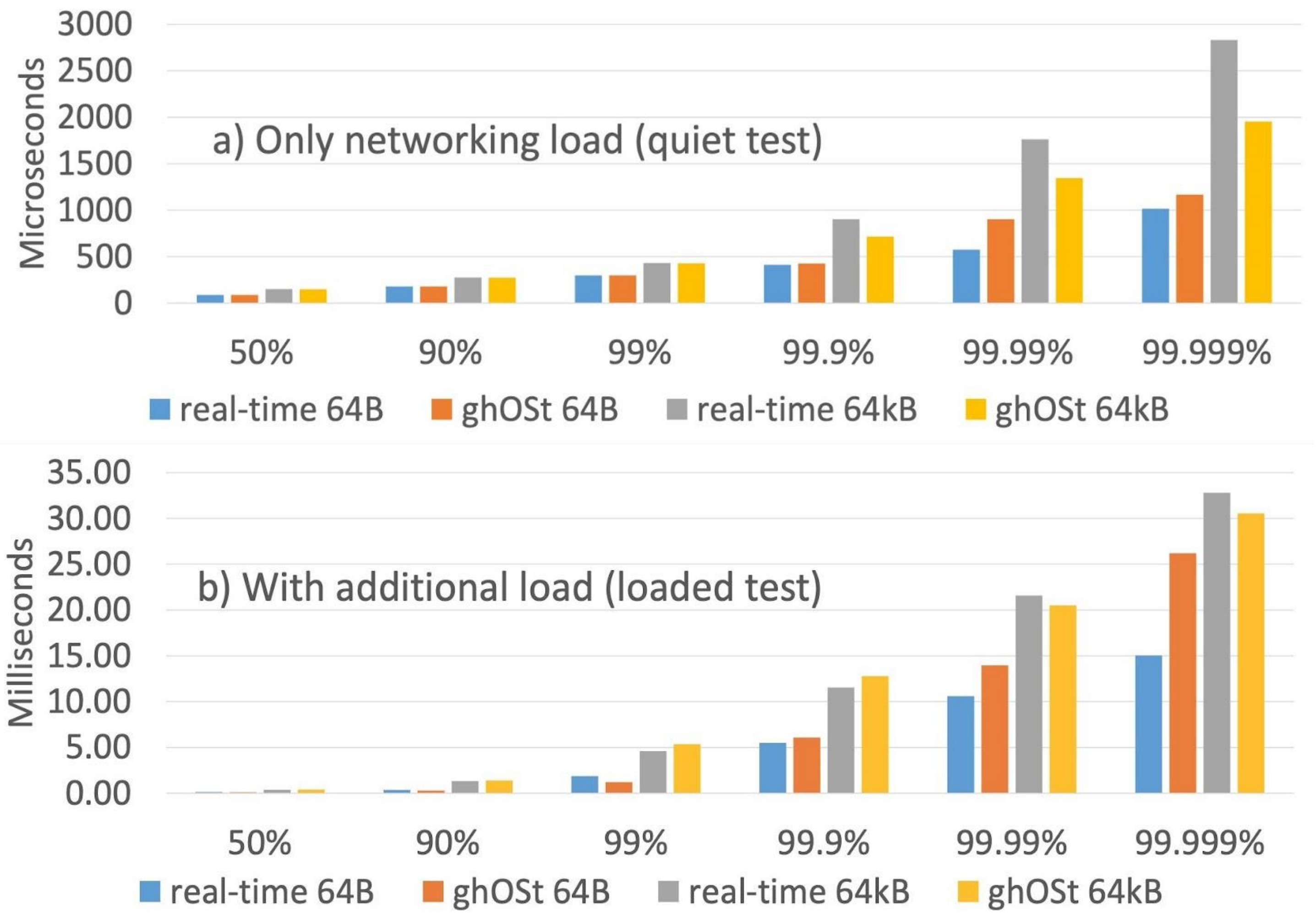


Centralized model accelerates network workloads

- No centralized scheduling model exists in Linux

- Centralized model is much more responsive to network load changes
- Faster rebalancing across cores (us-scale rather than ms-scale)
- Highly effective for us-scale workloads

Google Snap results



Future work:

- Memory management
- New policies
- Tighter integration with other system stacks (e.g., networking stack)
- Formal verification
    - Policy is isolated from complicated mechanisms

ghOSt Summary

- Runs scheduling policies in a user space 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

Another alternative: sched_ext

sched_ext: Scheduling policies as eBPF programs

- Write a scheduling policy in BPF; compile it; and load
- New sched_class, at a lower priority than CFS
- Safe cannot crash the kernel
  - Verifier ensures the kernel integrity
  - Watchdog boosts sched_ext scheduler if a runnable task isn't scheduled within some timeout
- BPF programs implement a set of callbacks:
  - Wakeup, enqueue/dequeue, state change, rebalancing, cgroup integration
  - Timeouts, # of tasks that can be dispatched