CertiKOS

An Extensible Architecture for Building Certified Concurrent OS Kernels

Ronghui Gu, Zhong Shao et al, Yale University, FLINT Research Group

“ Complete Formal Verification is the only way to guarantee that a system is free of programming errors ”

- seL4 (SOSP ‘09)

Shared Memory Consistency ???

- “Proofs about concurrent programs are much harder than proof about sequential programs” (seL4, SOSP ‘09)
  - I/O concurrency
  - Multithreading
  - Multiprocessors
- Can’t use a big lock, because performance.
  - Must use fine grained locks.
- “Verification to a kernel version with fine grained locking will far exceed the cost already paid for verifying the single core version” (Peters, ApSys ‘15)

What to prove to verify an OS kernel?

- Functional Correctness
  - Specification S, Kernel code K, User Program P
  - For all user program P, $K \bowtie P$ refines $S \bowtie P$
- Liveness
  - All the system calls will eventually return
  - Pretty hard, at the scale of the concurrent kernels

ASIDE

new technical contributions

certified concurrent layers

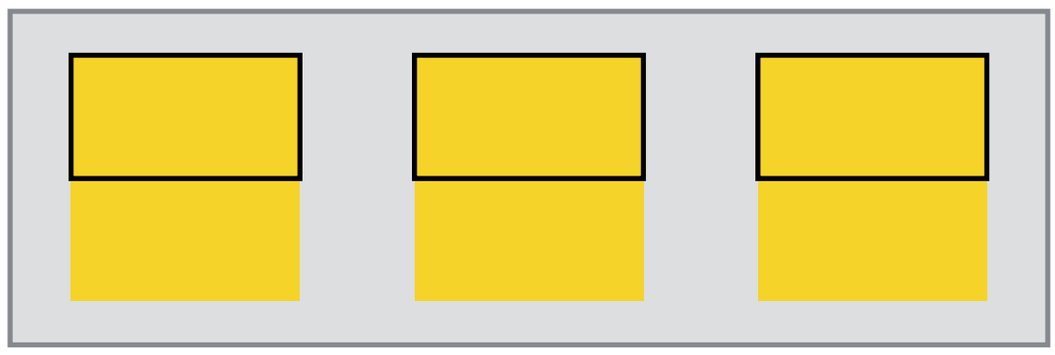
logical log + hardware scheduler
+ environment context

push/pull model

multicore machine lifting

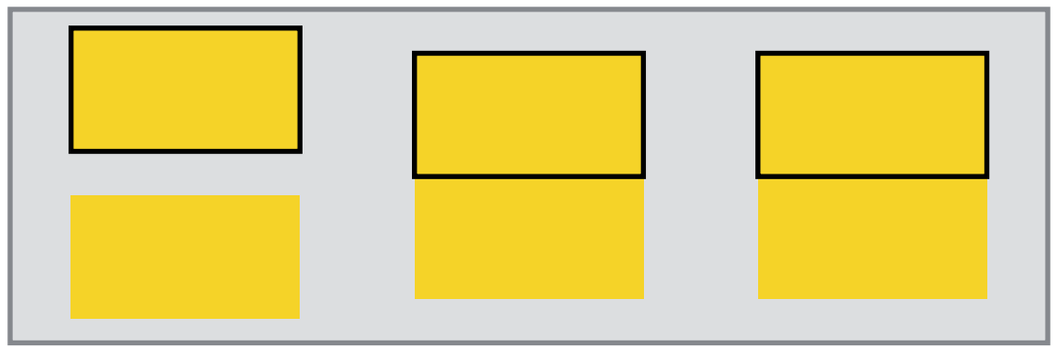
certified sequential layers

certified objects

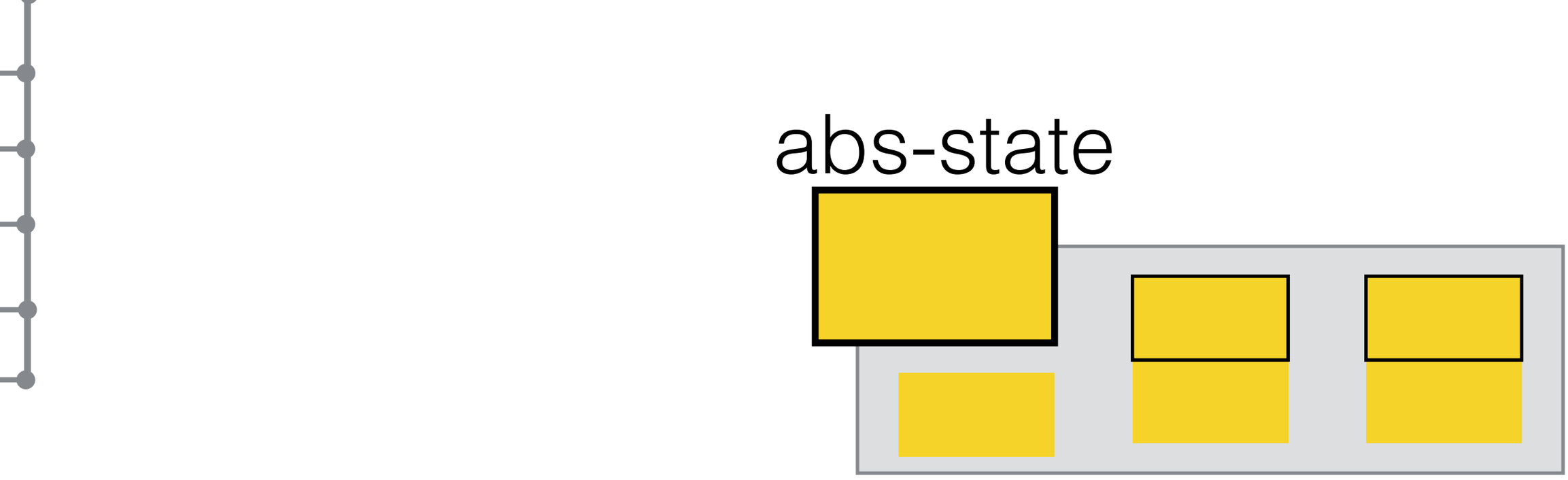


specification of modules to trust

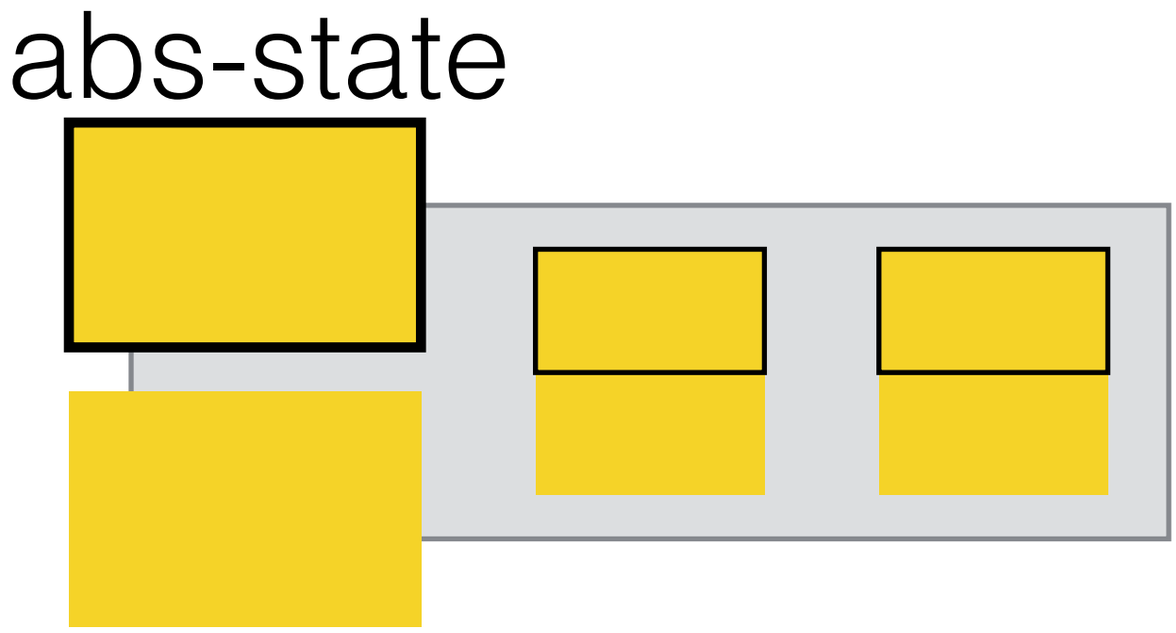
certified sequential layers

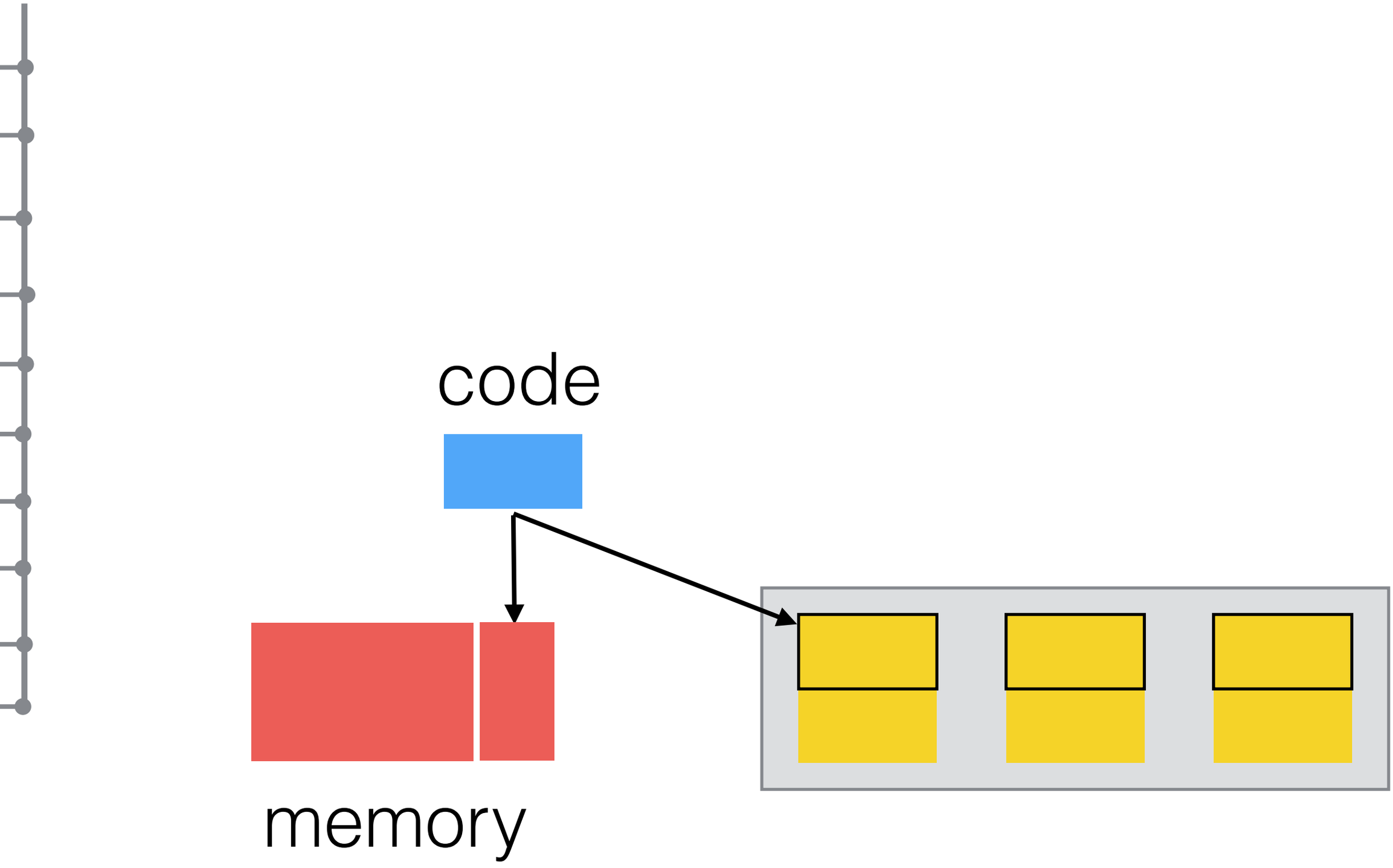


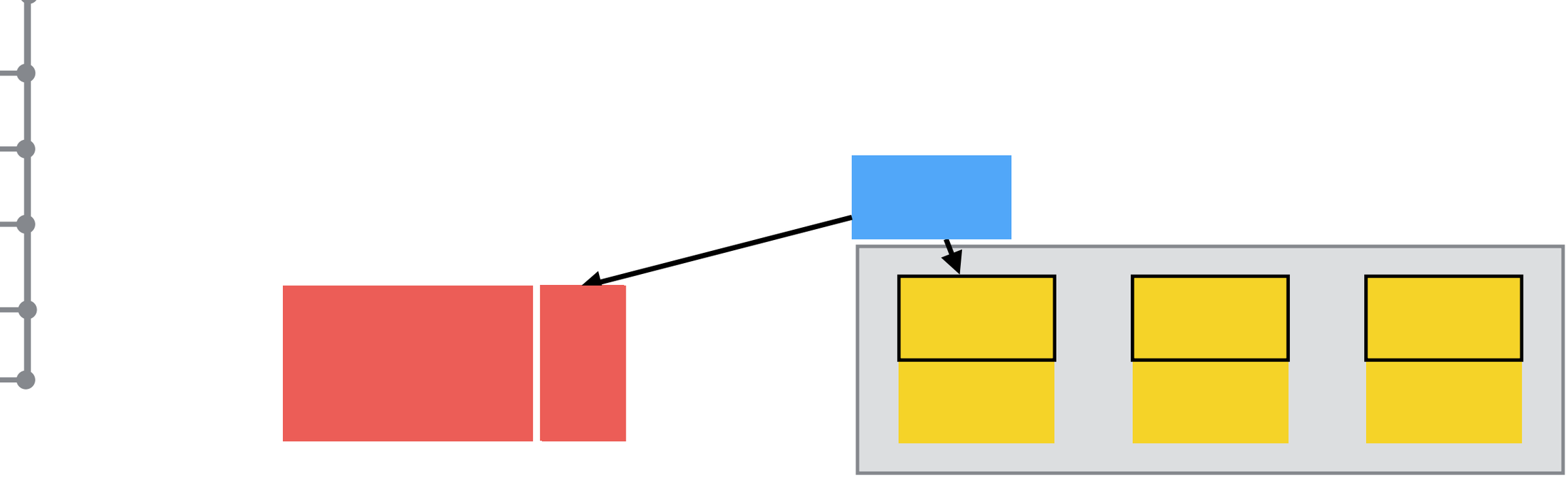
certified sequential layers

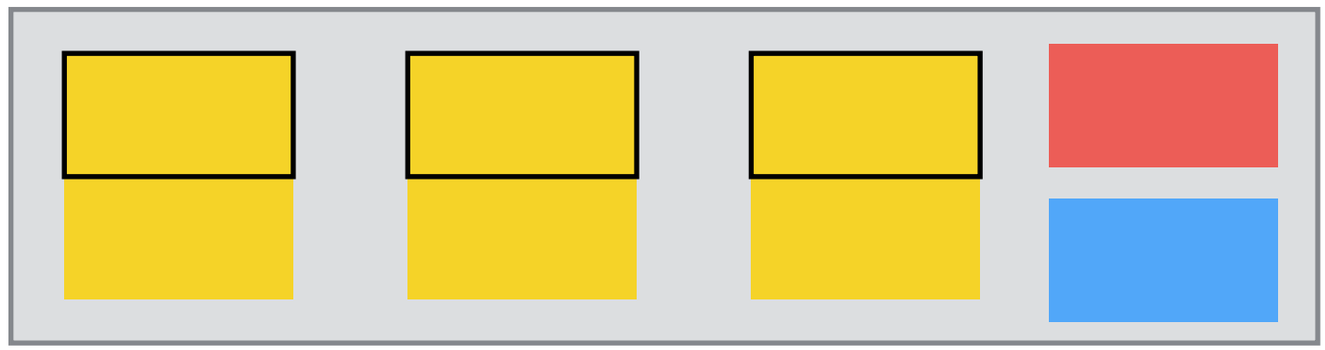


certified sequential layers

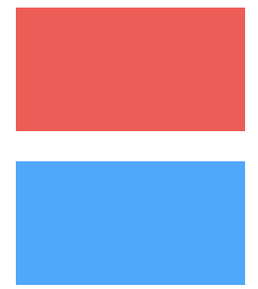


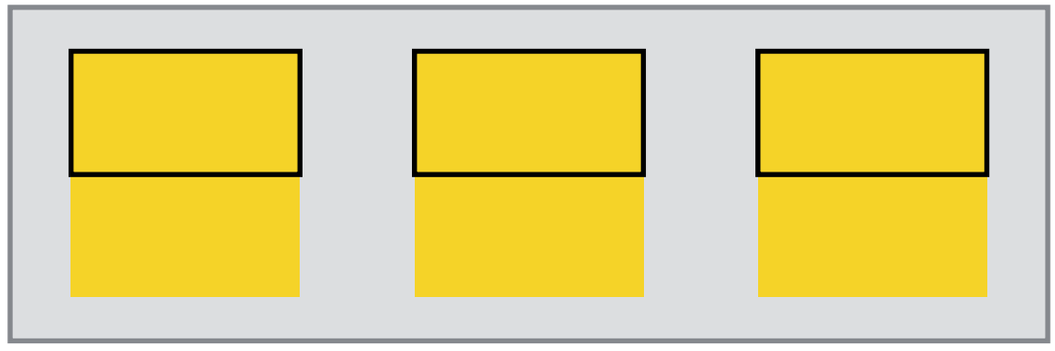

code
memory



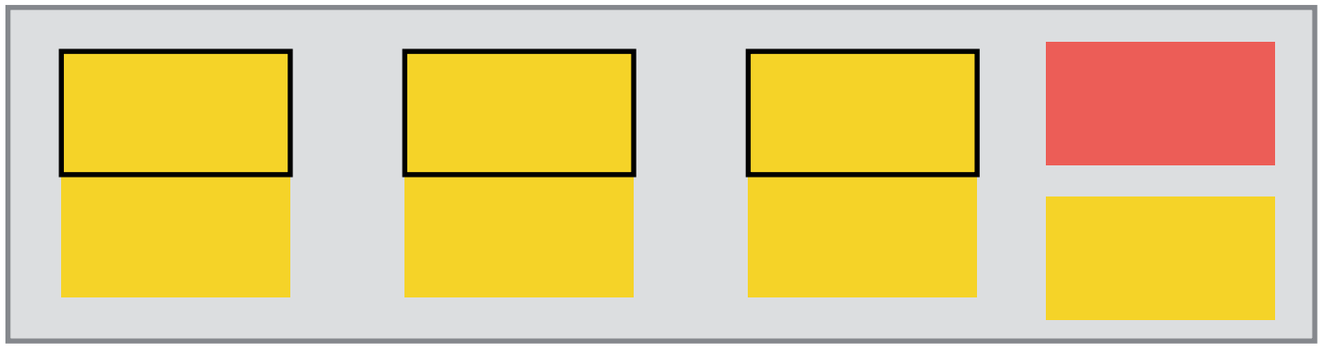


implementation

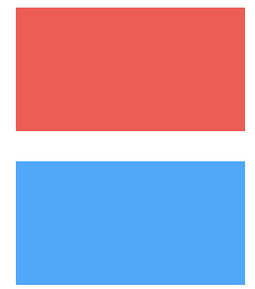


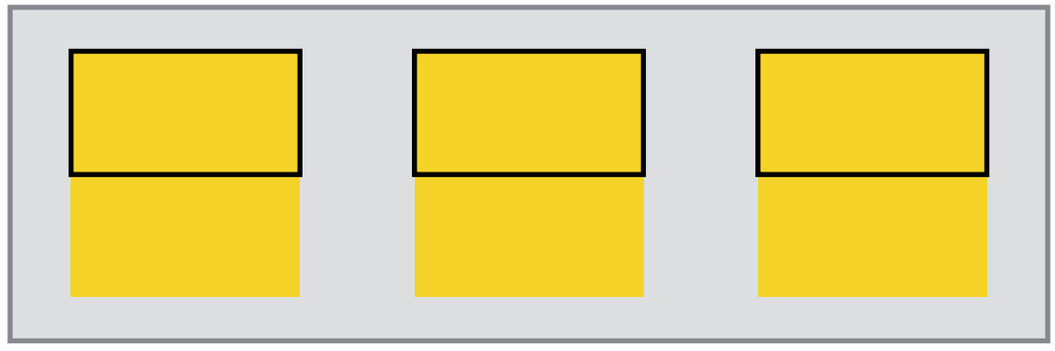


specification

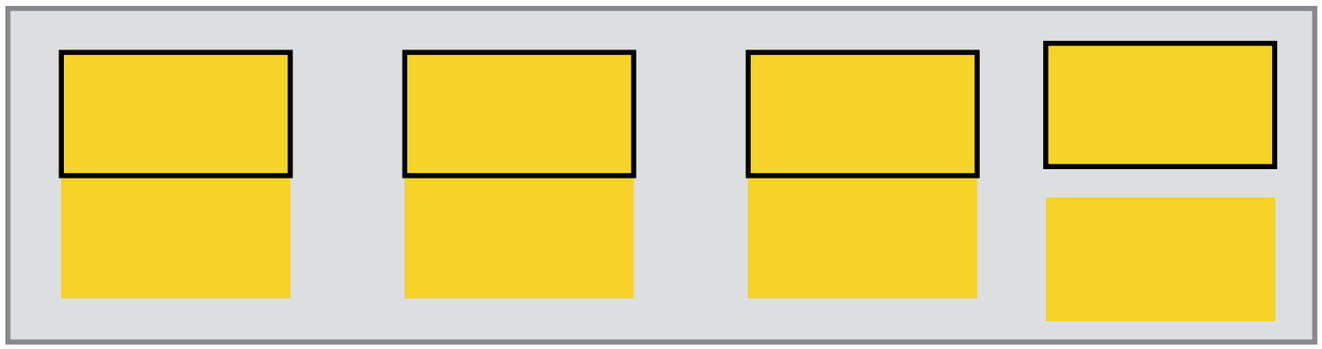


implementation

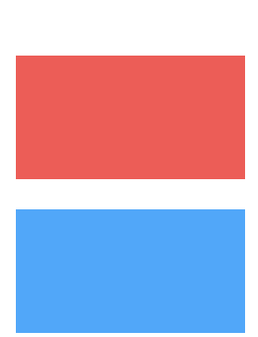


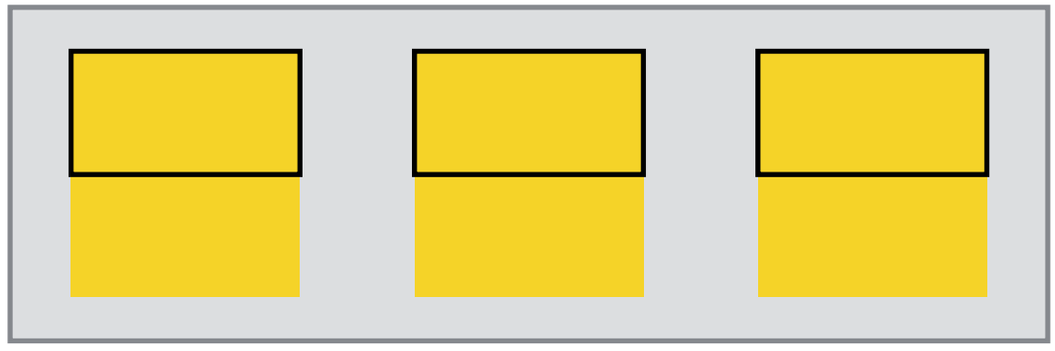


specification



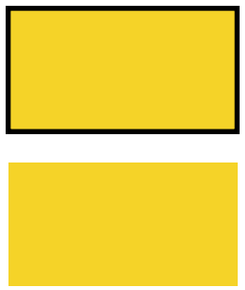
implementation

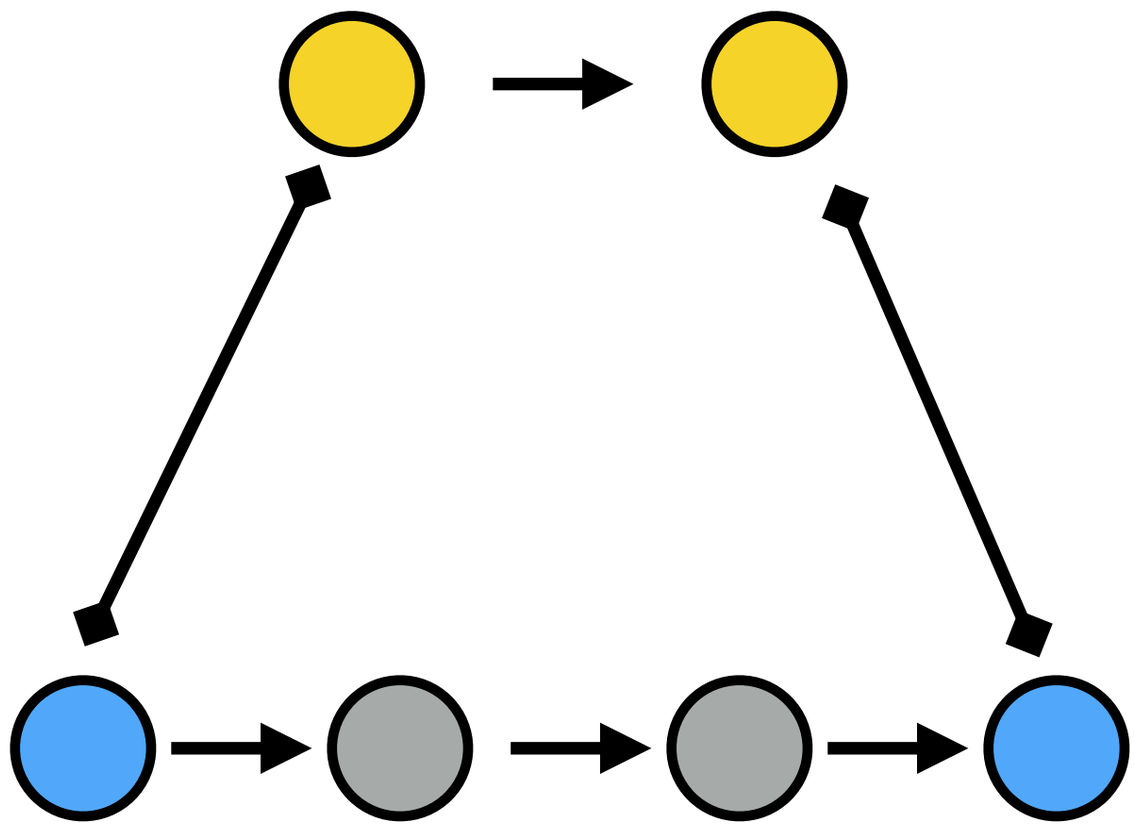


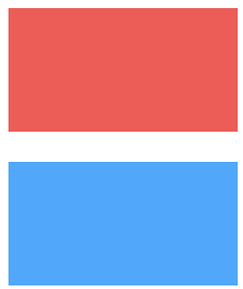


simulation proof

specification







implementation

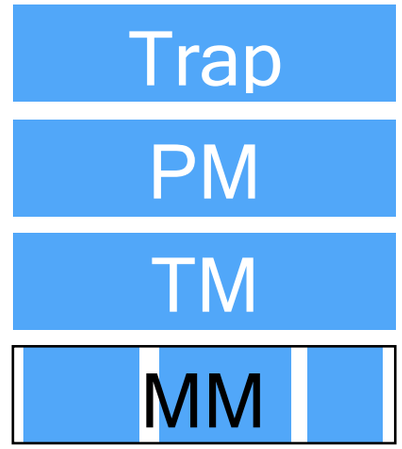
verify a sequential kernel

[POPL'15]

kernel

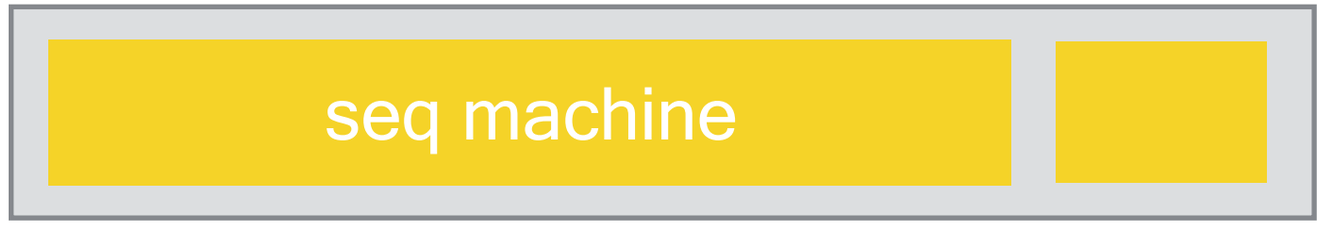
code

seq machine



seq machine

contributions



extensibility is the key to support

concurrency

support concurrency

contributions

trap

virt

proc

thread

mem

seq machine

multicore machine

contributions

- reuse

trap

virt

proc

thread

mem

CPU-local machine

multicore machine

contributions

reuse

trap

virt

proc

thread

mem

spin-lock

CPU-local machine

multicore machine

contributions

- reuse
- mix of 3

trap

virt

proc

thread-local machine

thread

mem

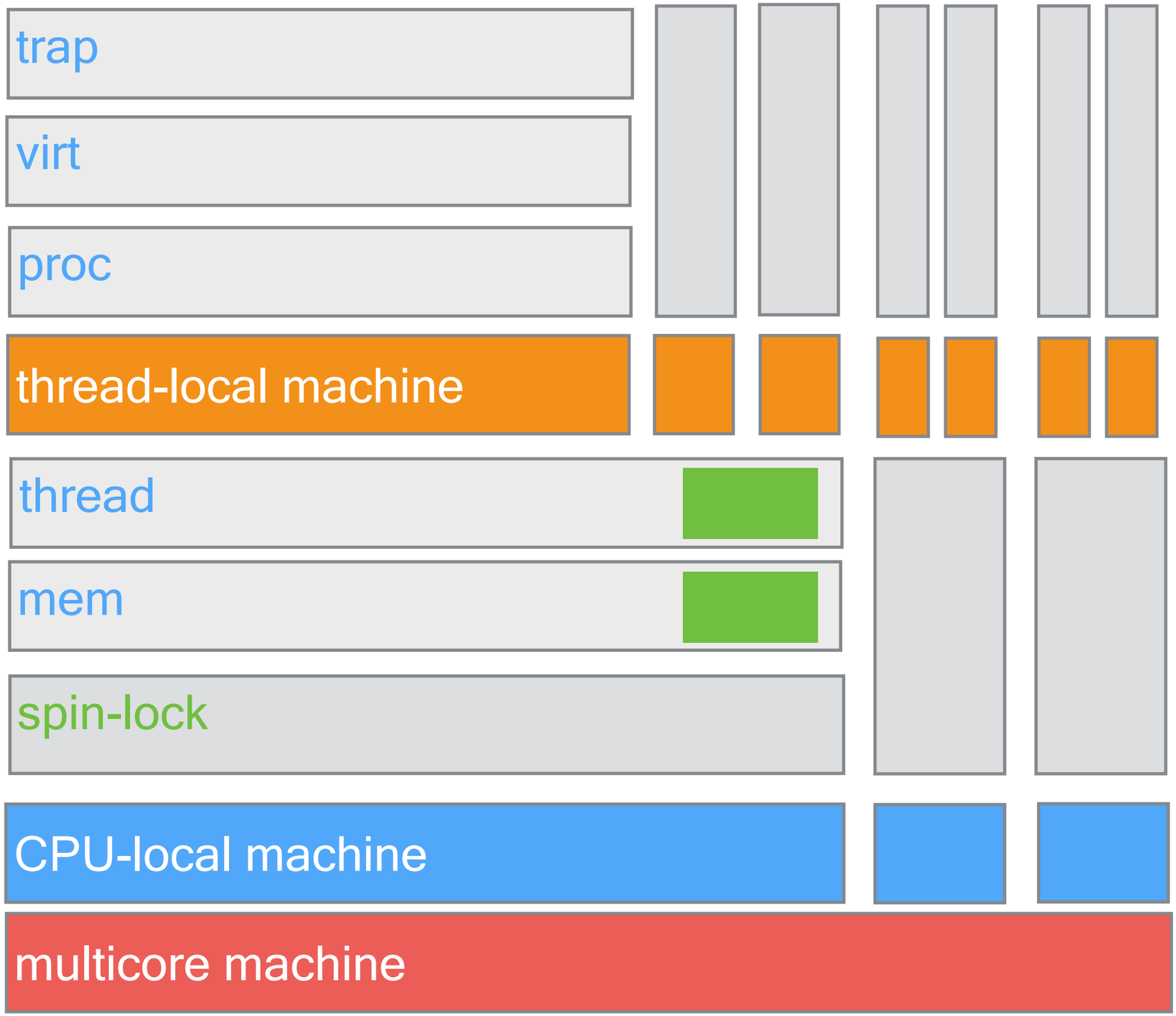
spin-lock

CPU-local machine

multicore machine

contributions

- mC2
- reuse
- mix of 3



certified concurrent layers

trap

virt

proc

thread

thread

mem

spin-lo

CPU-lo

multico

certified concurrent layers

local objects

certified concurrent layers

trap

virt

proc

thread

thread

mem

spin-lo

CPU-lo

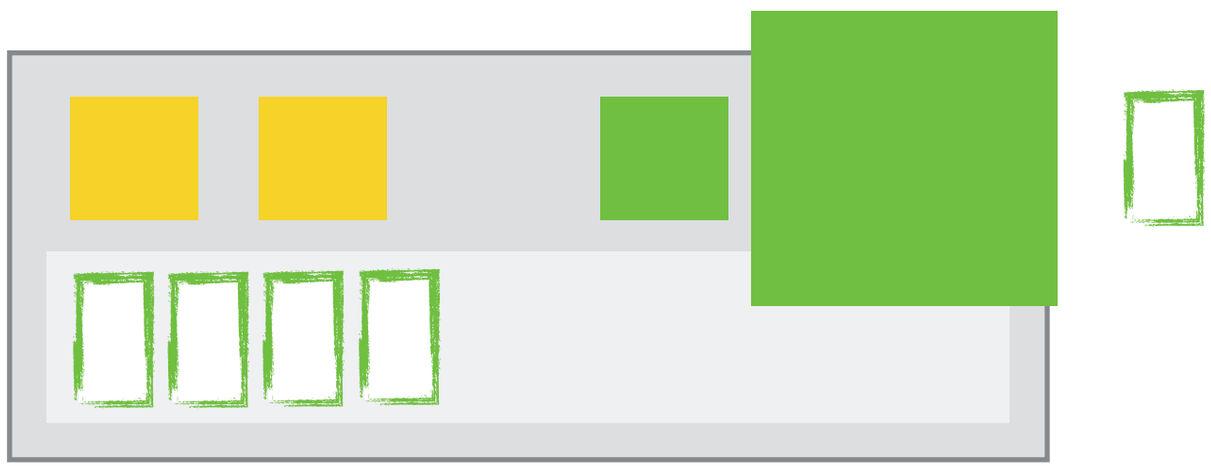
multico

atomic objects

logical log

a sequence of events

certified concurrent layers



trap

virt

proc

thread

thread

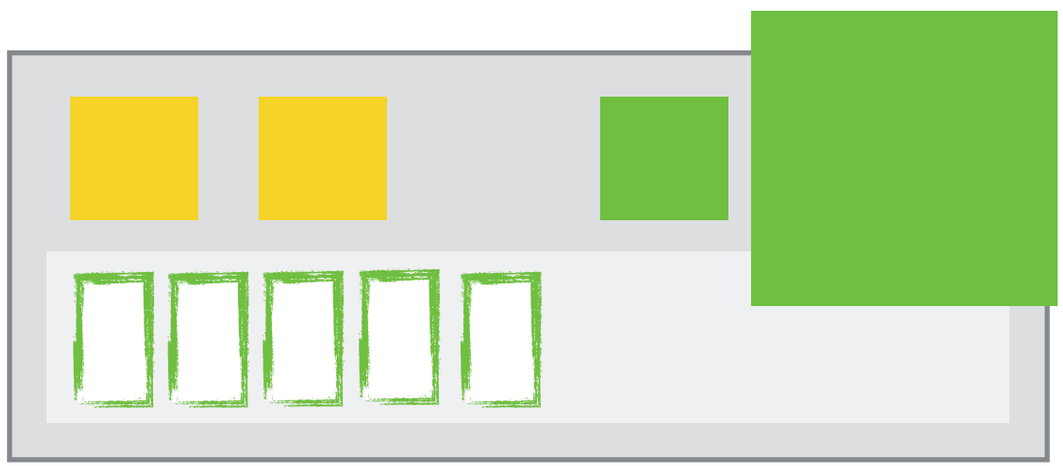
mem

spin-lo

CPU-lo

multico

certified concurrent layers



trap

virt

proc

thread

thread

mem

spin-lo

CPU-lo

multico

certified concurrent layers

trap

virt

proc

thread

thread

mem

spin-lo

CPU-lo

multico

share

trap

virt

proc

thread

thread

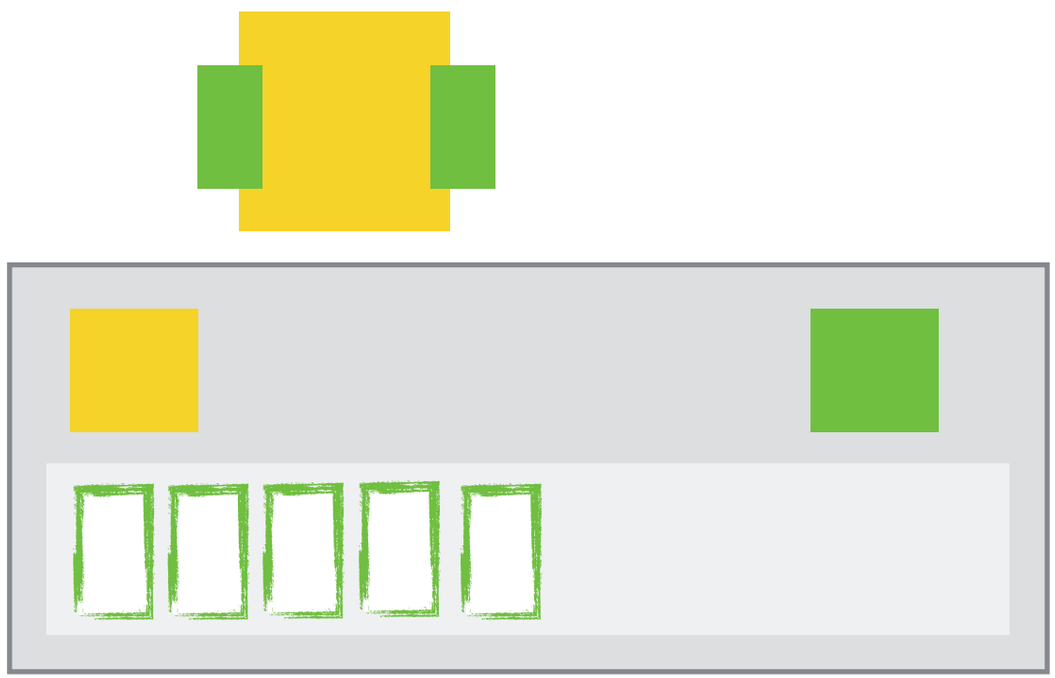
mem

spin-lo

CPU-lo

multico

fine-grained lock



trap

virt

proc

thread

thread

mem

spin-lo

CPU-lo

multico

fine-grained lock

trap

virt

proc

thread

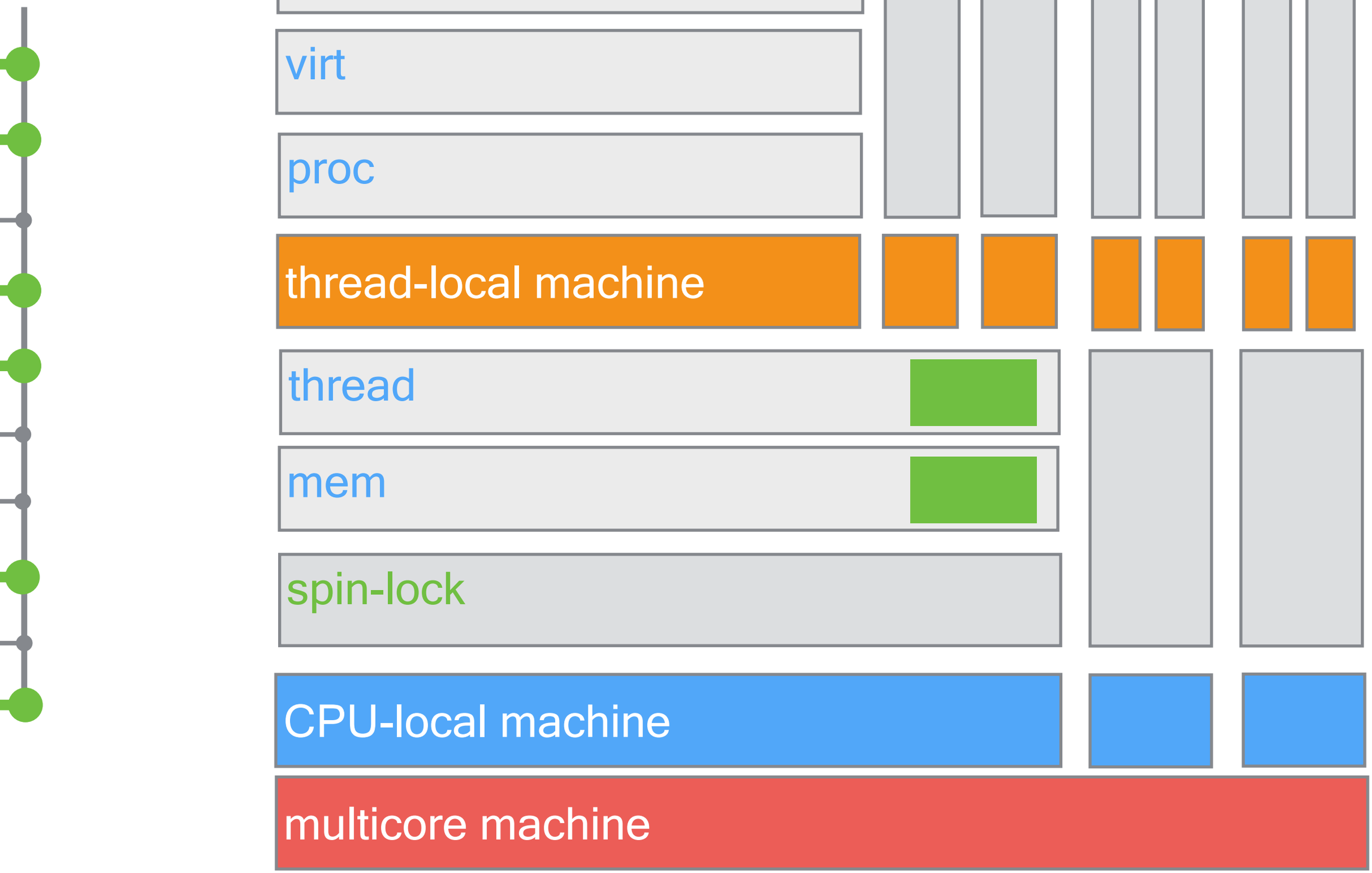
thread

mem

spin-lo

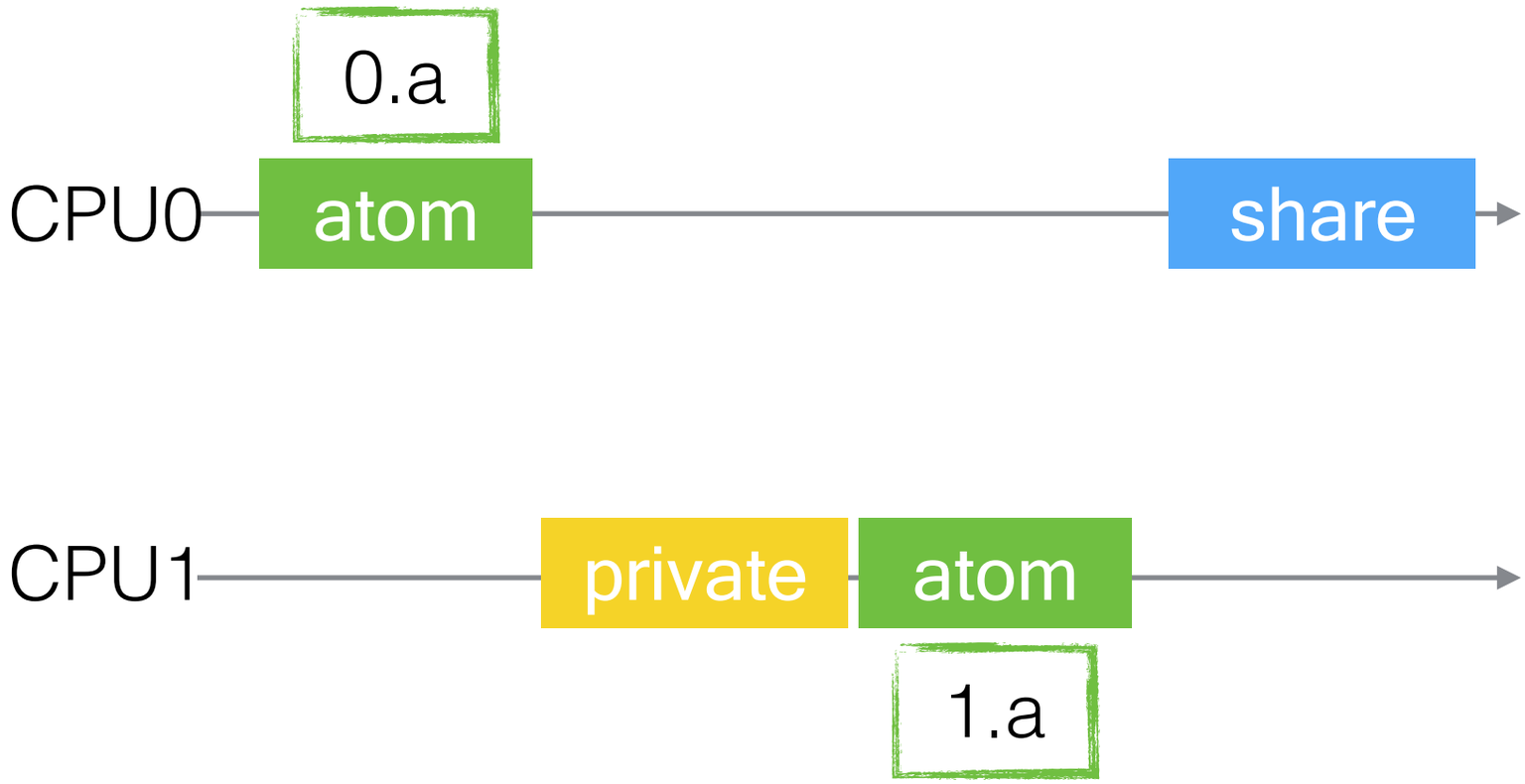
CPU-lo

multico


trap
virt
proc
thread-local machine
thread
mem
spin-lock
CPU-local machine
multicore machine

step 0: raw x86 multicore model

assume sequential consistency



trap

virt

proc

thread

thread

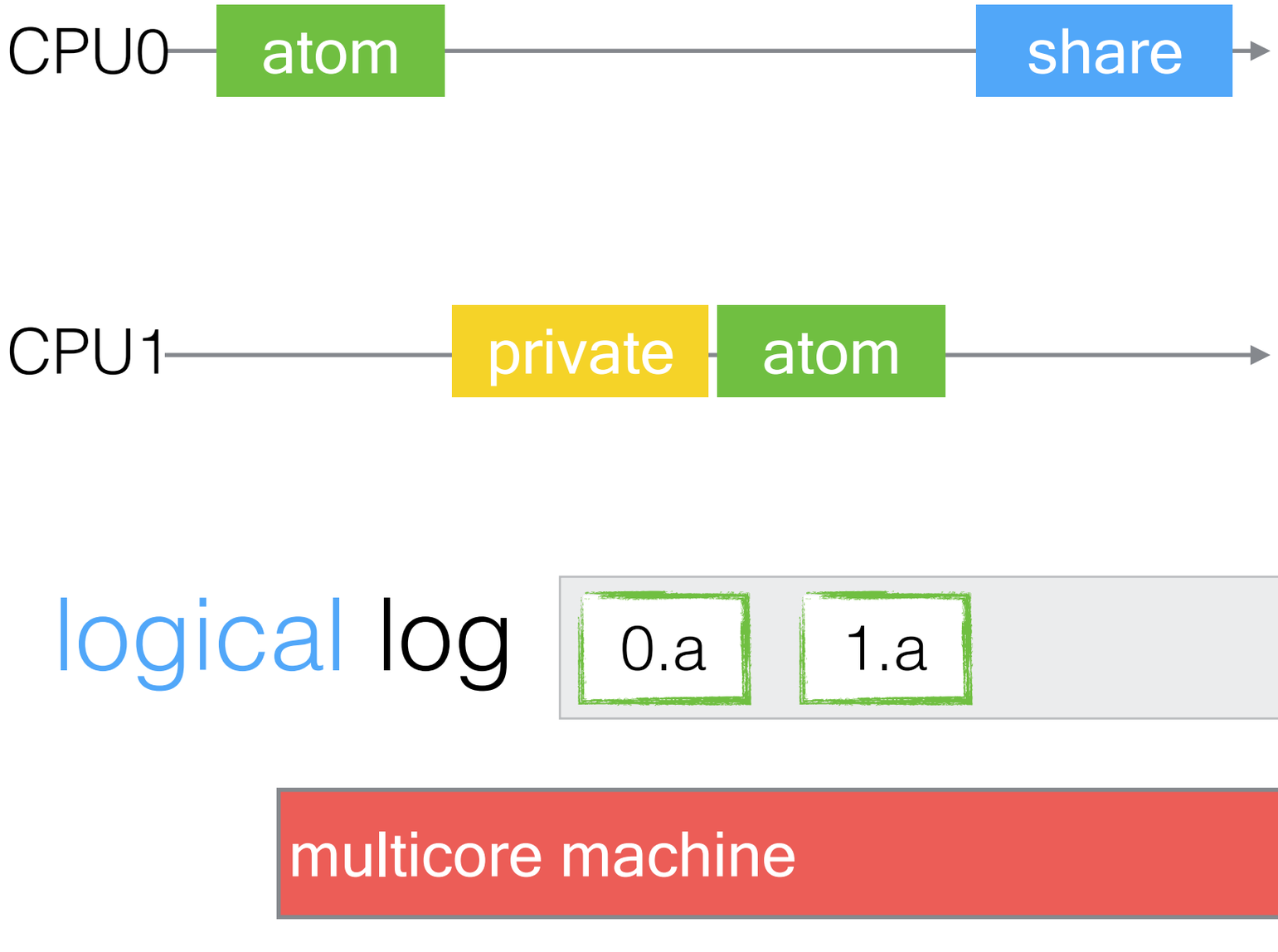
mem

spin-lo

CPU-lo

multicore machine

step 0: raw x86 multicore model



logical log

multicore machine

trap

virt

proc

thread

thread

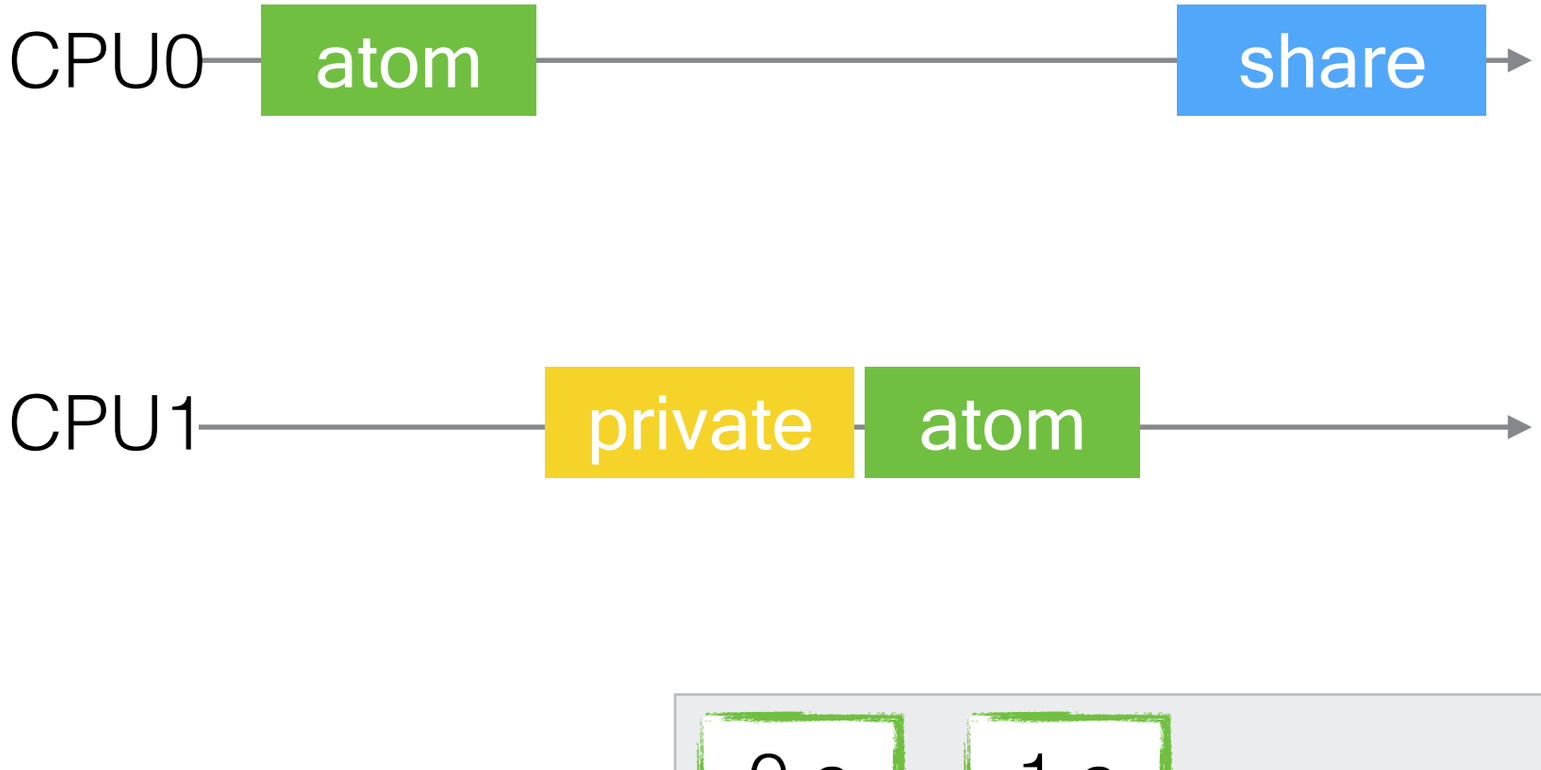
mem

spin-lo

CPU-lo

step 0: raw x86 multicore model

non-determinism



multicore machine

trap

virt

proc

thread

thread

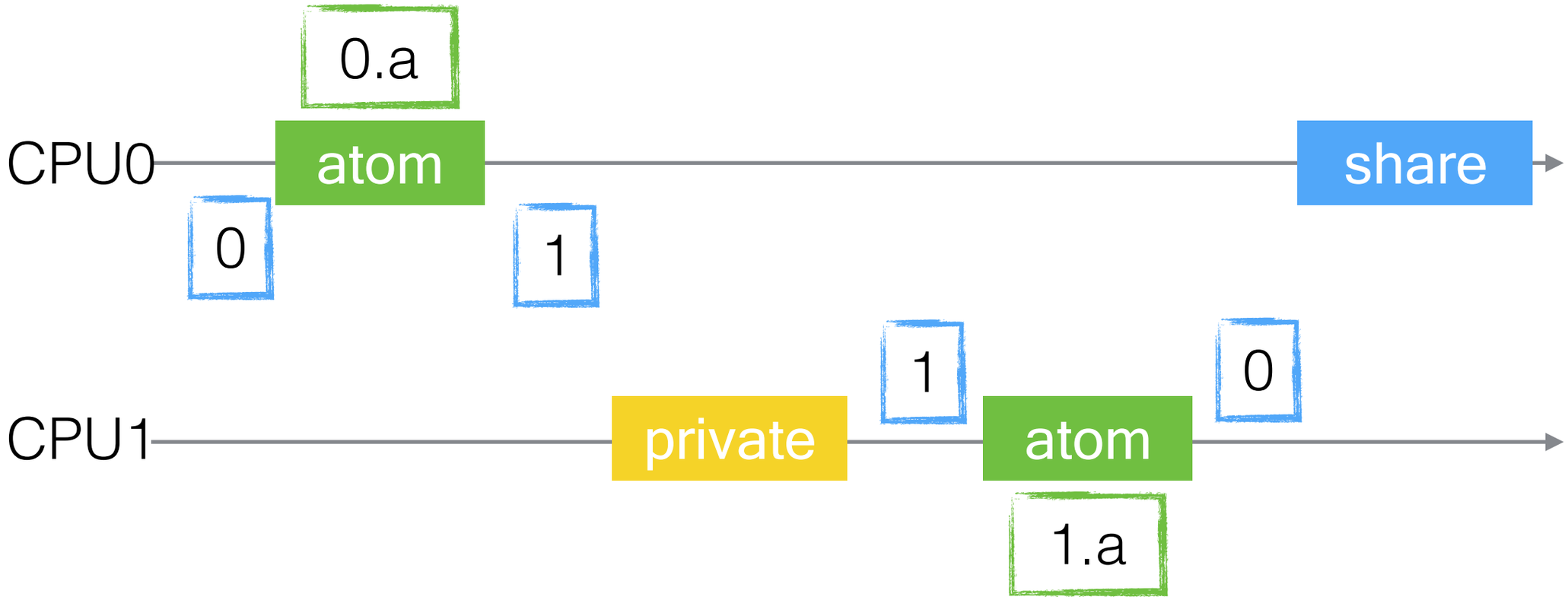
mem

spin-lo

CPU-lo

step 0: raw x86 multicore model

non-determinism



trap

virt

proc

thread

thread

mem

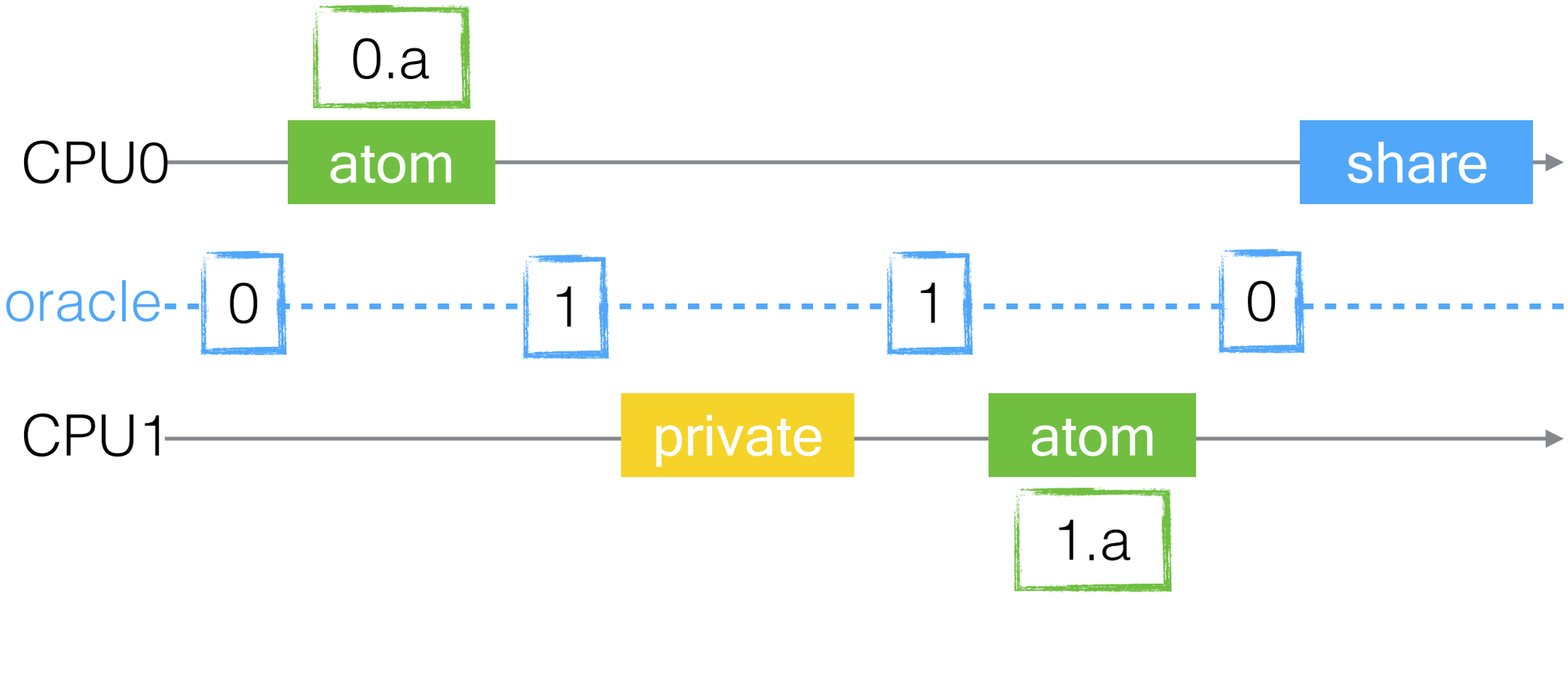
spin-lo

CPU-lo

multicore machine

step 0: raw x86 multicore model

non-determinism



trap

virt

proc

thread

thread

mem

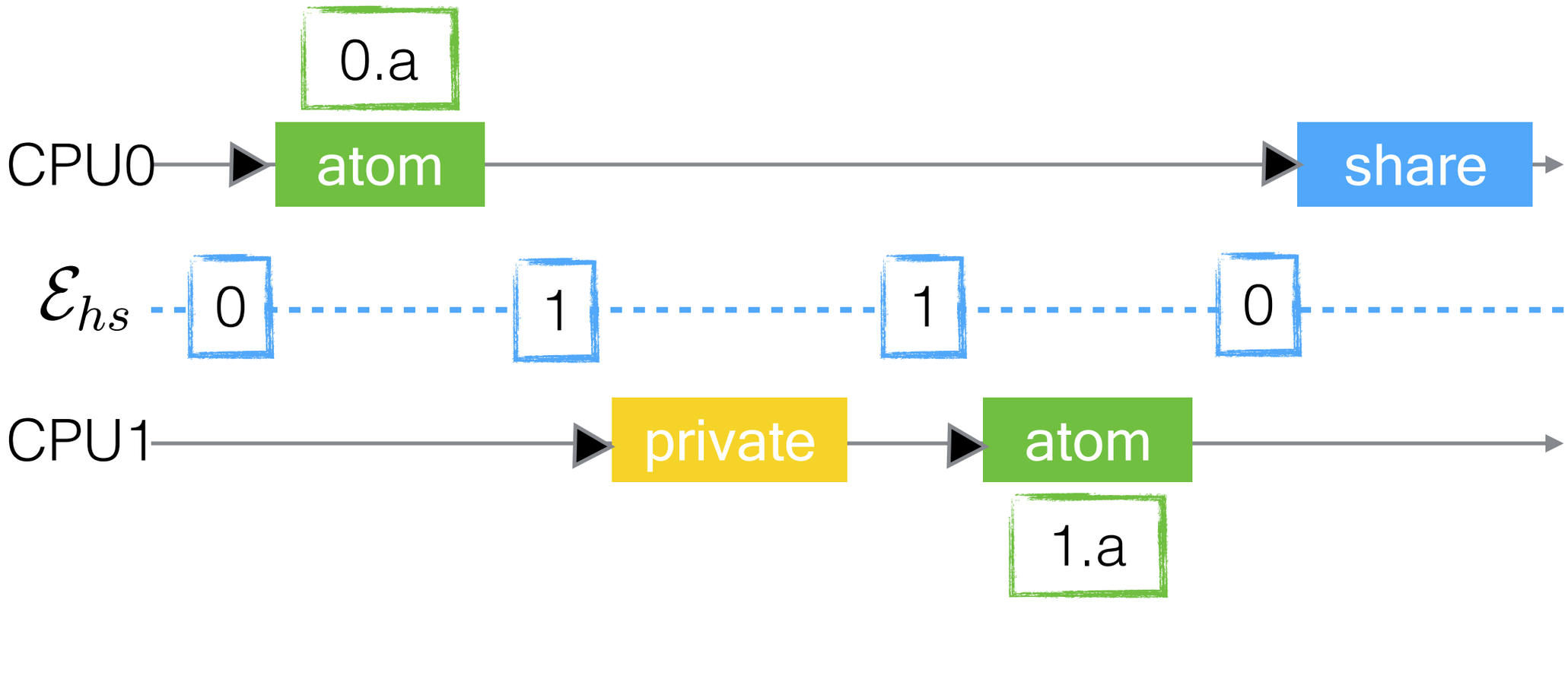
spin-lo

CPU-lo

multicore machine

step 1: hardware scheduler $\mathcal{E}_{hs}$

purely logical



trap

virt

proc

thread

thread

mem

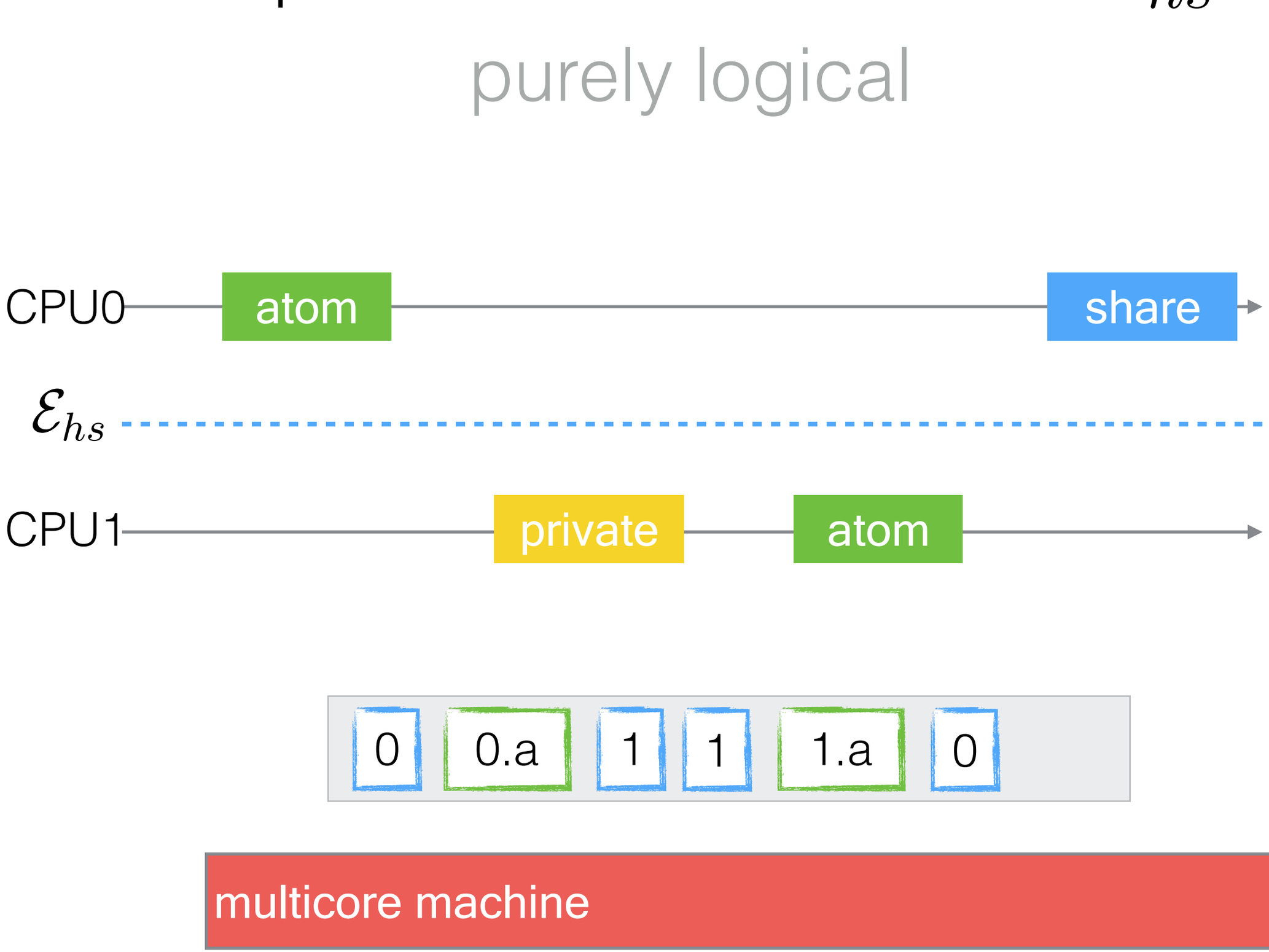
spin-lo

CPU-lo

multicore machine

step 1: hardware scheduler $\mathcal{E}_{hs}$

purely logical



trap

virt

proc

thread

thread

mem

spin-lo

CPU-lo

multicore machine

step 1: hardware scheduler
purely logical

multicore machine

trap

virt

proc

thread

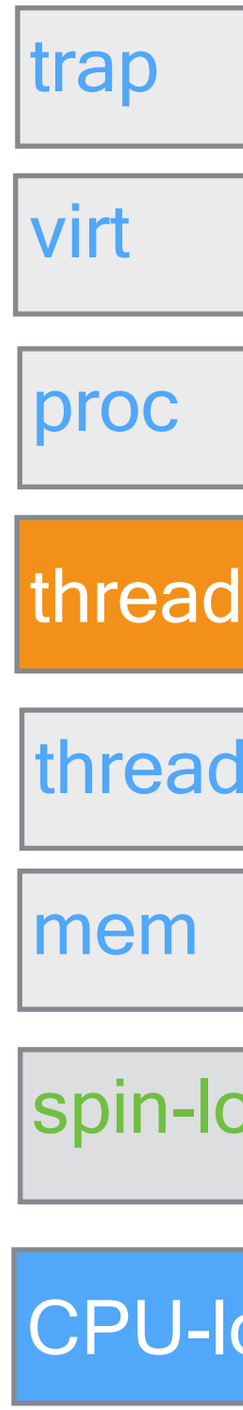
thread

mem

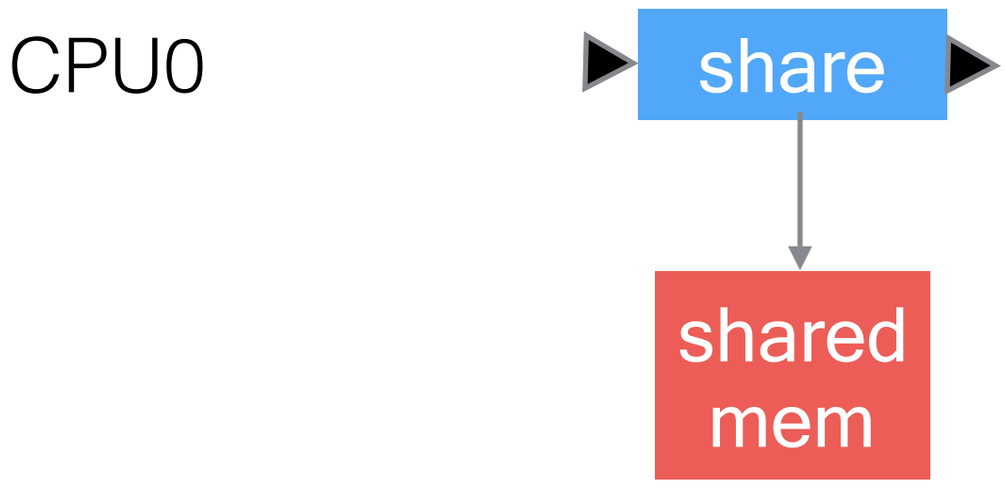
spin-lo

CPU-lo

step 1: hardware scheduler



step 2: push/pull model



trap

virt

proc

thread

thread

mem

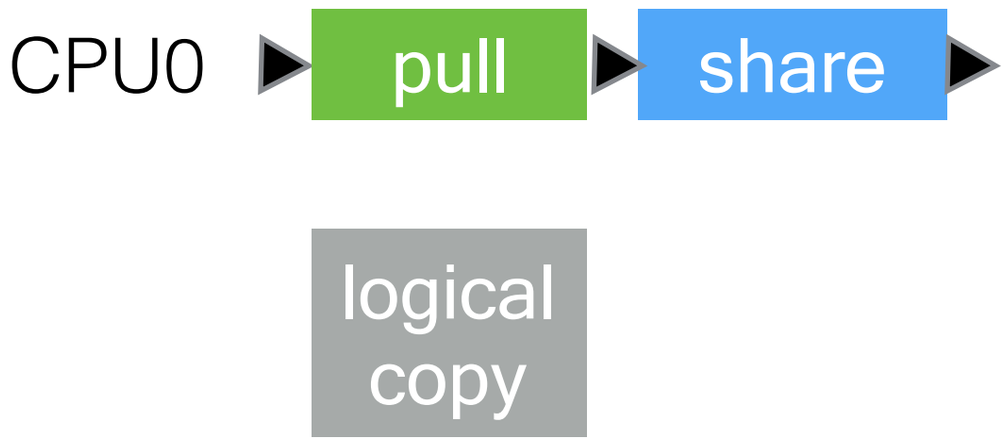
spin-lo

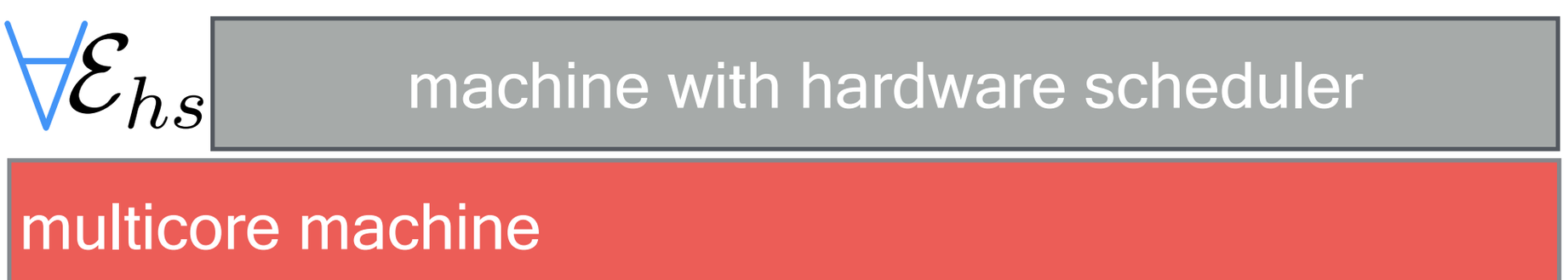
CPU-lo

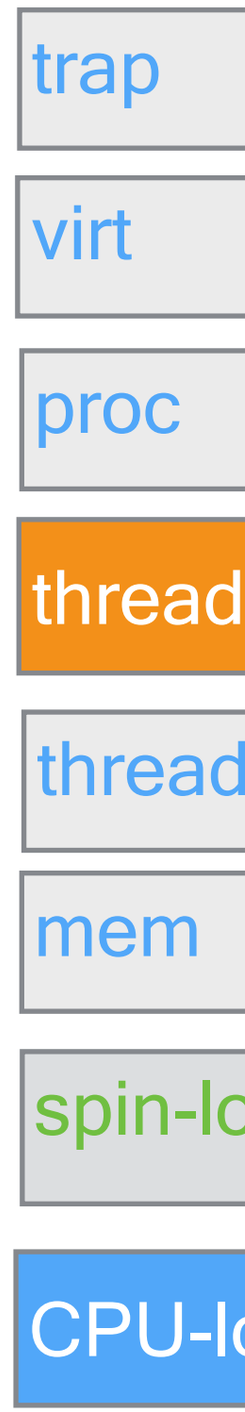
$\forall \mathcal{E}_{hs}$ machine with hardware scheduler

multicore machine

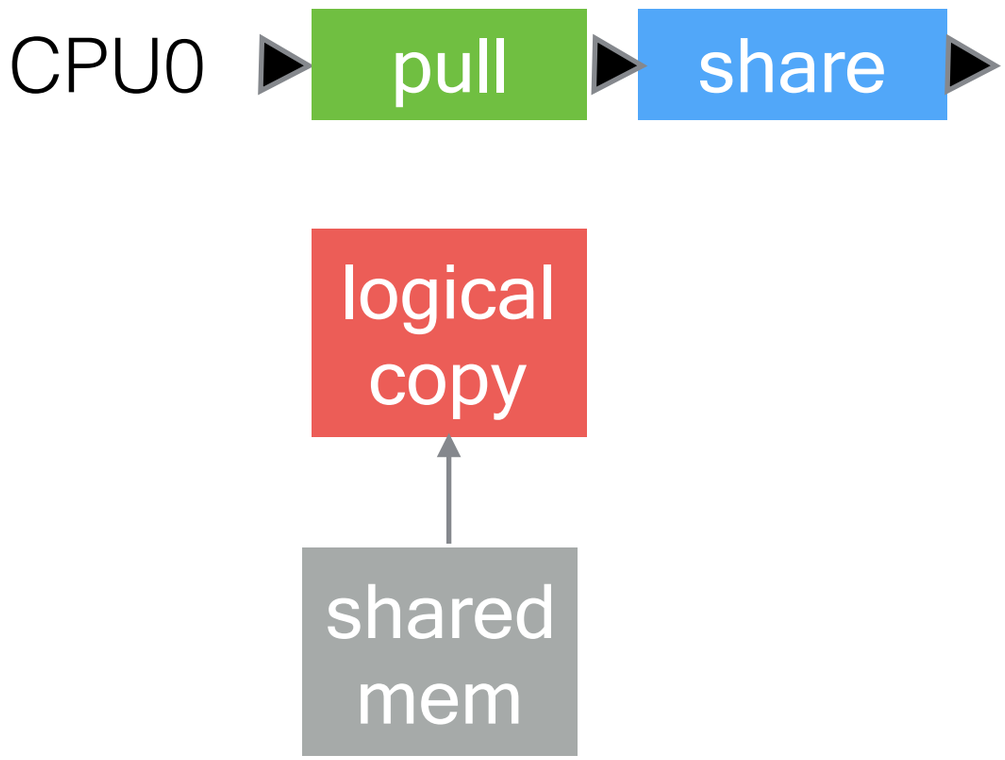
step 2: push/pull model

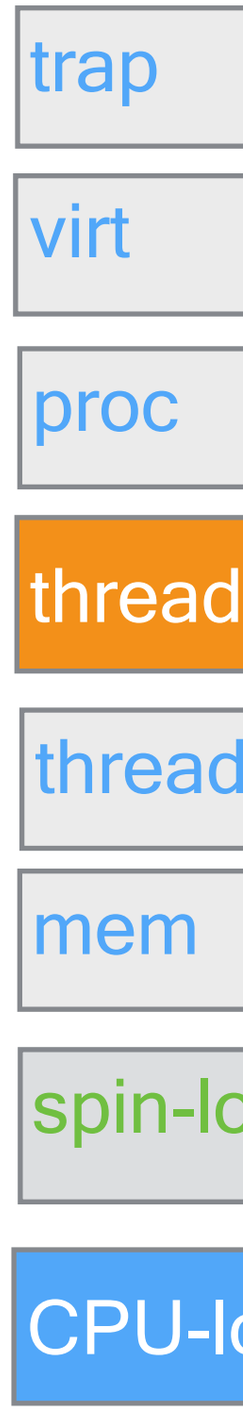




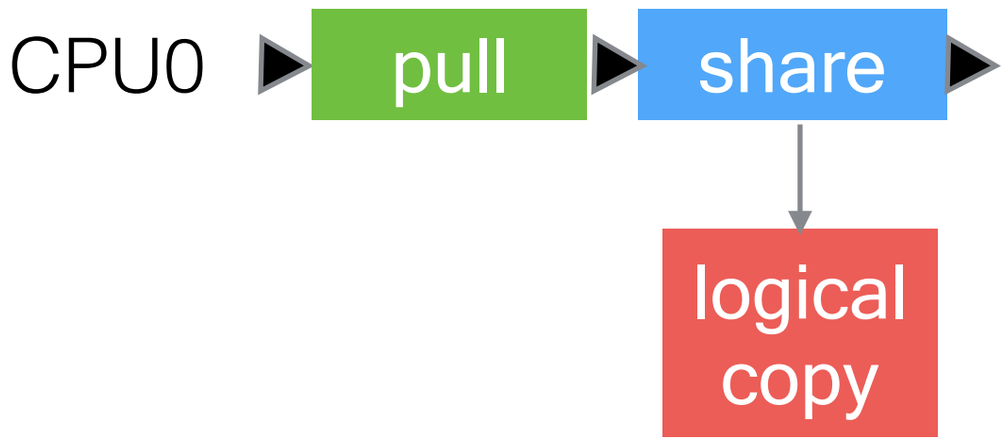


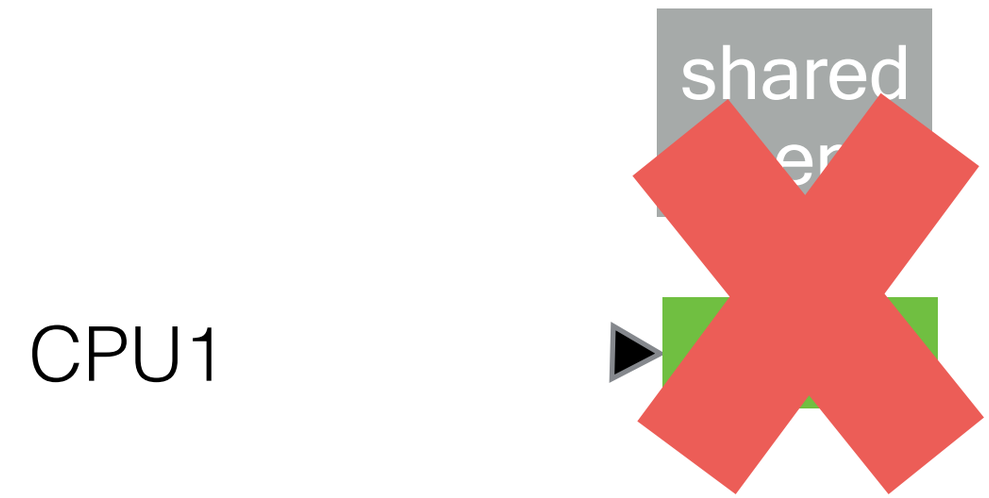
step 2: push/pull model





step 2: push/pull model





$\forall \mathcal{E}_{hs}$

machine with hardware scheduler

multicore machine

trap

virt

proc

thread

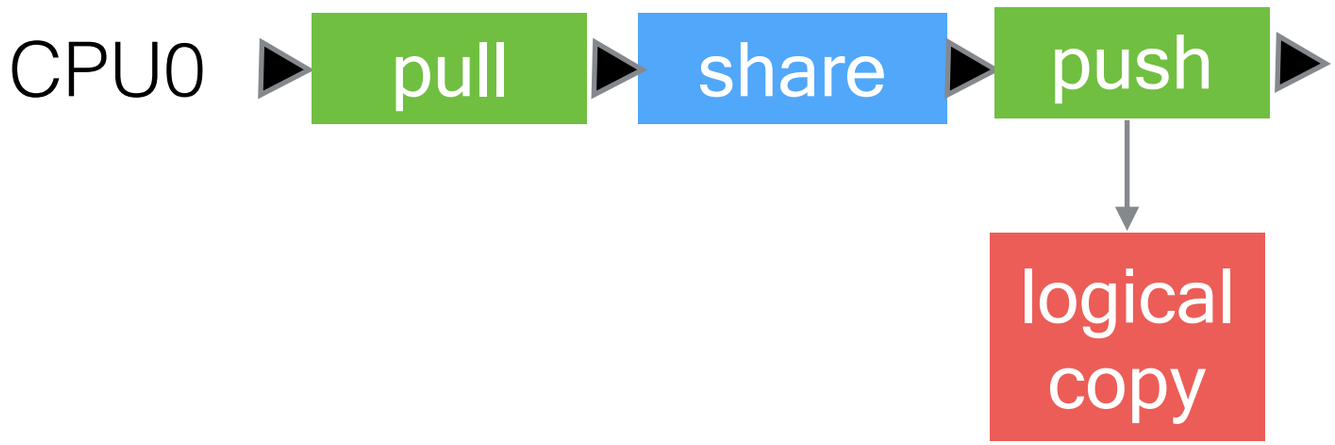
thread

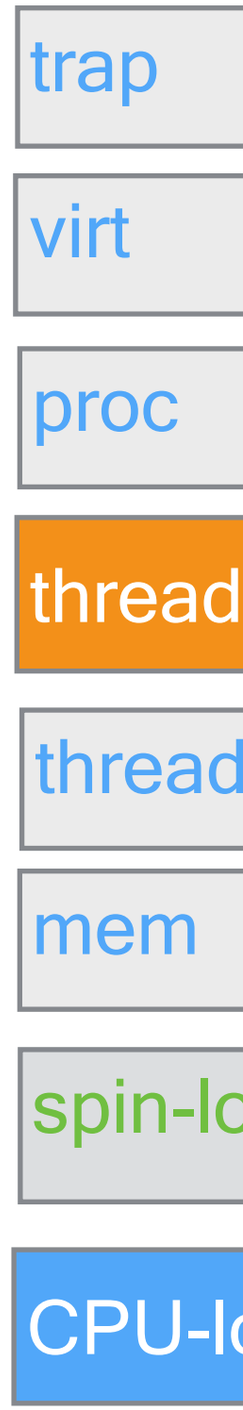
mem

spin-lo

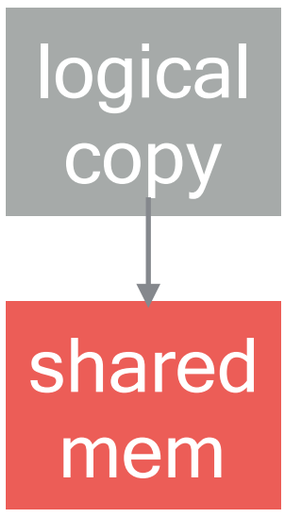
CPU-lo

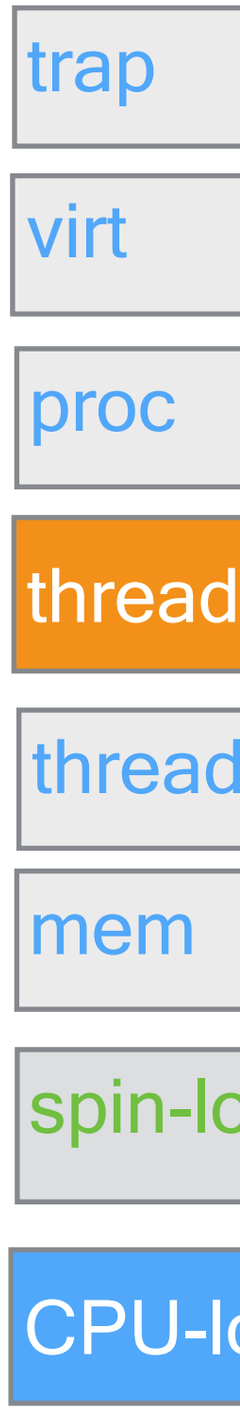
step 2: push/pull model





step 2: push/pull model





trap

virt

proc

thread

thread

mem

spin-lo

CPU-lo

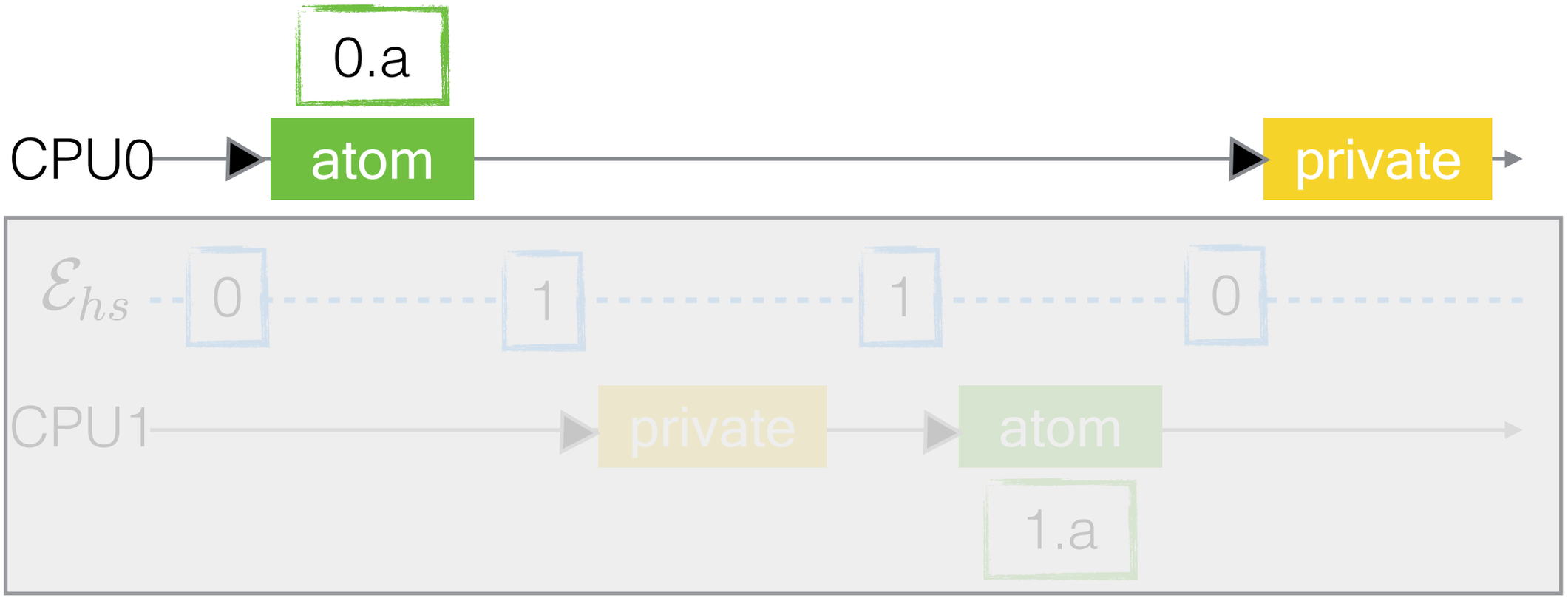
machine with local copy

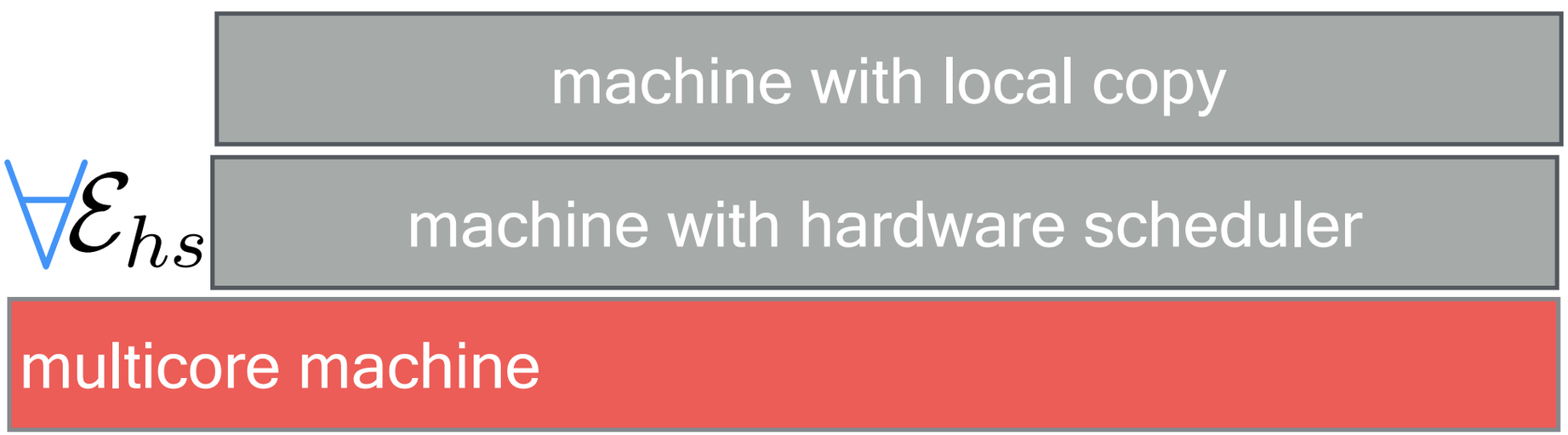
$\forall \mathcal{E}_{hs}$

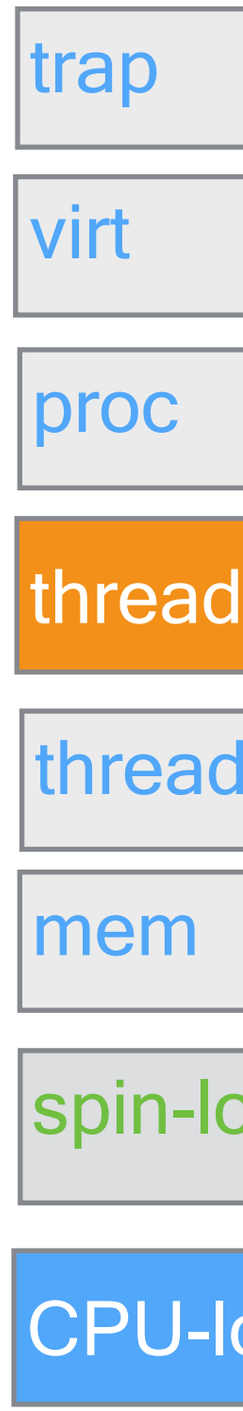
machine with hardware scheduler

multicore machine

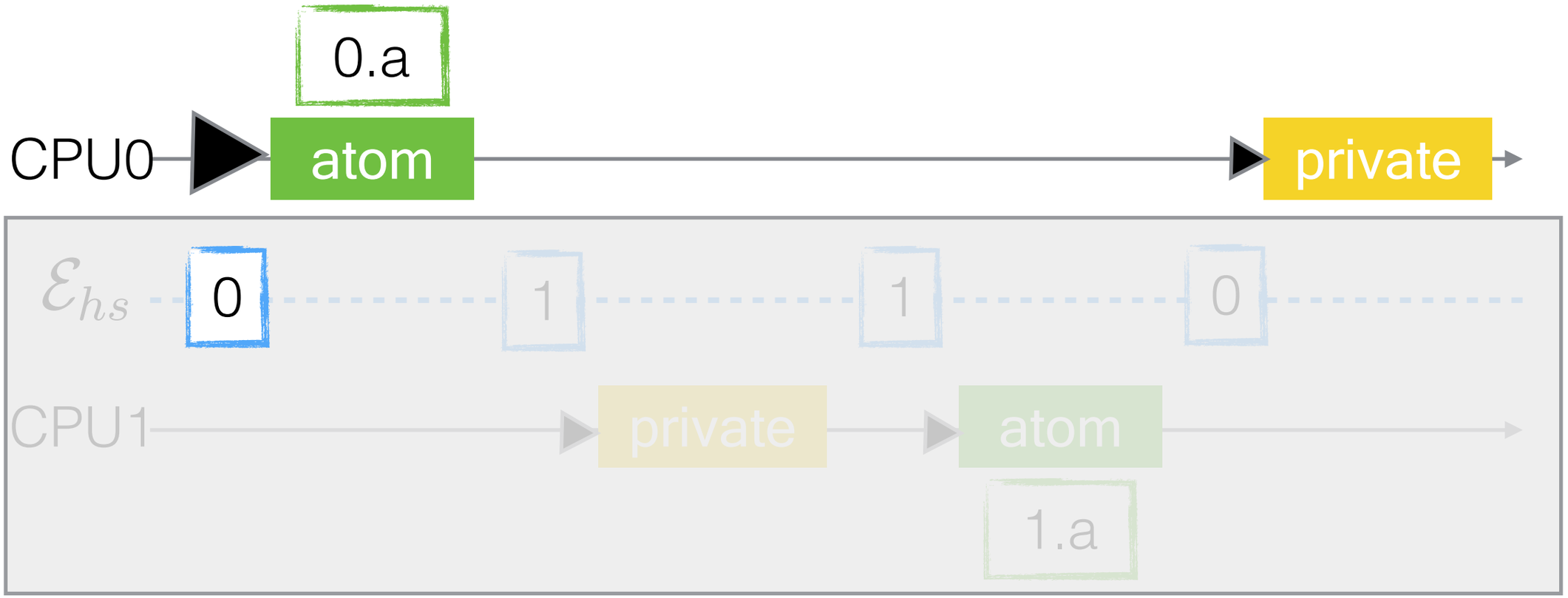
step 3: per-CPU machine







step 3: per-CPU machine



machine with local copy

$\forall \mathcal{E}_{hs}$ machine with hardware scheduler

multicore machine

trap

virt

proc

thread

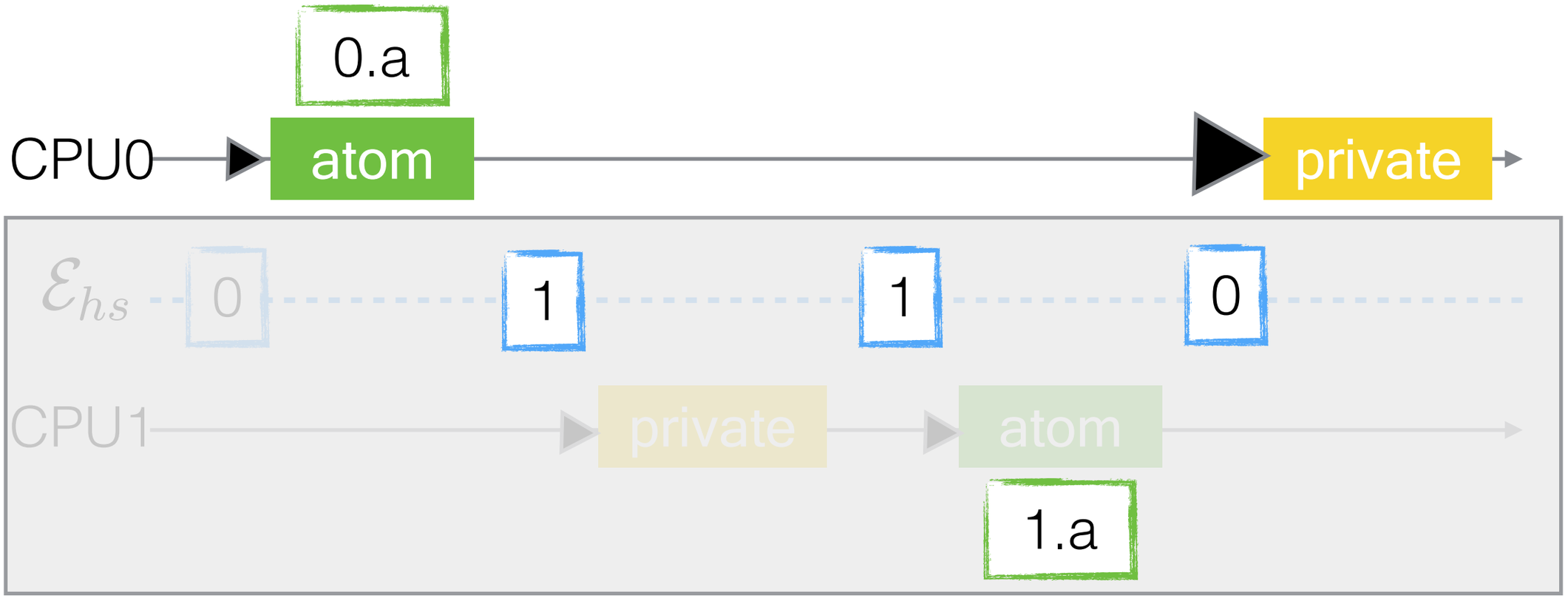
thread

mem

spin-lo

CPU-lo

step 3: per-CPU machine



machine with local copy

$\forall \mathcal{E}_{hs}$ machine with hardware scheduler

multicore machine

trap

virt

proc

thread

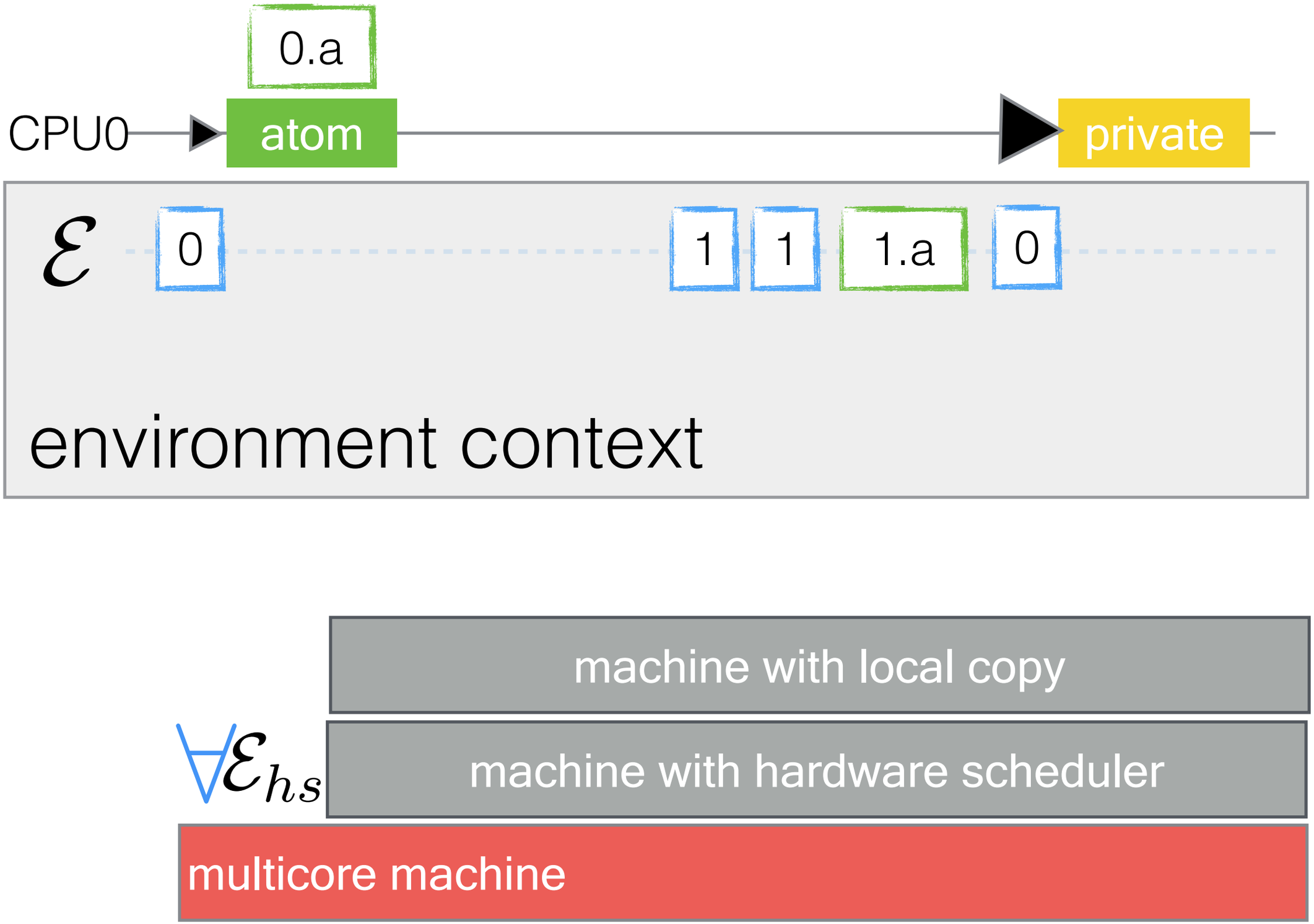
thread

mem

spin-lo

CPU-lo

step 3: per-CPU machine



trap

virt

proc

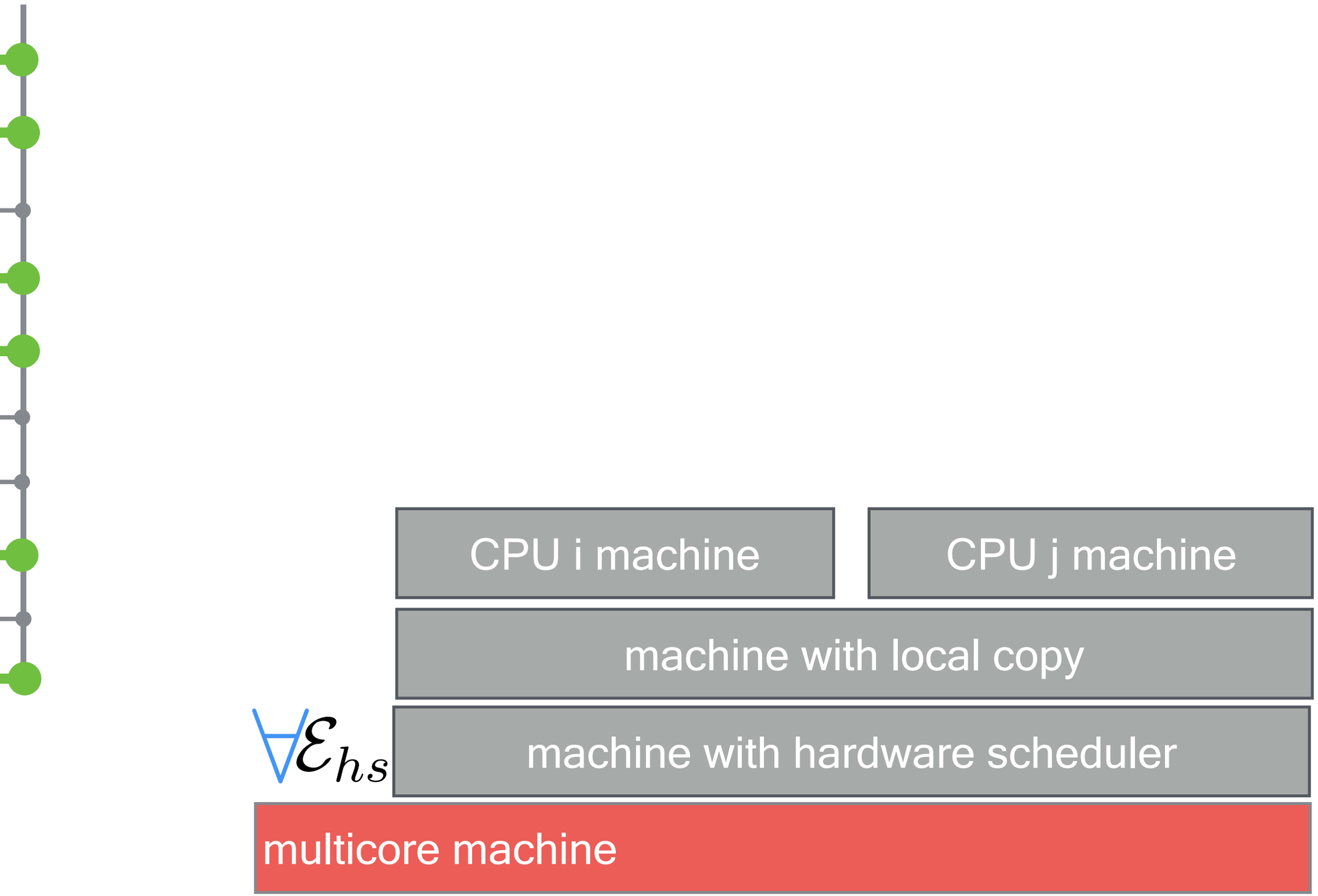
thread

thread

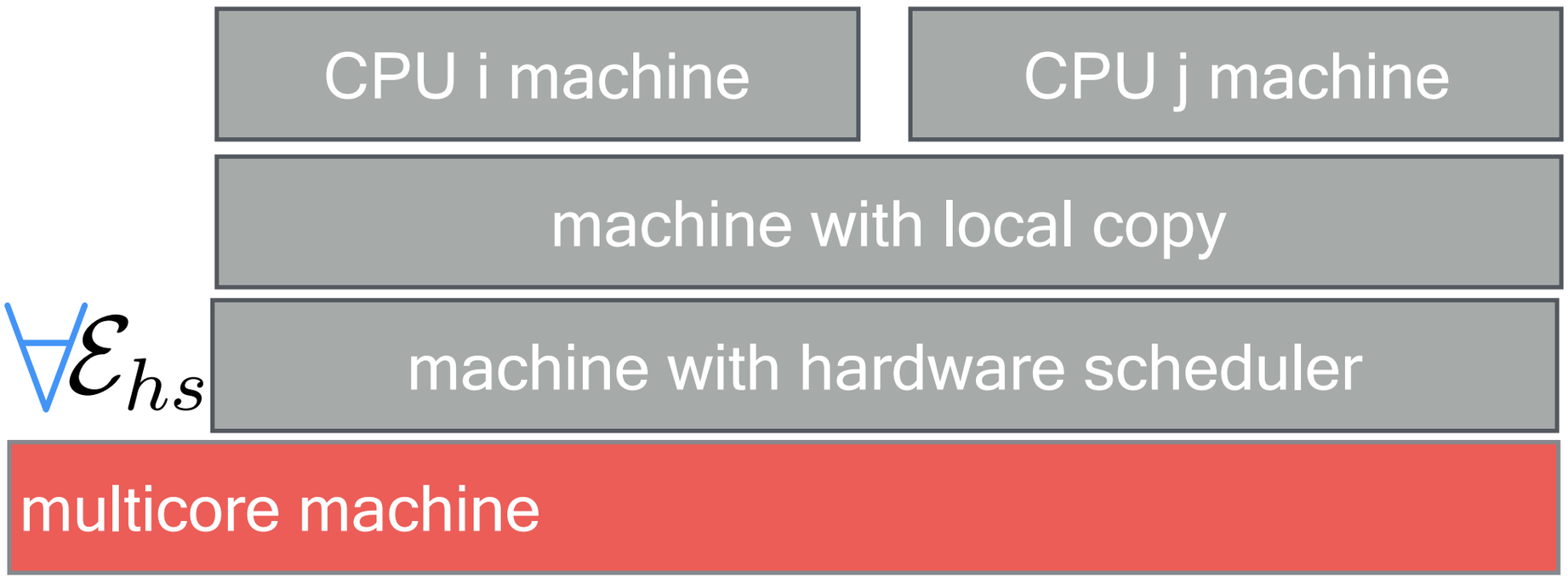
mem

spin-lo

CPU-lo


trap
virt
proc
thread
thread
mem
spin-lo
CPU-lo
CPU i machine
CPU j machine
machine with local copy
$\forall \mathcal{E}_{hs}$
machine with hardware scheduler
multicore machine

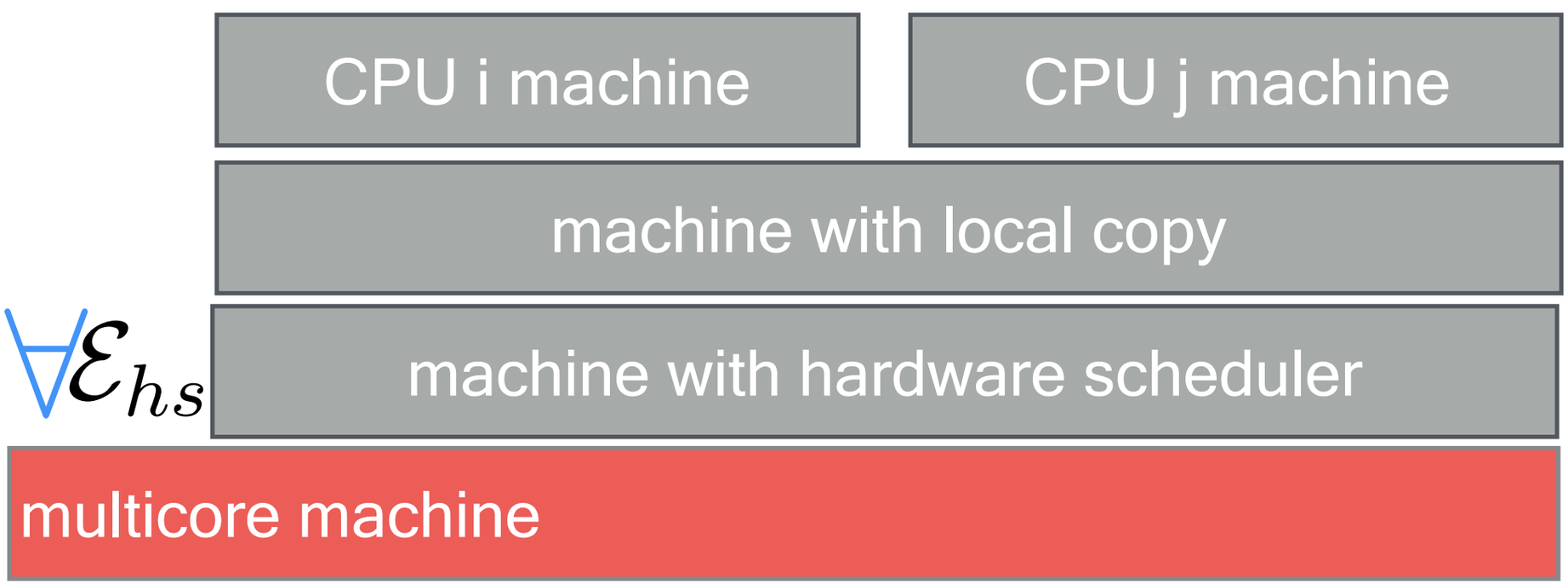
step 4: remove unnecessary interleaving



trap
virt
proc
thread
thread
mem
spin-lo
CPU-lo

step 4: remove unnecessary interleaving

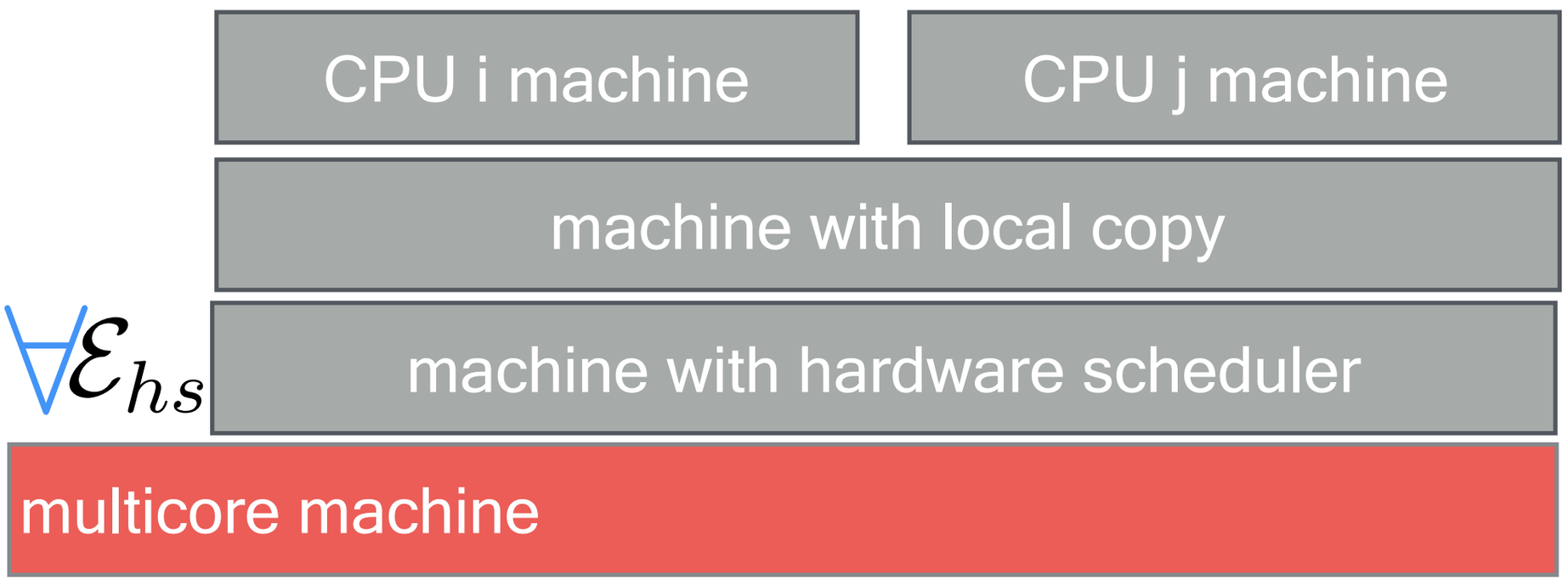
shuffle



trap
virt
proc
thread
thread
mem
spin-lo
CPU-lo

step 4: remove unnecessary interleaving

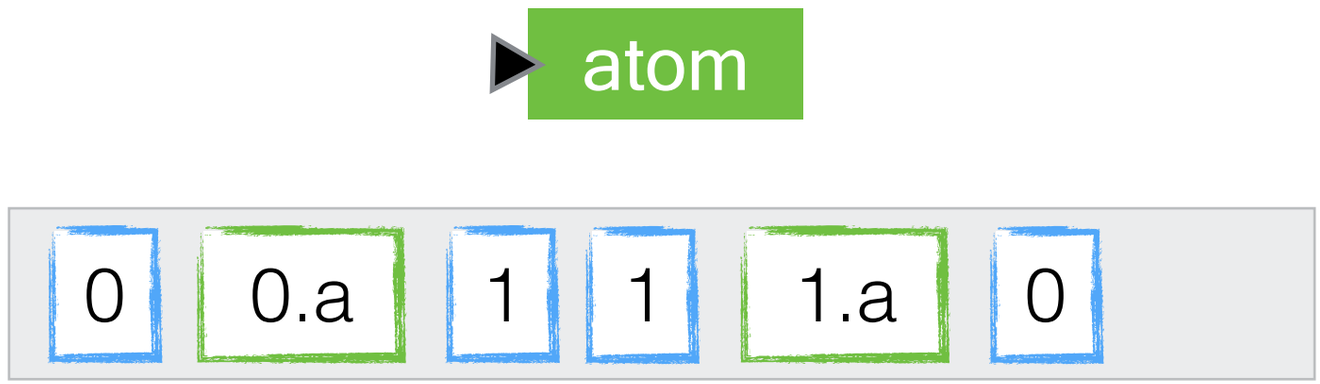
merge

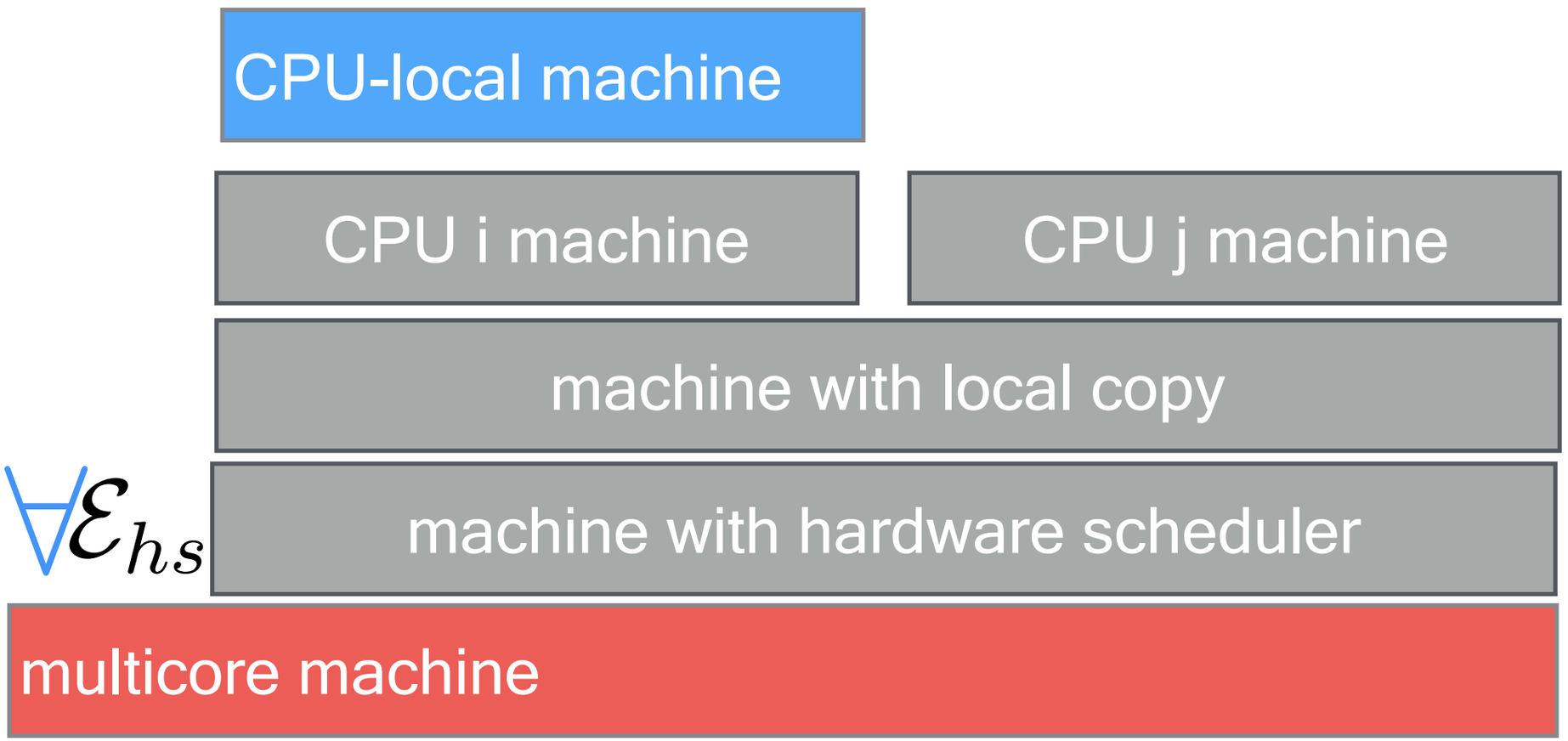


trap
virt
proc
thread
thread
mem
spin-lo
CPU-lo

contributions

- reuse





trap

virt

proc

thread

thread

mem

spin-lo

trap

virt

proc

thread

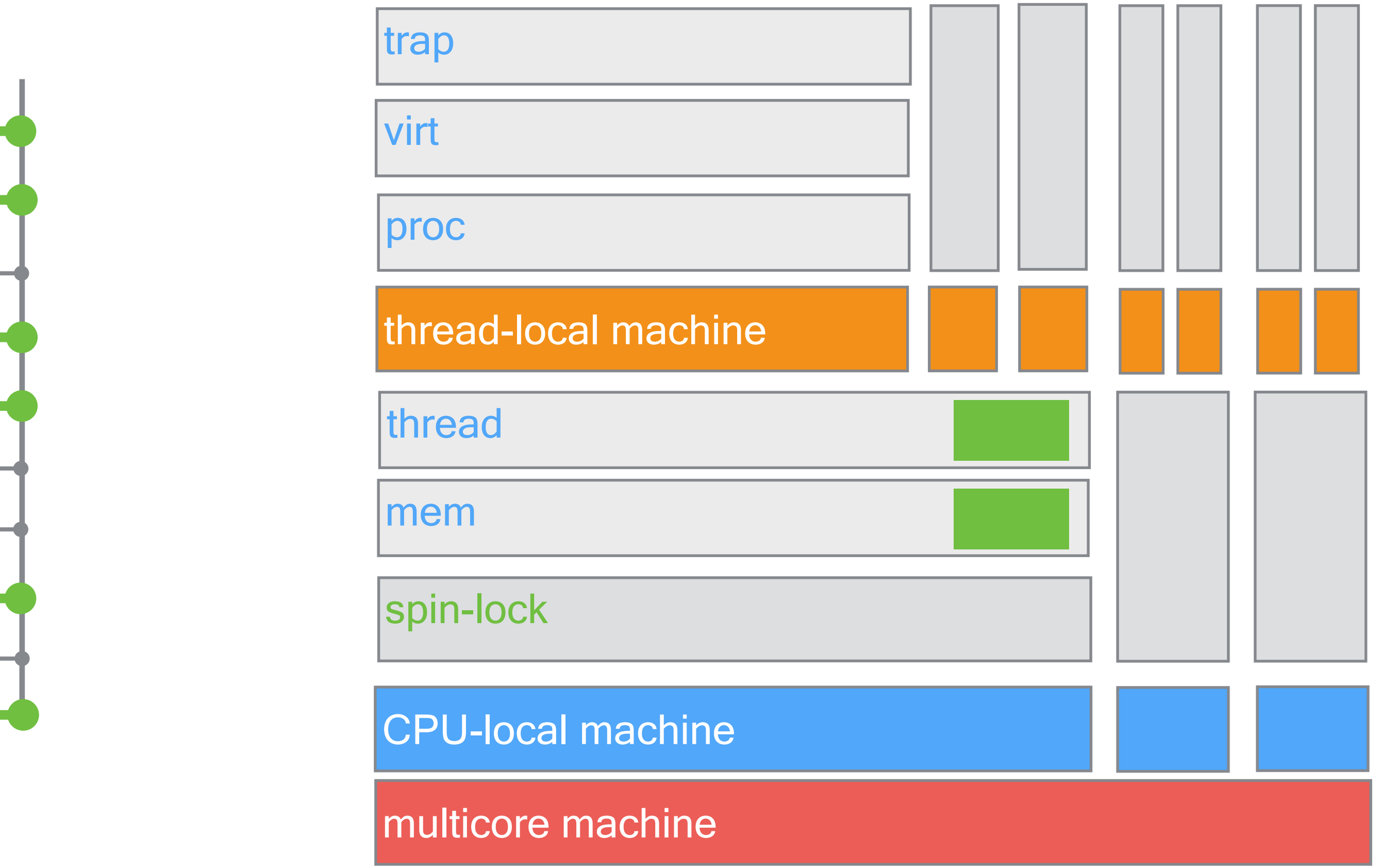
thread

mem

spin-lo

CPU-lo

multico

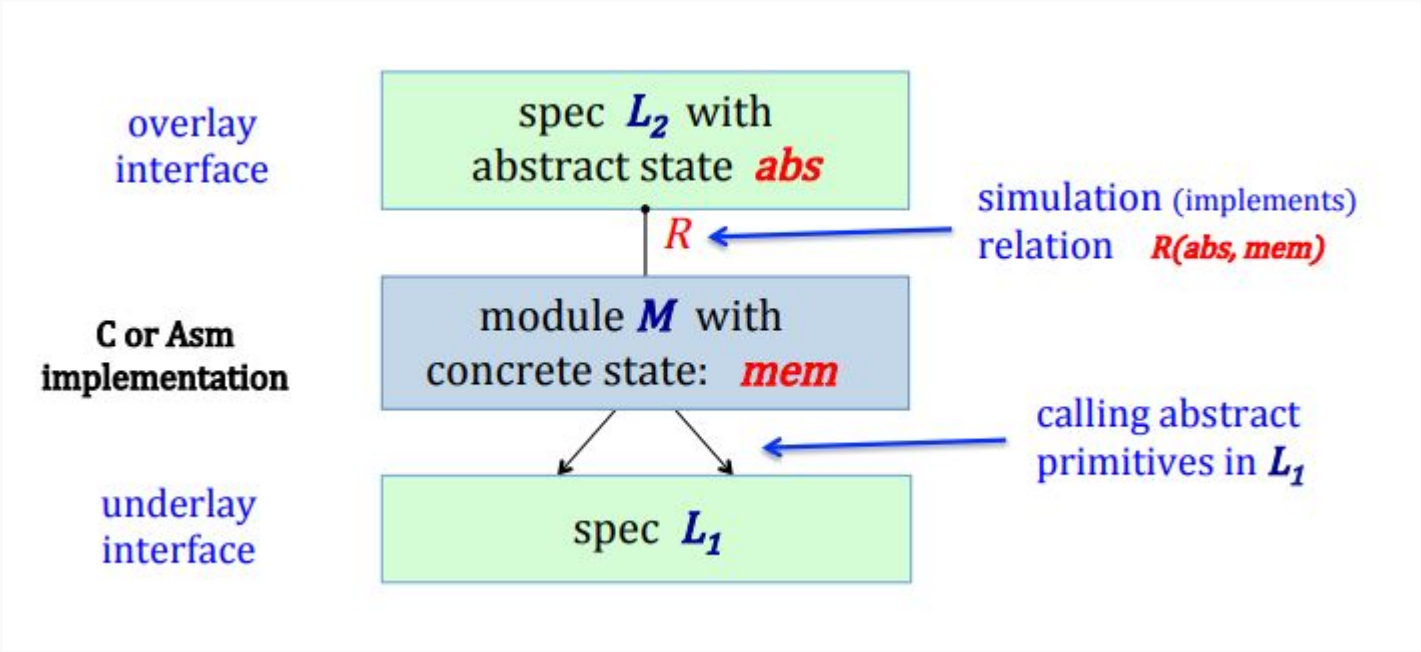

trap
virt
proc
thread-local machine
thread
mem
spin-lock
CPU-local machine
multicore machine

Certified Abstraction Layer

- A language construct (L_1, M, L_2) and a mechanized proof object
  - layer implementation M, built on top of the interface L1 (the underlay) is a contextual refinement of the desirable interface L2 above (the overlay).

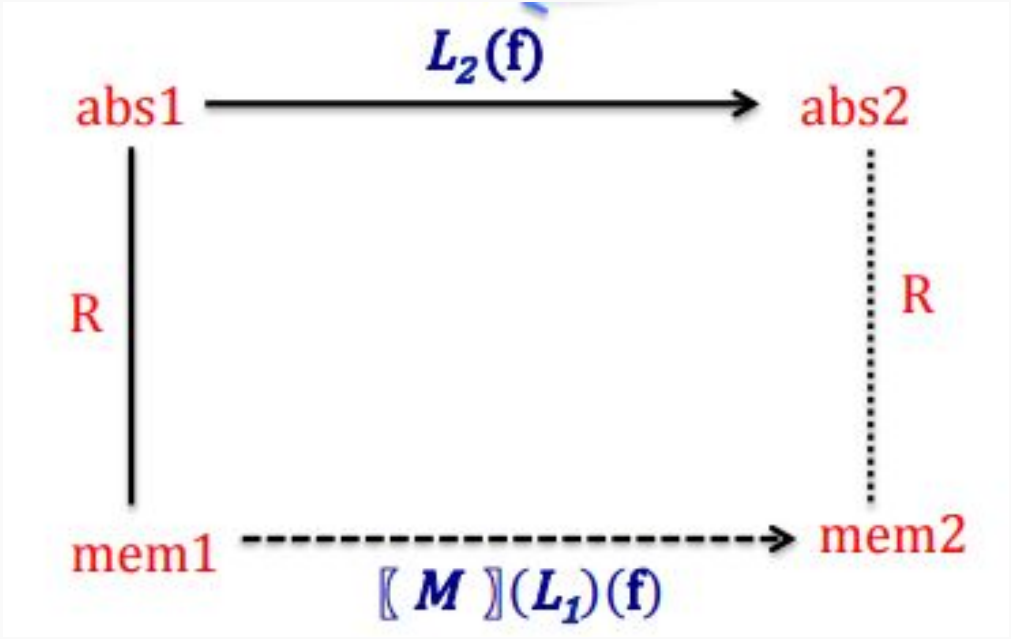
- A **deep specification** L_2 of a module *M* captures everything *contextually observable* about running the module over its underlay L1.
  - Once we have certified M with a deep specification L2, there is no need to look at M again.
  - Any property about M can be proved by L2 alone.

Certified Layers : (L1, M, L2) and a mechanized proof object

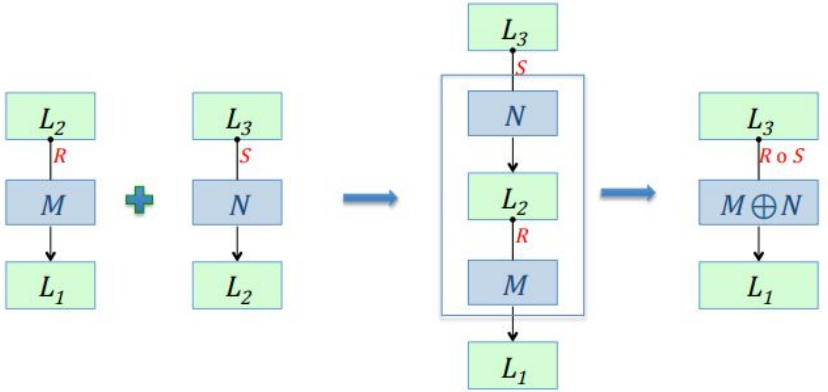


Implementation Independence

Simulation Relation



Composition of Deep Specifications



$$\frac{L_1 \vdash_R M : L_2 \qquad L_2 \vdash_S N : L_3}{L_1 \vdash_{R \circ S} M \oplus N : L_3}\ \textsc{Vcomp}$$

Layer Design

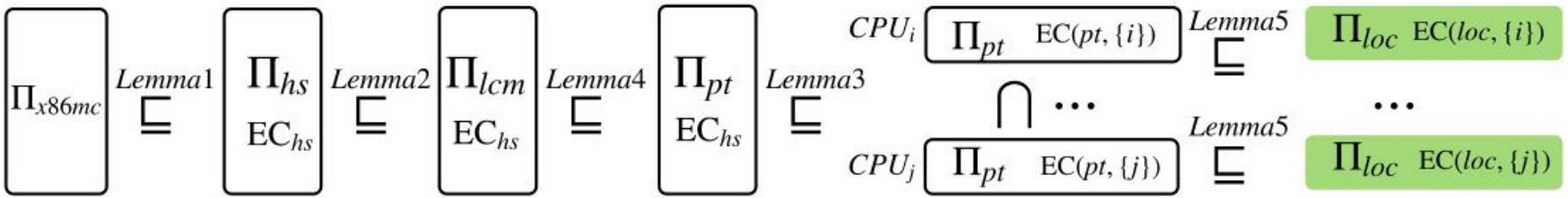


Figure 5: The contextual refinement chain from multicore hardware model Π_{x86mc} to CPU-local model Π_{loc}

Multicore Hardware Model

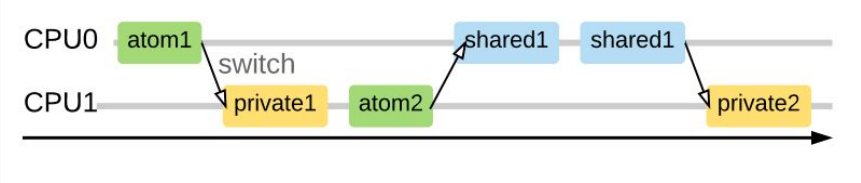
- Allows all CPUs to access the same piece of memory simultaneously
- Logically distinguish
  - Private Memory: single CPU/threads (no need to synchronize)
  - Shared Memory: multiple CPU/threads (synchronize using atomic hardware instructions)
- Each shared memory operation can be viewed as if it were atomic.

Atomic Object

- Abstraction of well-synchronized shared memory.
  - Set of primitives
  - Initial State
  - Logical log
- Current state can be Initial State + Replay log

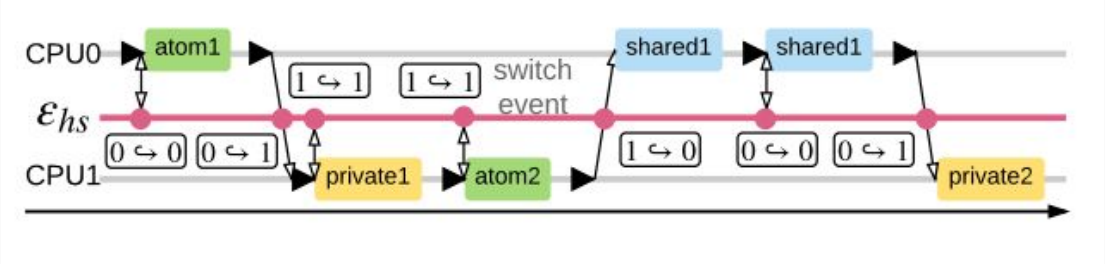
Step 0: Multicore Hardware Model Π_{x86mc}

- Allows arbitrary interleavings at the level of assembly instructions.
  - Hardware will non-deterministically choose 1 CPU.
  - Execute next assembly instruction on that CPU.
- Each instruction is classified as:
  - Atomic, Shared and Private
- Only atomic operations generate events.
  - Logical log = [0.atom1, 1.atom2]



Step 1: Model + Hardware Scheduler Π_{hs}

- Add hardware scheduler ε_{hs} that specifies a particular interleaving.
  - Deterministic machine model
- Insert **logical switch points.**
  - Query the scheduler to get the CPU id that will execute the next instruction.
  - All the switch decisions are stored in the log.

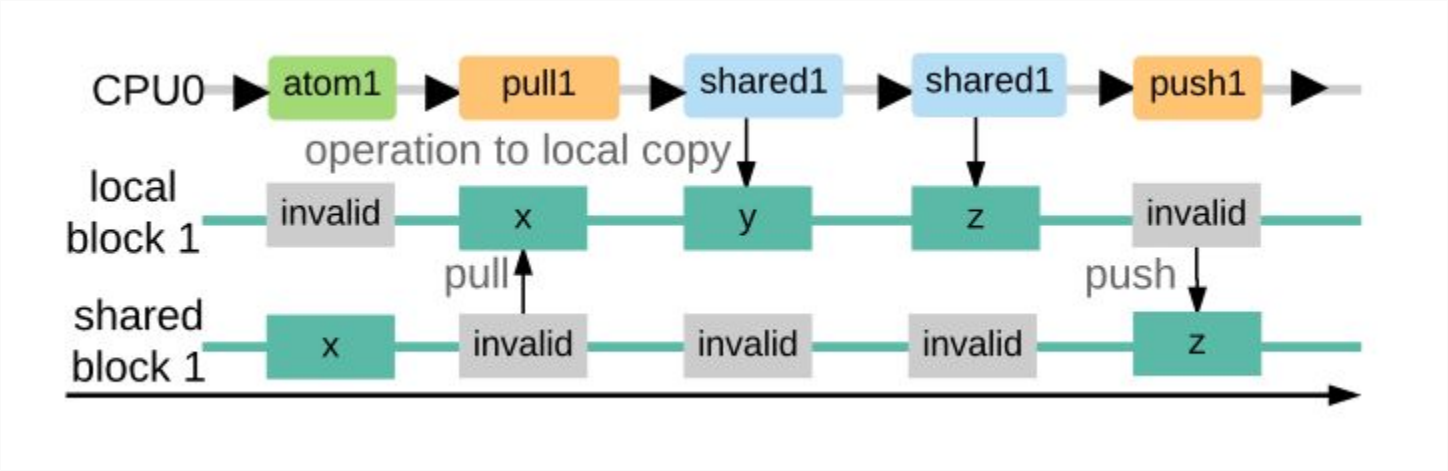


$$[0 \hookrightarrow 0, 0.\mathtt{atom}_1, 0 \hookrightarrow 1, 1 \hookrightarrow 1, 1 \hookrightarrow 1, 1.\mathtt{atom}_2, 1 \hookrightarrow 0, 0 \hookrightarrow 0, 0 \hookrightarrow 1]$$

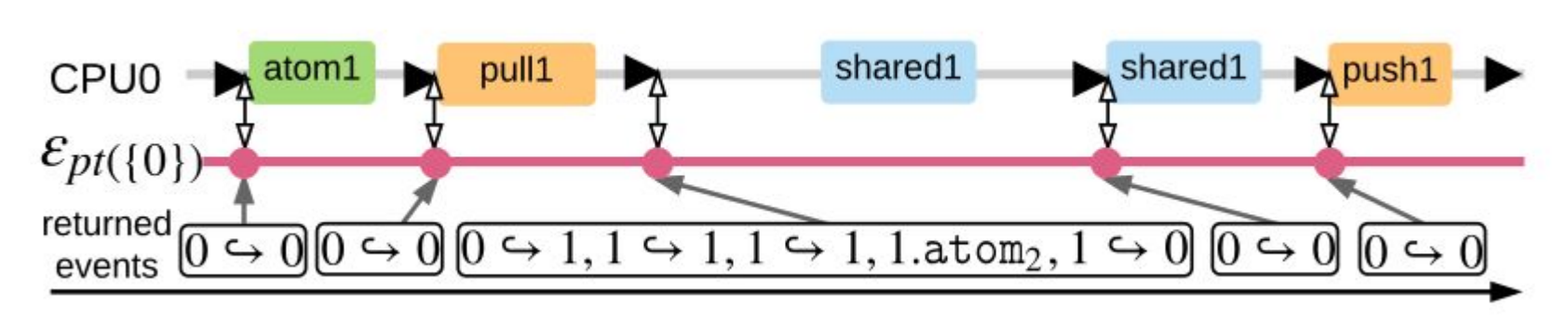
Contextual Refinement

- We say that layer L_0 contextually refines layer L_1 if and only if
  - For any P that does not go wrong on Π_{L1}
    - P does not go wrong with piLo on an configuration
- The behavior of running a program P over this model with a hardware scheduler ε_{hs} is denoted by $[[P]]_{hs} = \{ \Pi_{hs}(P, \varepsilon_{hs}) \mid \varepsilon_{hs} \in EC_{hs} \}$.
- Theorem:
  - $\forall$ P, $[[P]]_{x65mc}$ refines $[[P]]_{hs}$

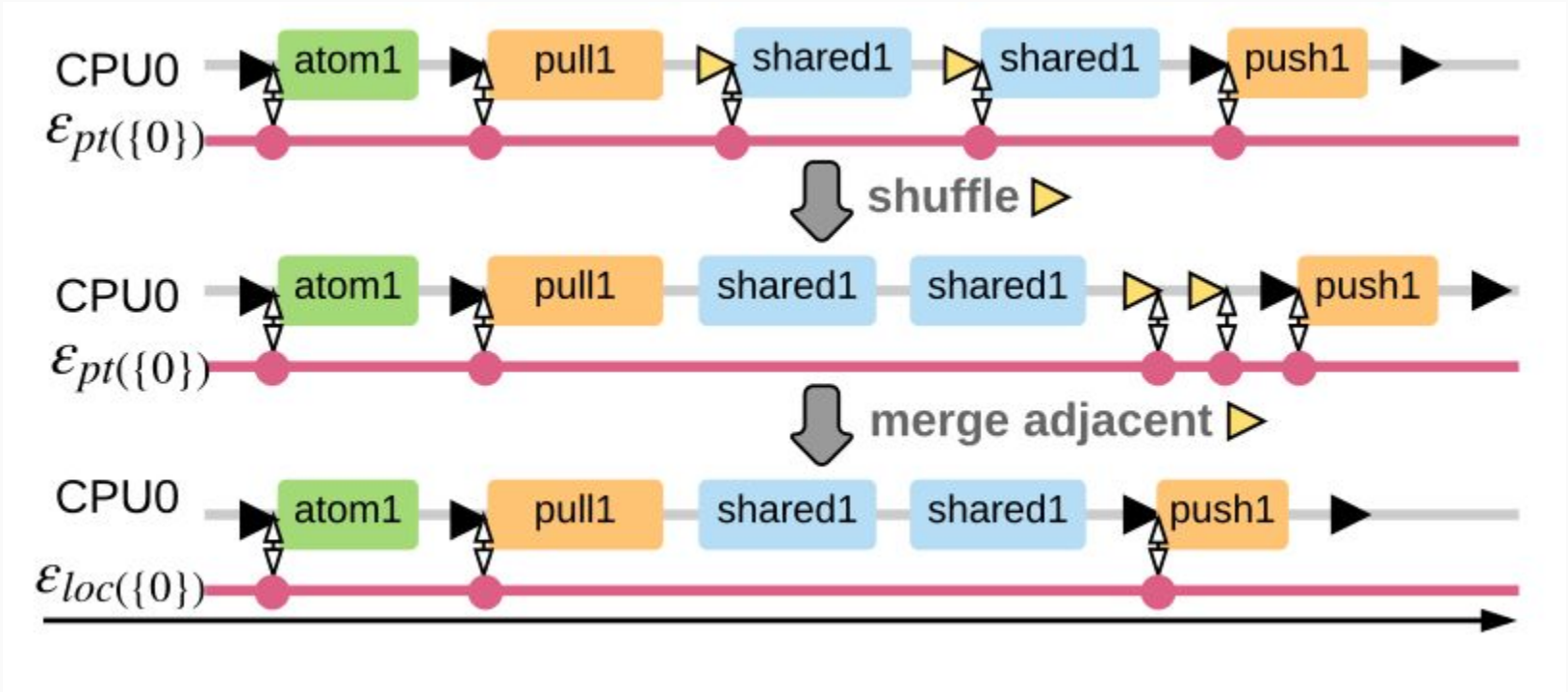
Step 2: Machine with local copy of shared memory



Step 3: Partial Machine with environment context



Step 4: CPU Local Machine Models



END ASIDE

Certifying the mC2 Kernel

- Layer Architecture
- System Architecture
- Spinlocks
- Memory Management
- Thread Management

System Architecture

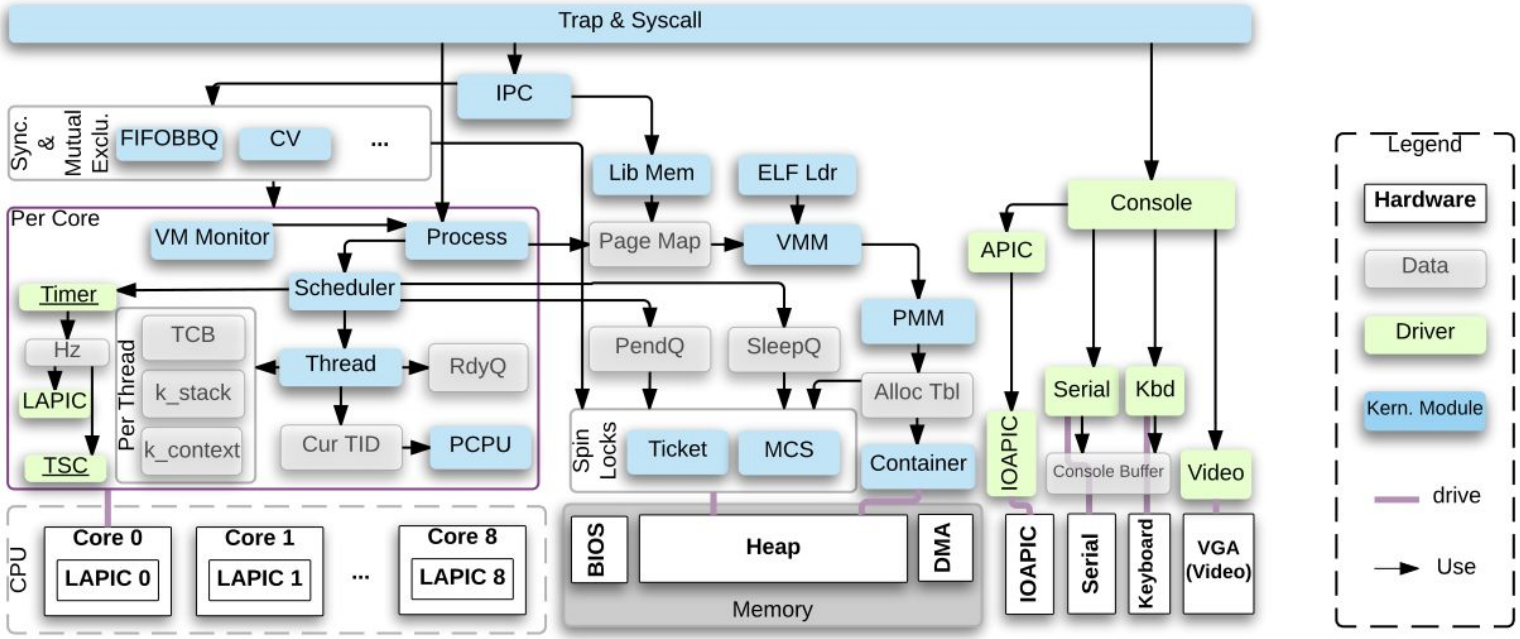


Figure 3: System architecture for the mC2 kernel

Layer Architecture

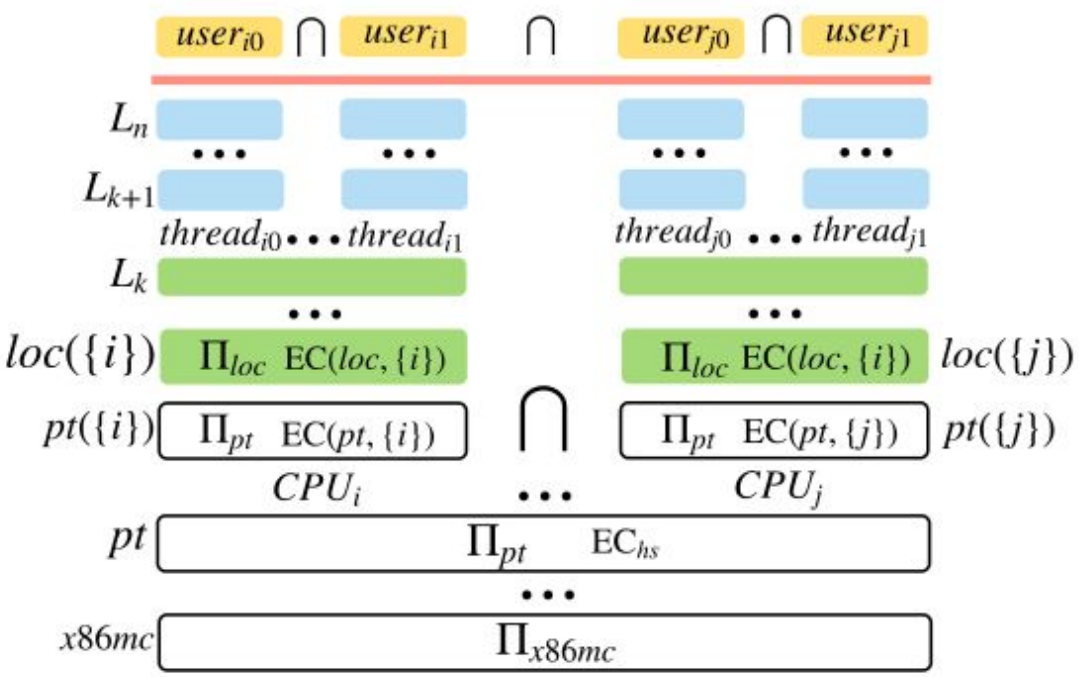
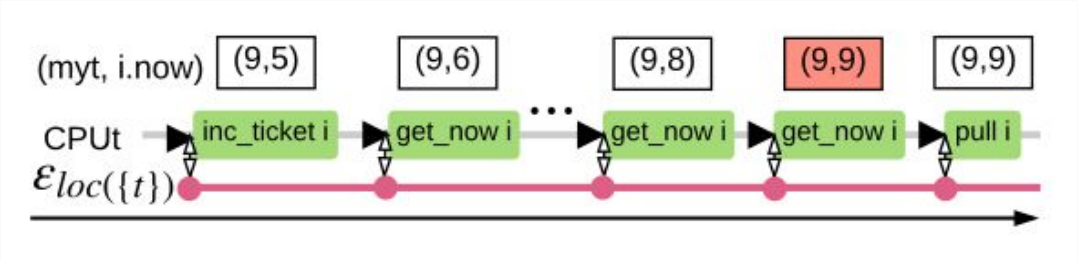


Figure 2: Contextual refinement between concurrent layers

Certifying the mC2 Kernel

- Design abstraction layers in a way
  - complex interdependent kernel components are untangled
  - well-organized object stack with clean specification.
- Bottom layer that connects to the CPU-local machine model Π_{loc}
  - instantiated with a particular active CPU.
- Trap handler is the top layer that provides system call interfaces
  - serves as a specification of the whole kernel
  - instantiated with a particular active thread running on that active CPU.
- Any global property proved at the top abstraction layer can be transferred down to the lowest hardware machine.

Spin lock module -- Ticket lock




```
1  typedef struct {
2    volatile uint ticket;
3    volatile uint now;
4  } ticket_lock;
5  ticket_lock L[NUM_LOCK];
6
7  void acq_lock (uint i) {
8    uint t;
9    t= ▶FAI(&L[i].ticket);
10   while( ▶L[i].now!=t){}
11    ▶pull (i);
12 }
13 void rel_lock (uint i) {
14    ▶push (i);
15    ▶L[i].now ++;
16 }
```

Figure 7: Pseudocode of the ticket lock implementation

Starvation Freedom for locks

- Invariant
  - Environment context that holds lock(i) will never acquire lock(i) again before releasing it.
  - Always releases lock(i) within k steps
- Acquiring the ticket-lock eventually succeeds
  - Mechanized in Coq
  - Proof idea : upper bound on the number of events generated by context of threads before the current one.
- MCS locks
  - Better scalability than ticket locks
  - Similar Proof for starvation freedom

Memory Management

- A thread acquiring a lower layer lock cannot acquire a lock defined at a higher layer.
  - Prevent deadlock
- Global invariants:
  - Paging is enabled only after all the page maps are initialized
  - Pages that store kernel specific data have kernel only permission in all page maps
  - The kernel page map is an identity map
  - Non shared parts of user processes’ memory are isolated

Page Allocation -- Allocation Table, Quotas and Locks

```
1  int palloc (uint tid) {
2    if (cn[tid].quota < 1)
3      return ERROR;
4    ▶acq_lock (lock_AT);
5    uint i=0,fp=nps;
6    while(fp==nps&&i<nps){
7      if (!AT[i].free)
8        fp = i;
9      i++; }
10   if (fp != nps) {
11     AT[i].free = 0;
12     AT[i].ref = 1;
13     cn[tid].quota --;
14   }
15   else fp = ERROR;
16   ▶rel_lock (lock_AT);
17   return fp;
18 }
```

Figure 8: Pseudocode of `palloc`

Thread Management and Scheduling -- yield, sleep and wakeup

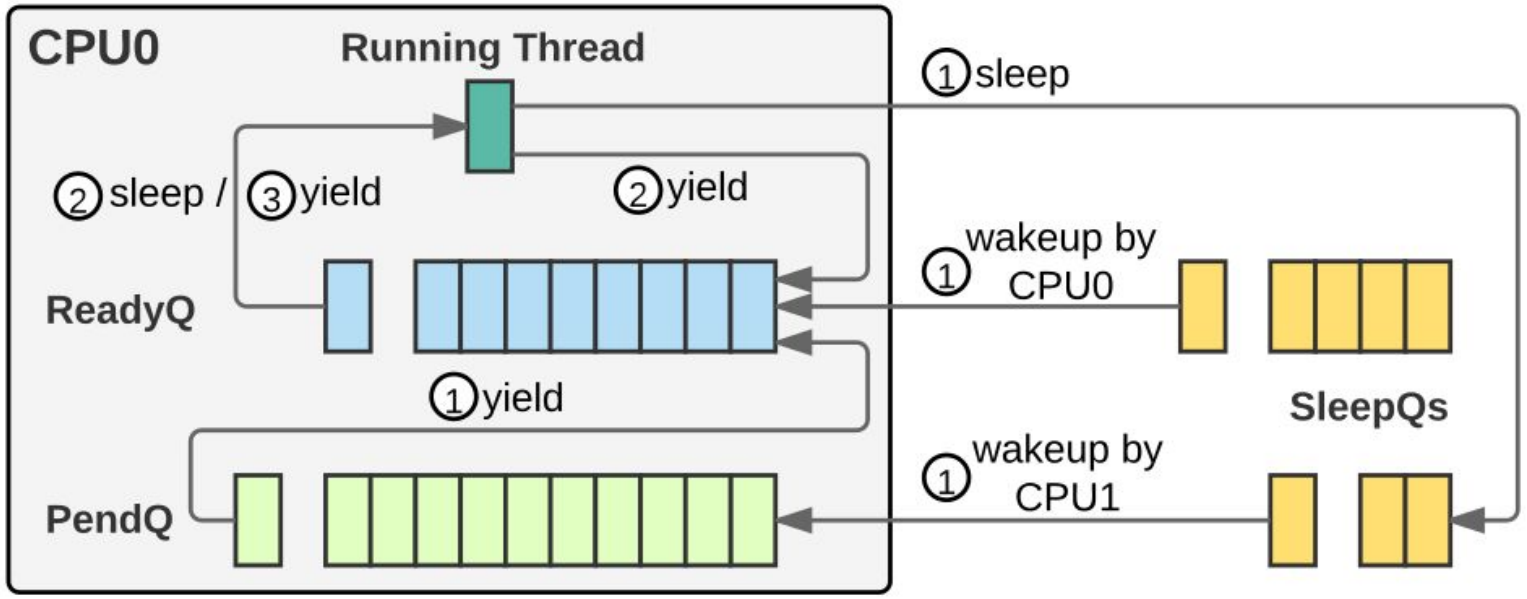


Figure 9: Scheduling routines `yield`, `sleep`, and `wakeup`

Evaluation

Contents

- Proof effort and cost of change
- Performance evaluation
- Concurrency overhead
- IPC Performance
- Hypervisor Performance

Proof Effort and Cost of Change

- Code : 6500 lines of C + x86 assembly over 2 person years
- Specification (these are part of the trusted computing base):
  - 943 lines of code for the lowest layer axiomatizing the hardware machine model.
  - 450 lines of code specifying the abstract system call interfaces.
- 5249 lines of additional specifications
  - For auxiliary definitions, theorems, invariants etc.
- 50k lines of Coq proofs.
  - At least ⅓ of this is semi-automatically generated and redundant.

Performance Evaluation

- Not the main concern, but still run benchmarks.
- Since power control code has not been verified
  - Turbo boost and power management features of the hardware have been disabled.

Concurrency Overhead

- Runtime overhead is dominated by:
  - Latency of the spinlocks
  - Contention of shared data.
- All shared objects pre-allocated a fine-grained lock.
- MCS locks vs Ticket Locks
  - Efficiency remains the same for ticket locks but increases for the MCS locks.

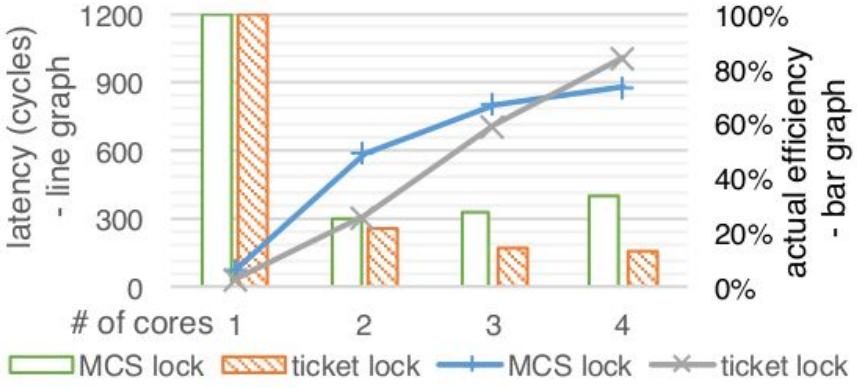


Figure 11: The comparison between actual efficiency of ticket lock and MCS lock implementations in mC2

IPC Overhead mC2 vs seL4

- seL4 fastpath/slowpath:
  - 1200/1800 cycles
- Mc2 IPC:
  - ~3800 cycles
- seL4 is optimized and tailored for hardware performance.

Hypervisor Performance

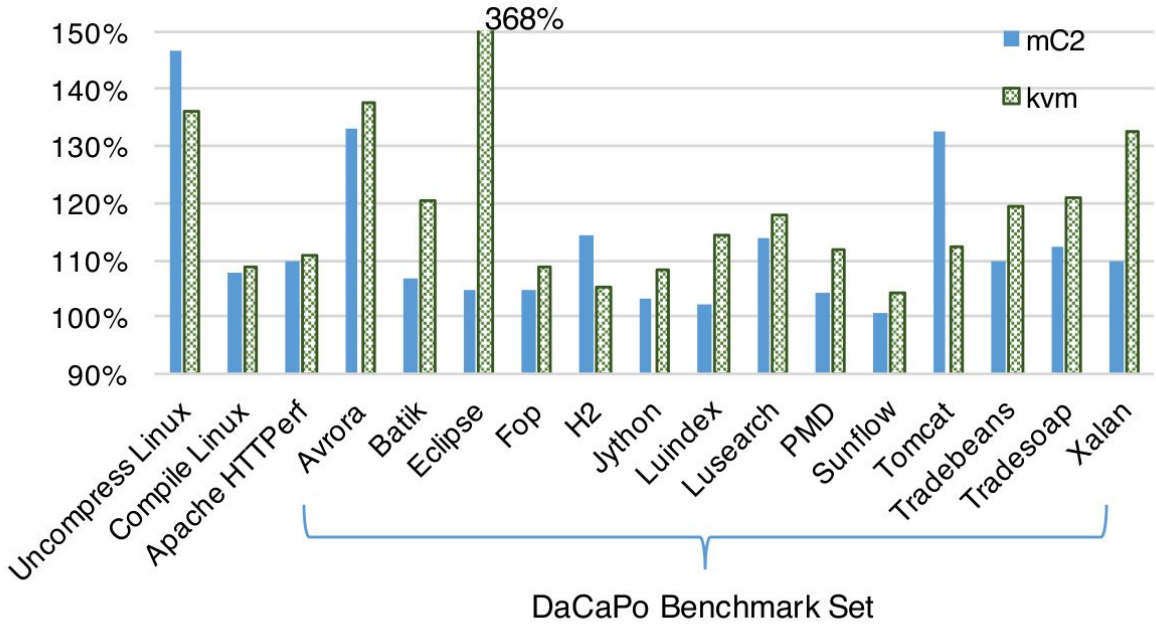


Figure 13: Normalized performance for macro benchmarks running over Linux on KVM vs. Linux on mC2; the baseline is Linux on bare metal; a smaller ratio is better

Conclusions

- Extensible architecture
- Clean, Rigorous, Layered, Practical