Real-Time CertiKOS: A Step Toward Resource Adaptive Certified OS Kernels

Zhong Shao
Yale University

June 18, 2018

Joint work with Mengqi Liu, Lionel Rieg, David Costanzo, and Ronghui Gu

Motivation

How to build resource adaptive certified OS kernels?

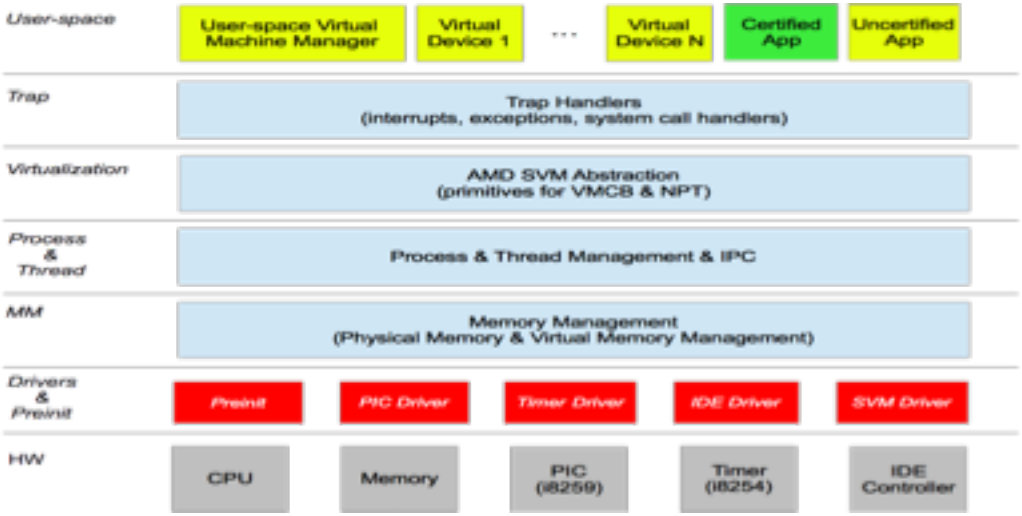
- **Security of partitions**: running multiple components on one processor requires proper isolation
  - Spatial isolation: memory quotas, process id randomization, etc.
  - Oblivion to context switches: no information leakage (proper saving/restoration of state)
  - **Temporal isolation**: preemption is required to assign CPU resources in a controlled manner

- **Hard real-time**: end-to-end guarantee of all tasks meeting deadlines is non-trivial
  - Tasks are closely intertwined: higher priority tasks could starve lower priority ones
  - Priority inversion: exclusive access to shared memory could result in lower priority tasks preempting higher priority ones

Background: mCertiKOS (2015)

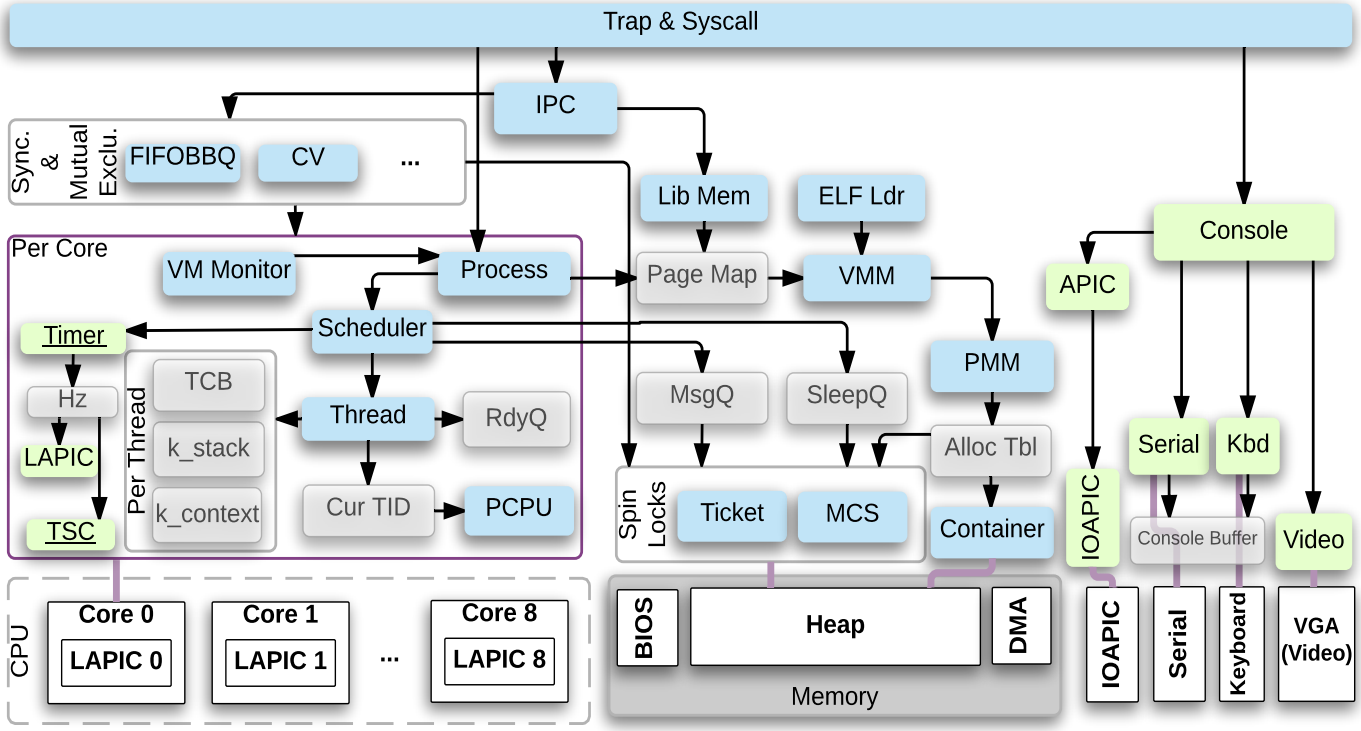
Single-core version of CertiKOS (developed under DARPA CRASH & HACMS programs), 3 kloc, can boot Linux

Aggressive use of abstraction over deep specs (37 layers in ClightX & LAsm)



Concurrent CertiKOS [OSDI'16, PLDI'18]

The first fully certified concurrent kernel w. fine-grained locking



Also see tomorrow afternoon's talk on multithreaded and multicore linking by Jieung Kim and Friday's PLDI talk on CCAL by Ronghui Gu

This Talk: Real-Time CertiKOS

- Extended the CertiKOS machine model with interrupts and timing registers, and obtained a formal proof of functional correctness of a preemptive OS kernel
- A new framework for information flow security of a preemptive OS kernel, and a formal non-interference proof of a preemptive OS kernel with a fixed-priority real-time scheduler
- Formal description of real-time properties, and the *virtual timeline* framework for hard real-time reasoning

Rest of The Talk

1. Modeling preemption
2. Information flow security
3. Hard real-time

Modeling Preemption

Preemption in the user mode

- Practical: time consuming services could be performed in the user space, resulting in low latency
- Economical: kernel verification is not affected much

Modeling Preemption

Hardware mechanism

- TSC (Time Stamp Counter) register: common on x86 architectures, incrementing at a constant rate on many models, at a granularity comparable to the cycle interval
- Interrupt: triggered by external devices, checked by the processor before executing each instruction

Modeling Preemption

LAsm machine model

- State rs m: the state consists of register set (rs) and memory (m)
- TSC register: the TSC register is introduced to keep track of the passage of time
- Intr_trigger: it is parameterized with a cost model such that execution of each instruction will increment the TSC, which may (or may not) trigger timer interrupts
- Intr_handler: jumps to the corresponding handler if interrupt is triggered, nop otherwise

Modeling Preemption

LAsm machine model

- cooperative

……

exec_instr i rs m = Next rs' m'

---

step (State rs m) (State rs' m')

enable interrupts
in user mode

- preemptive

…… kern_mode m

exec_instr i rs m = Next rs' m'

---

step (State rs m) (State rs' m')

…… ~kern_mode m

intr_trigger rs m = Next rs_1 m_1

exec_instr i rs_1 m_1 = Next rs_2 m_2

intr_handler rs_2 m_2 = Next rs' m'

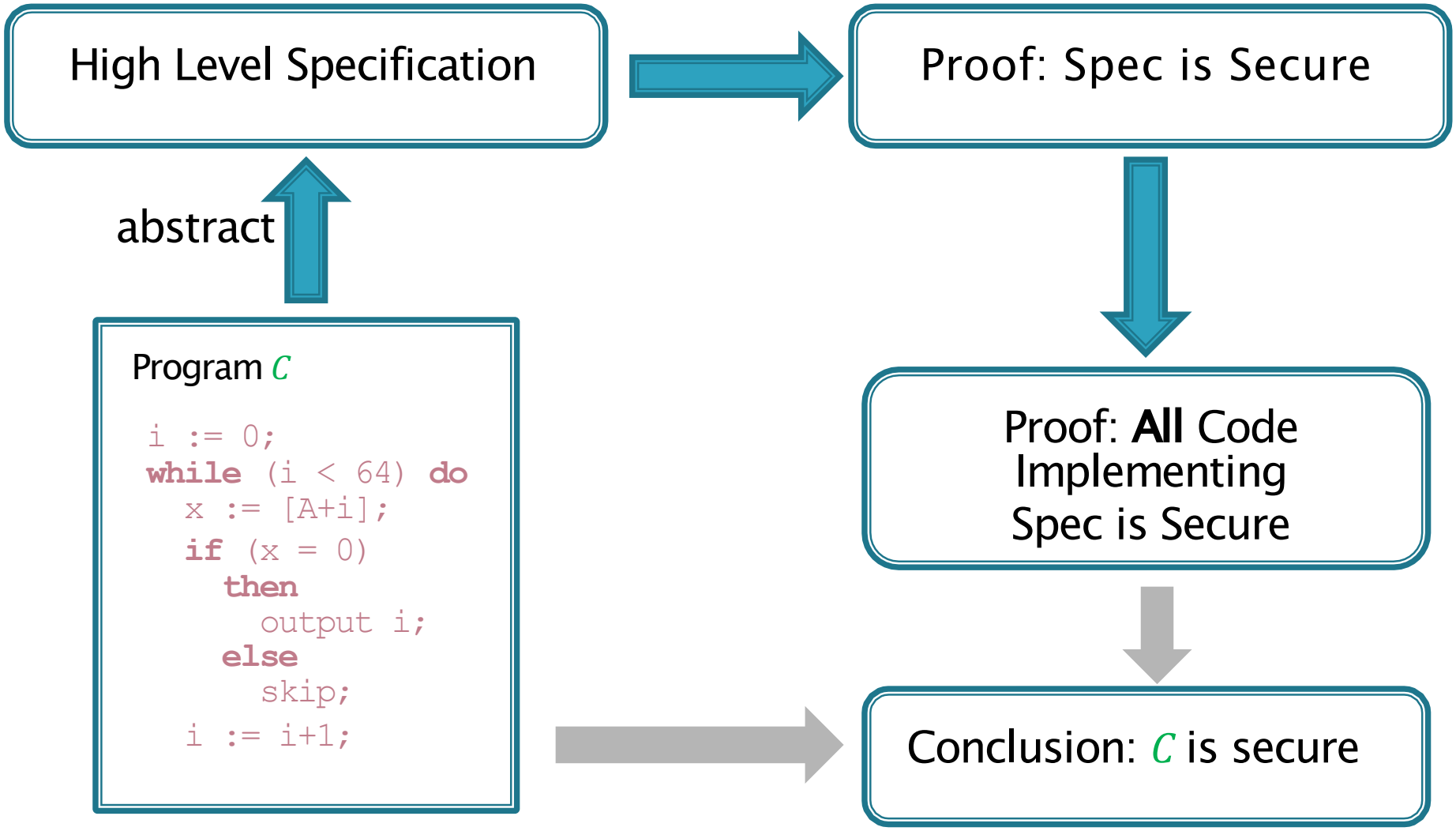
---

step (State rs m) (State rs' m')

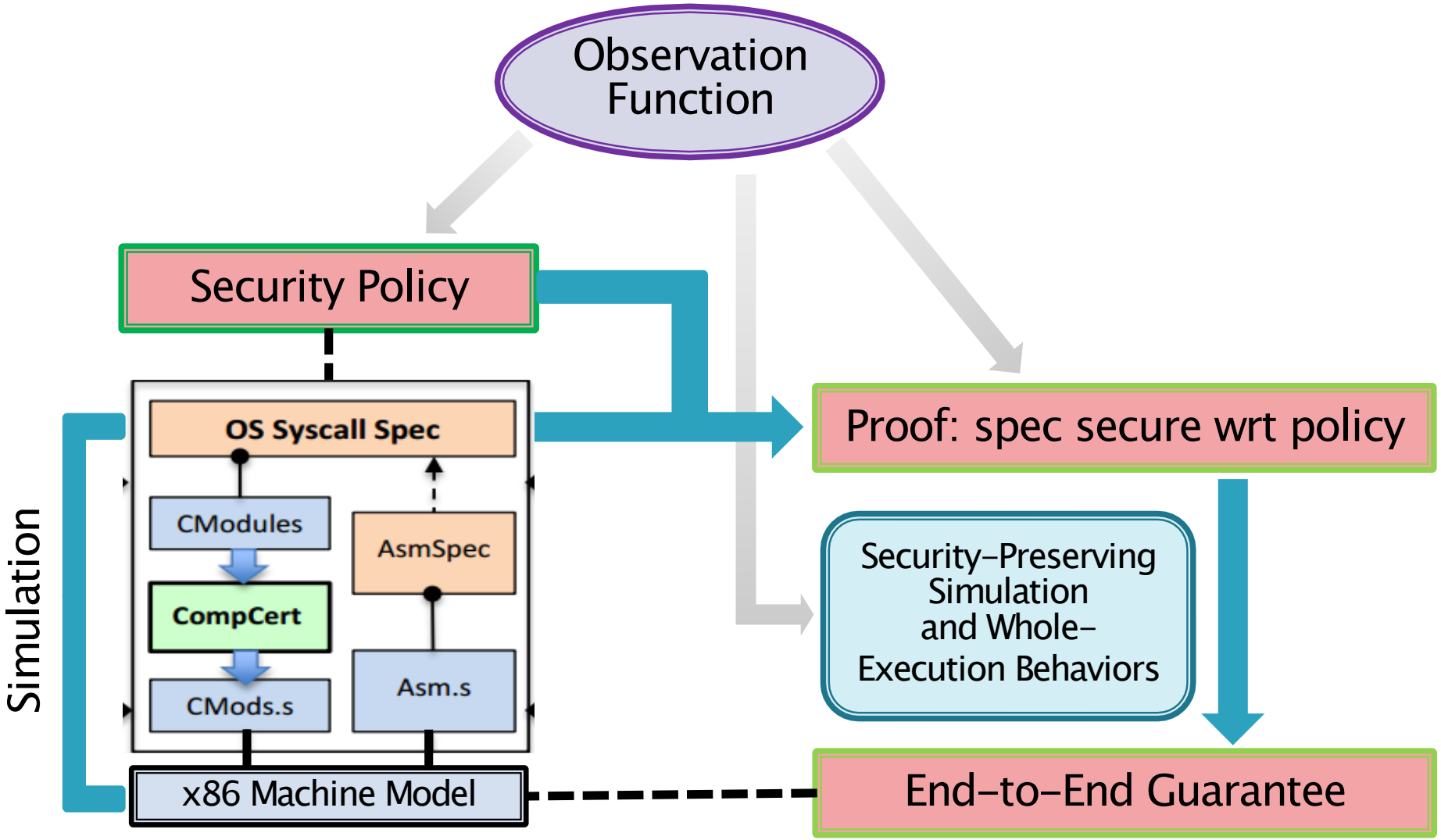
Rest of The Talk

1. Modeling preemption
2. Information flow security
3. Hard real-time

The CertiKOS Approach

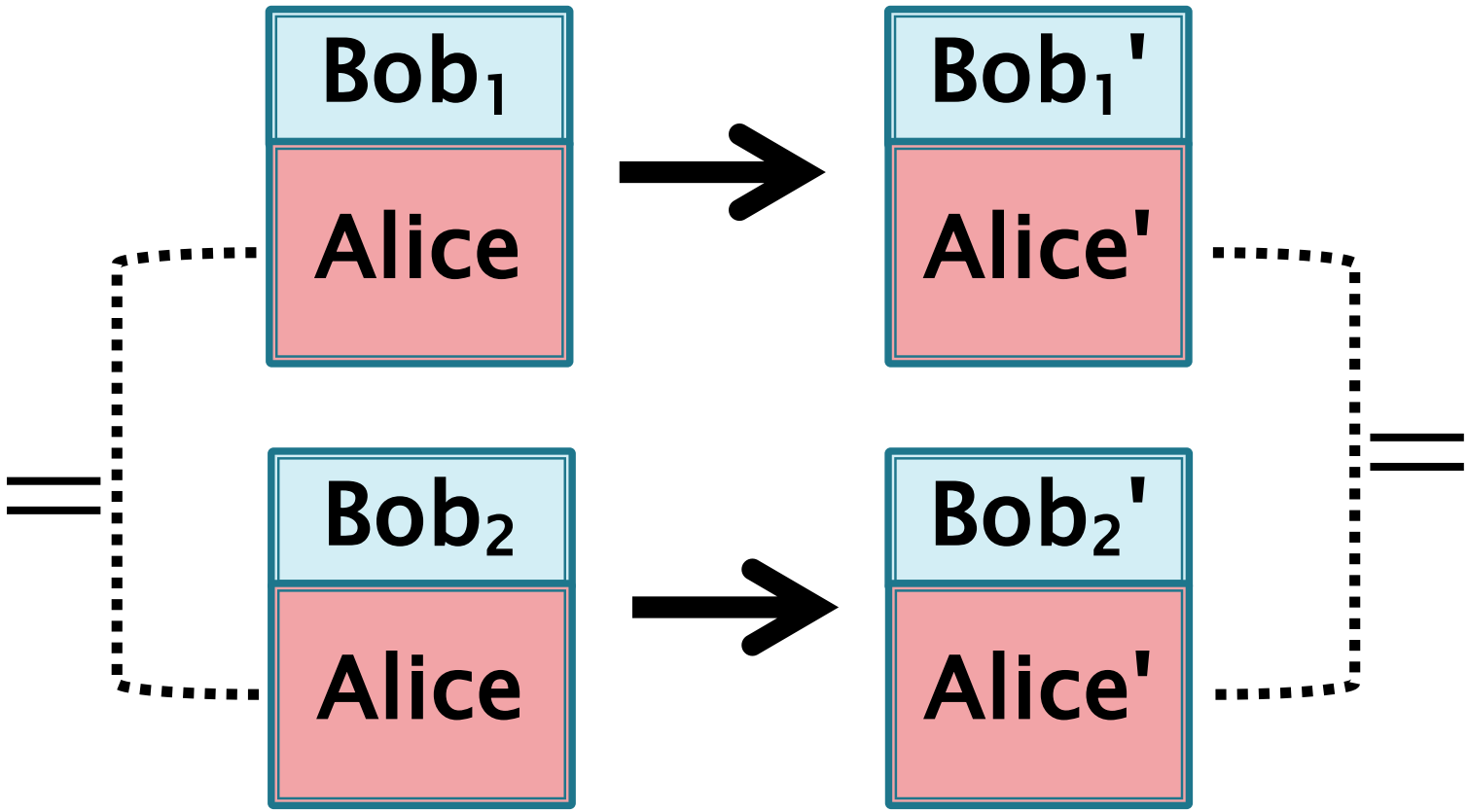


The CertiKOS Approach (cont'd)



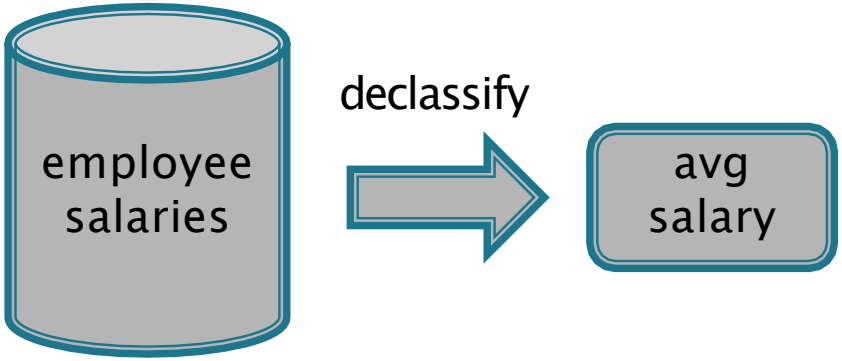
Pure Noninterference

"Alice's behavior is influenced only by her own data."



Common end-to-end security property for systems using security-label reasoning.

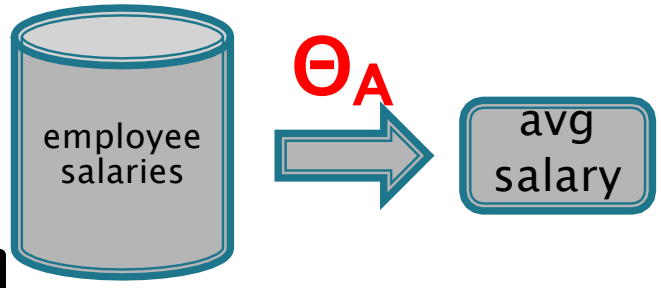
More Complex Policies




```
void printAvg() {
  int sum = 0;
  for int i = 0 to db.size-1
     sum += db[i];

  double avg = double(sum) / (db.size-1);
  print(avg);
}
```

Generalized Noninterference



avg(σ) = (σ(0) + σ(1) + ... + σ(size−1)) / (size−1)

printAvgSpec(σ) = σ{out ➤ out(σ) ++ [avg(σ)]}

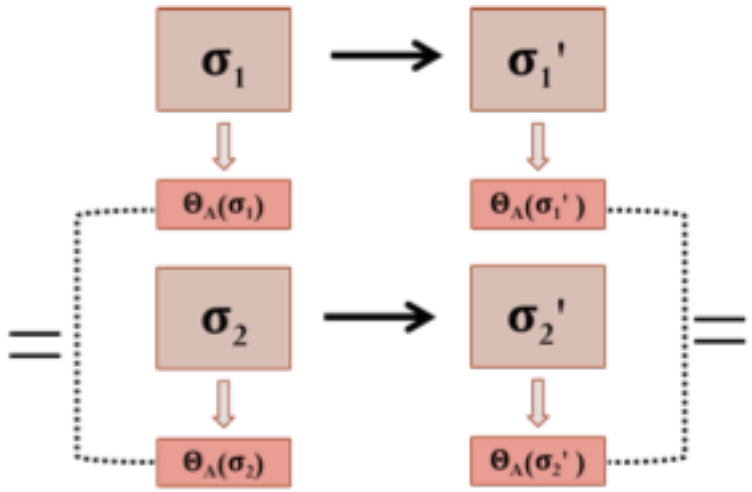
$\Theta_A(\sigma) = (avg(\sigma), out(\sigma))$

abstract

```
void printAvg() {
  int sum = 0;
  for int i = 0 to db.size-1
     sum += db[i];

  double avg = double(sum) / (db.size-1);
  print(avg);
}
```

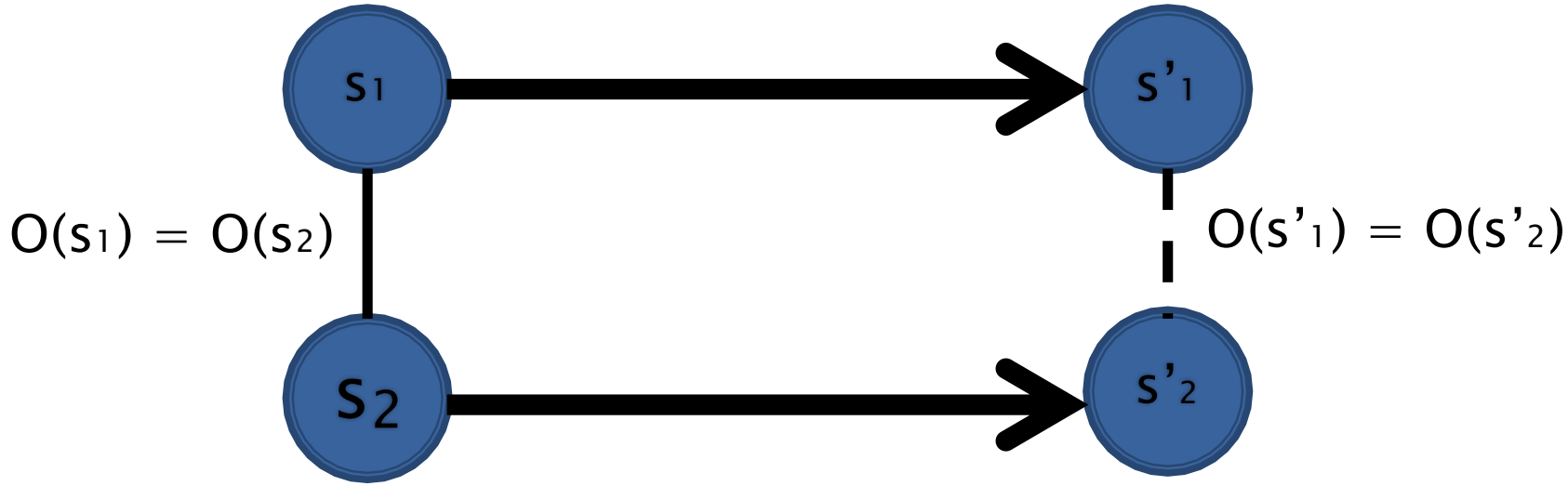
Proof: Generalized Noninterference



CertiKOS Information Flow Security

Generalized Non-interference [Costanzo et al PLDI'16]

- The execution of a process should not depend on other processes. It only depends on the *observables* of its own



CertiKOS Information Flow Security

What is observable

- Register set when the observer is active
- The observer's memory quota
- The observer's virtual memory space
- ……

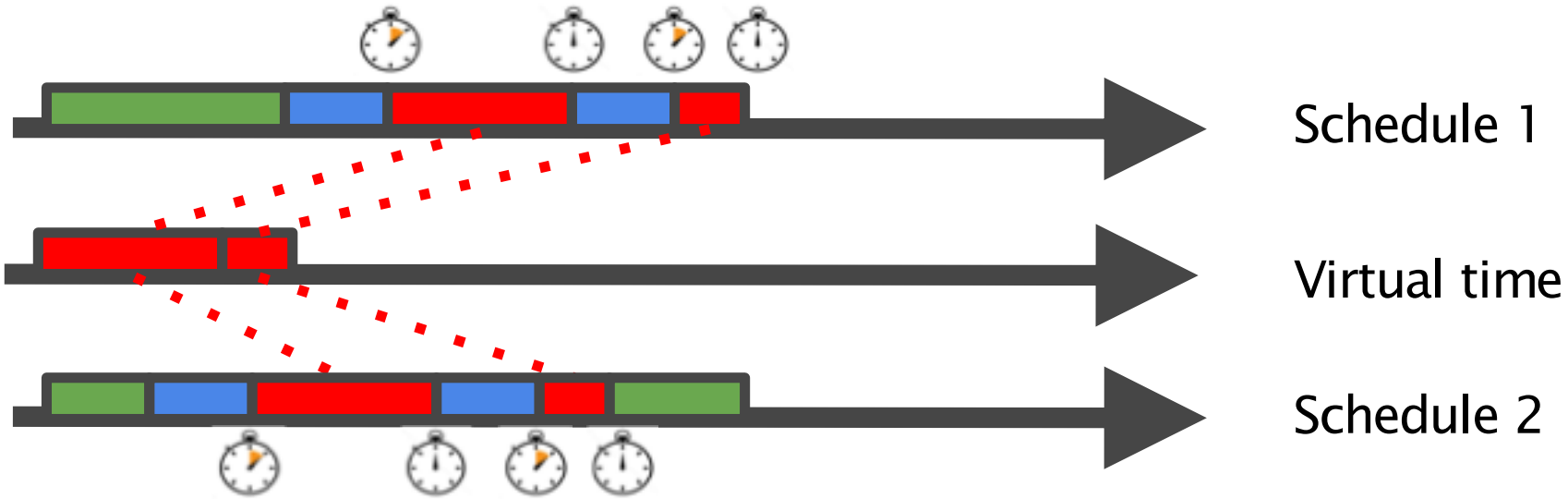
Get_Time

What if a process could call get_time?
Returning physical time (TSC value) leaks info

- Round-Robin scheduling: may be used to infer the total number of processes
- Priority scheduling: may be utilized to infer time consumption of higher priority processes

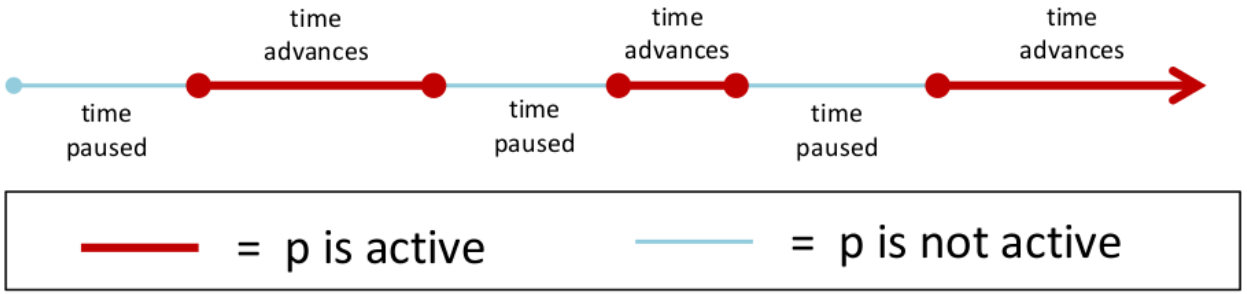
Get_Time

How about returning virtual time?
Virtual time: the observer's own execution time, similar to CLOCK_THREAD_CPUTIME_ID in Linux



Get_Time

Stopwatch implementation




```
int gettime() {
  int p = get_cid();
  int t = rd_tsc();
  return (sum_p + (t - cur));
}
```

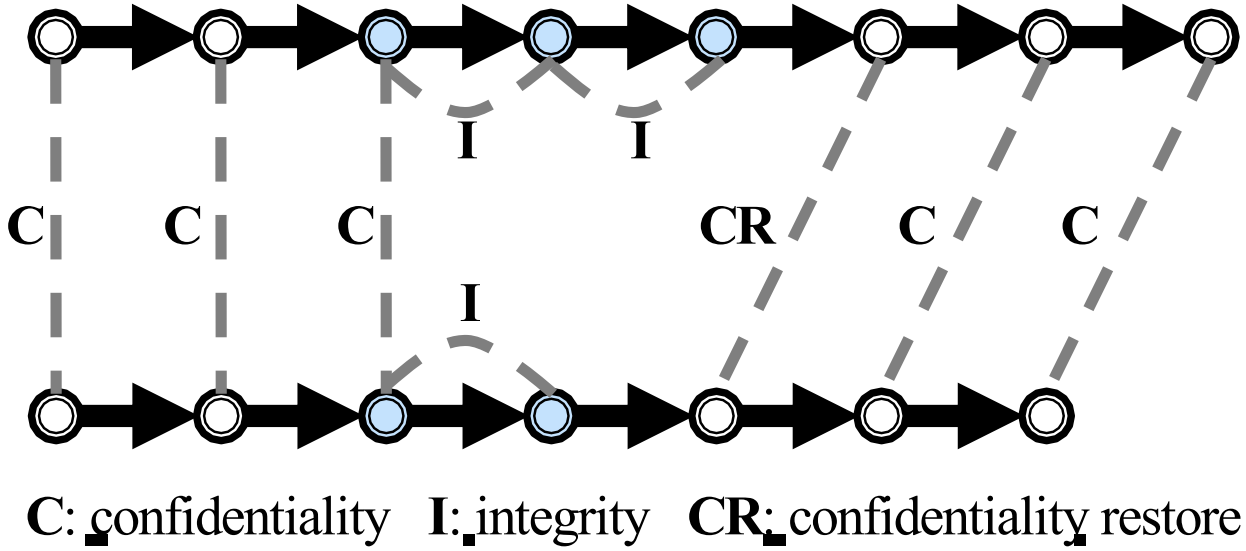
```
void stoptime() {
  int p = get_cid();
  int t = rd_tsc();
  sum_p += t - cur;
}
```

```
void starttime() {
  cur = rd_tsc();
}
```

Non-Interference with Preemption

The challenge

- The previous work: processes yield at deterministic points

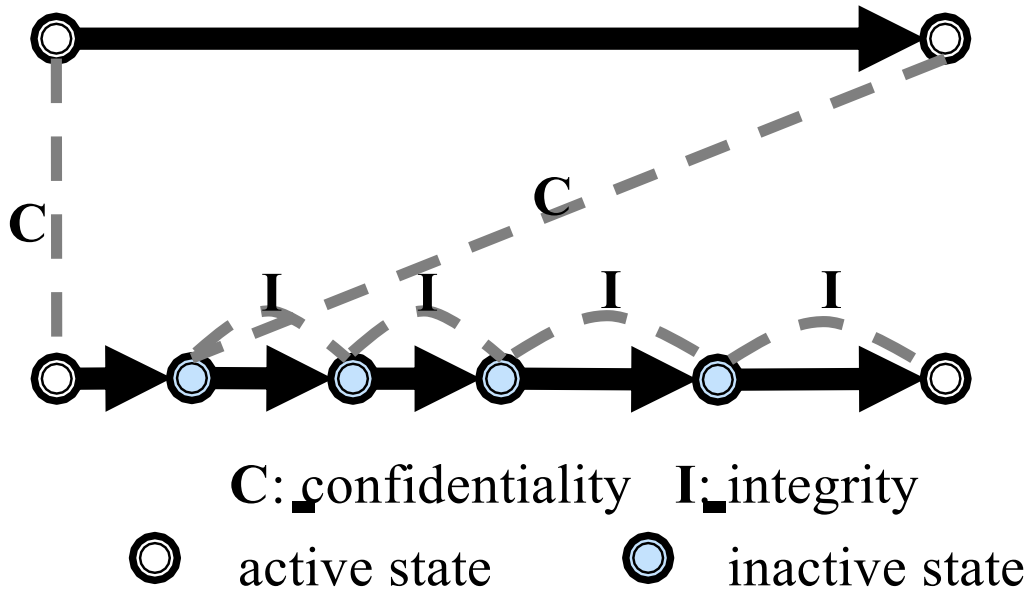


- Current work: timer interrupts occur non-deterministically

Non-Interference with Preemption

The new framework

- Integrity: when inactive, the observables do not change
- Confidentiality: when active, successors of two observably equivalent states are also observably equivalent
- Liveness: inactive states eventually lead to active ones



Rest of The Talk

1. Modeling preemption
2. Information flow security
3. Hard real-time

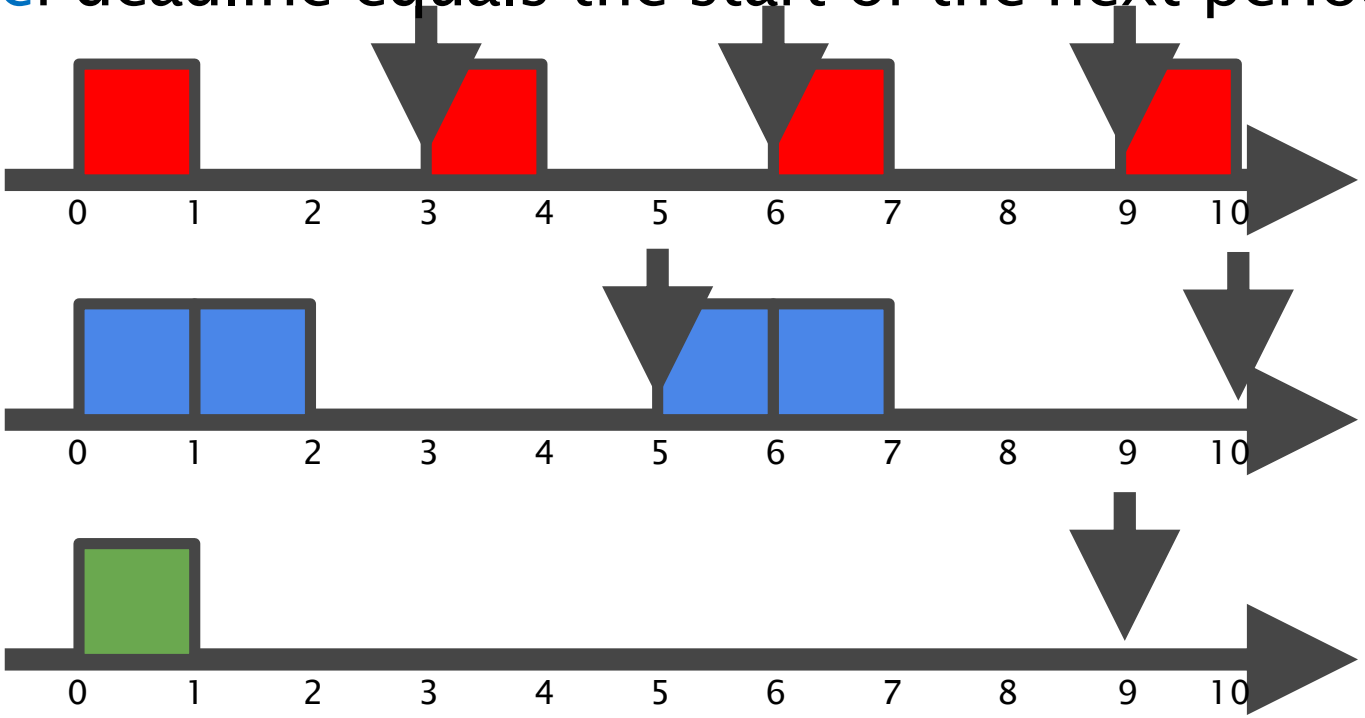
Hard Real-Time

Why hard real-time

- It is part of the correctness property of real-time systems: execution finishes before the deadline
- It guarantees liveness of the system

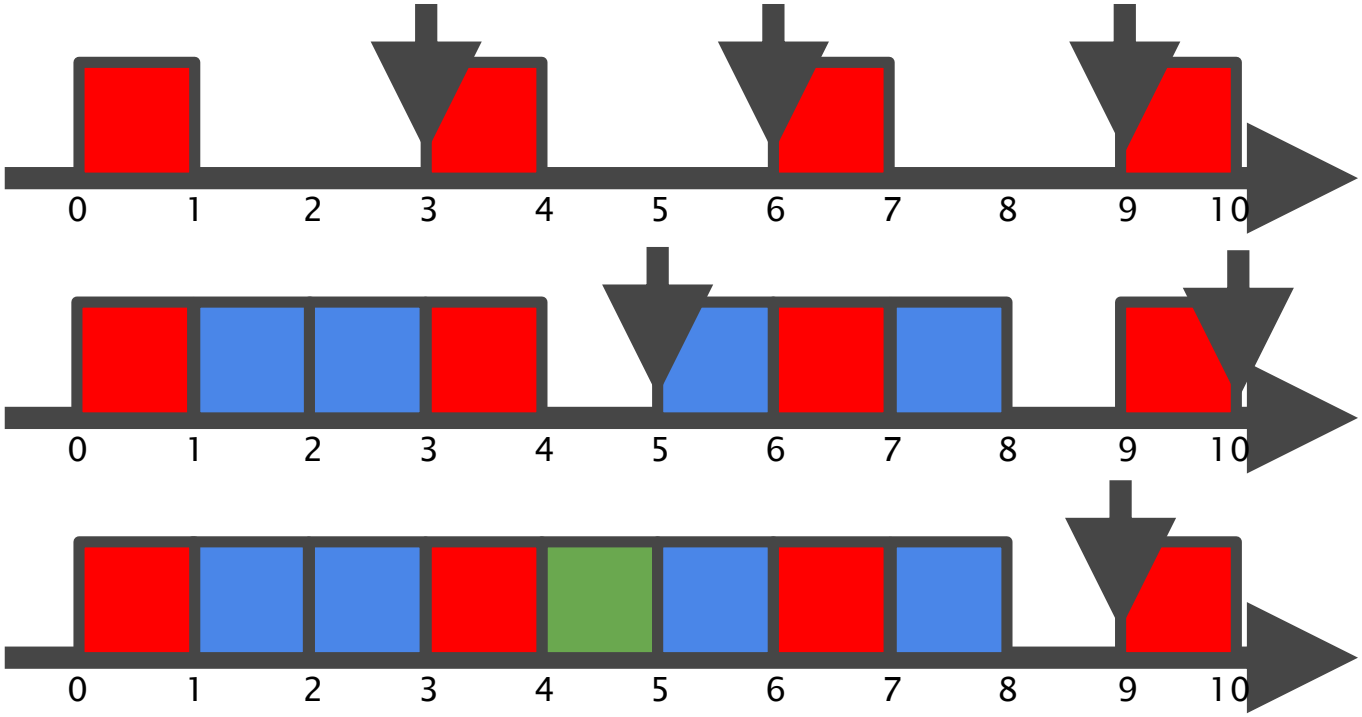
Terminologies

- Period: tasks are triggered at a regular period
- WCET: *worst case execution time* of a task within each period. *The kernel enforces WCET by cutting off overruns.*
- Deadline: deadline equals the start of the next period



Implementation

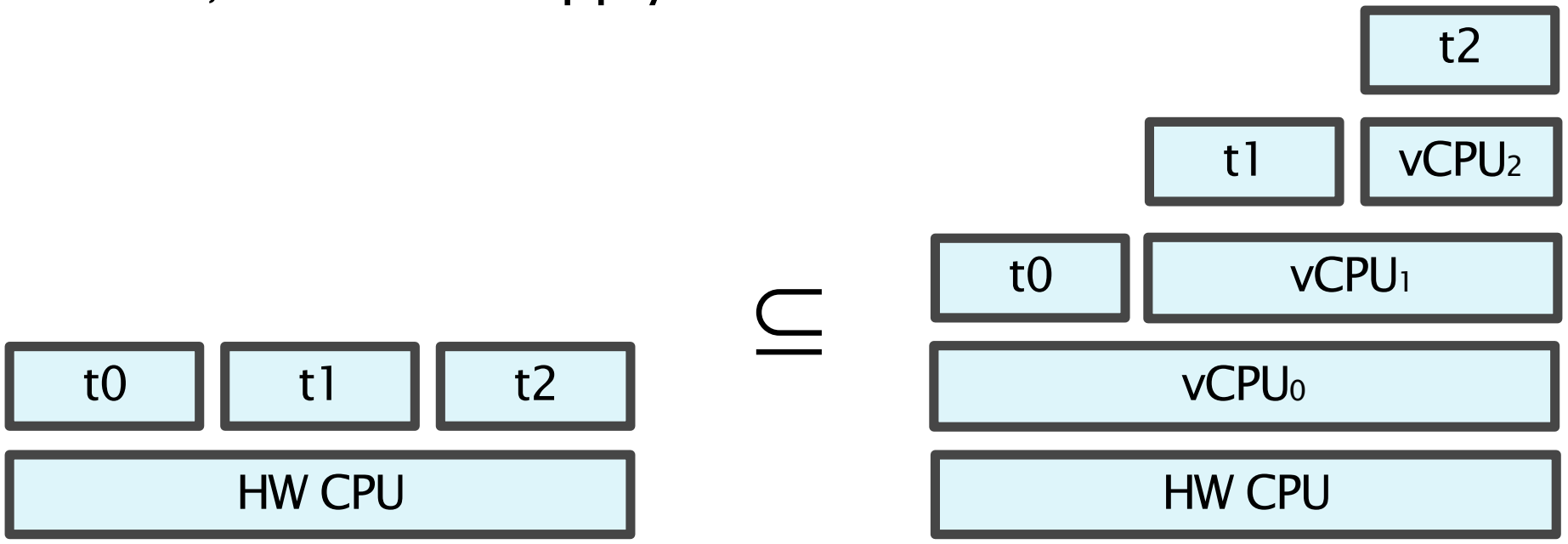
Priority scheduling:
The challenge is to reason about interleaving between tasks



Virtual Timeline

A way to construct compositional reasoning

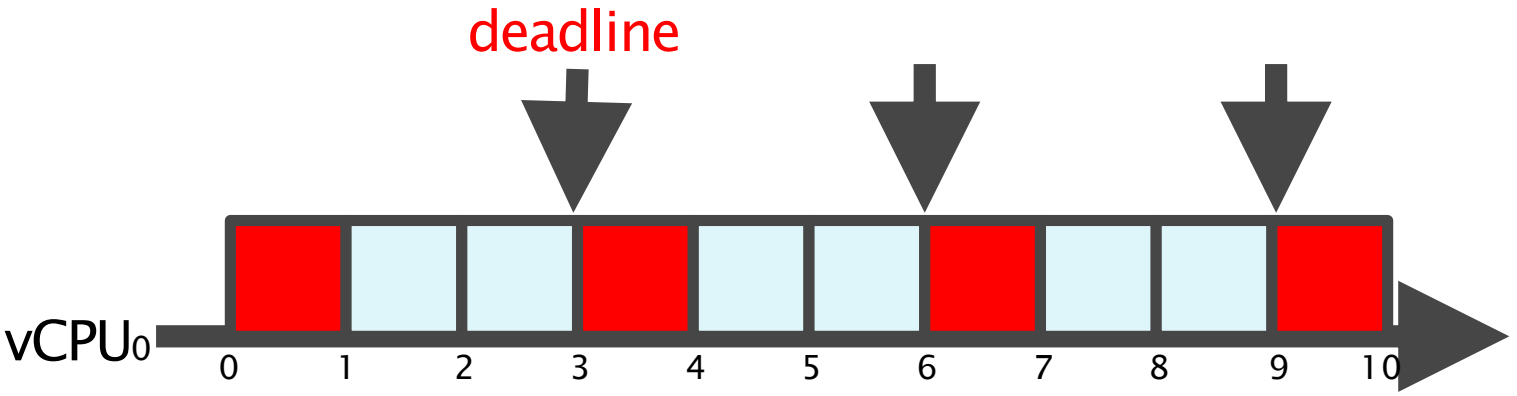
- Virtual time: time available for a specific priority level
- Virtual timeline: mapping from physical time to virtual time, similar to supply bound function



Virtual Timeline

deadline = 3,
WCET = 1

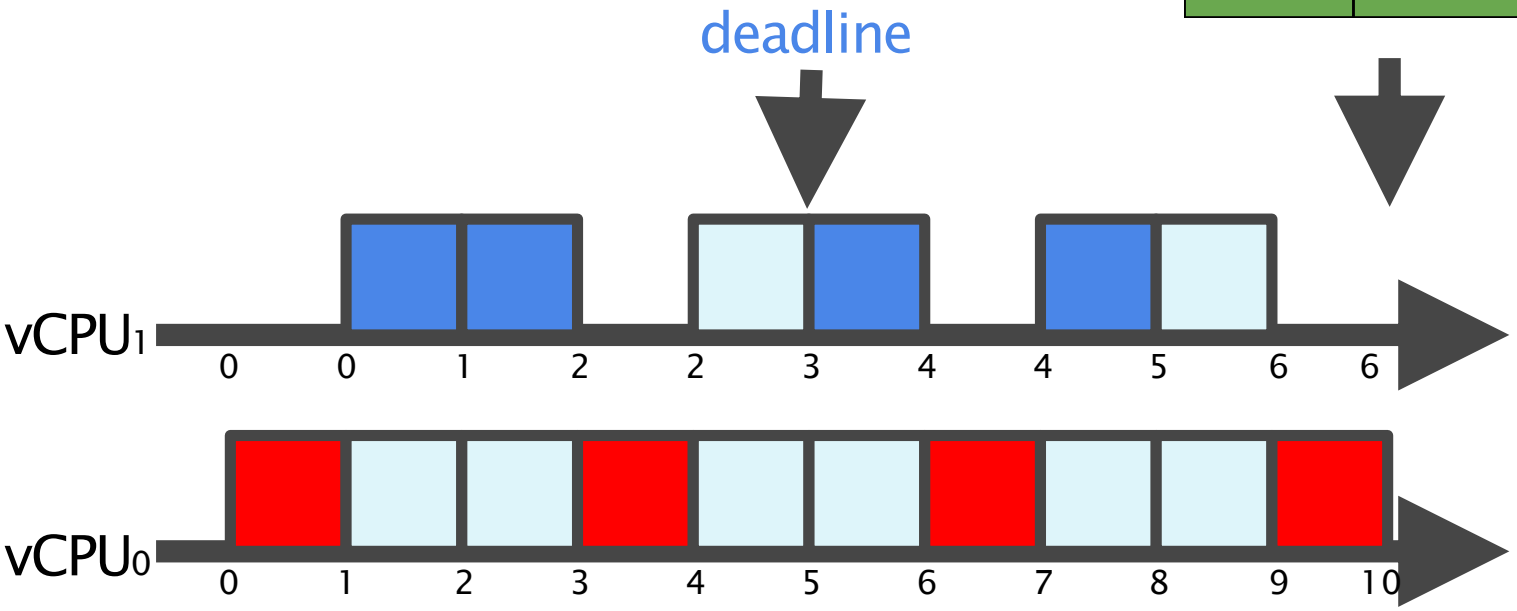
priority	period	WCET
0	3	1
1	5	2
2	9	1



Virtual Timeline

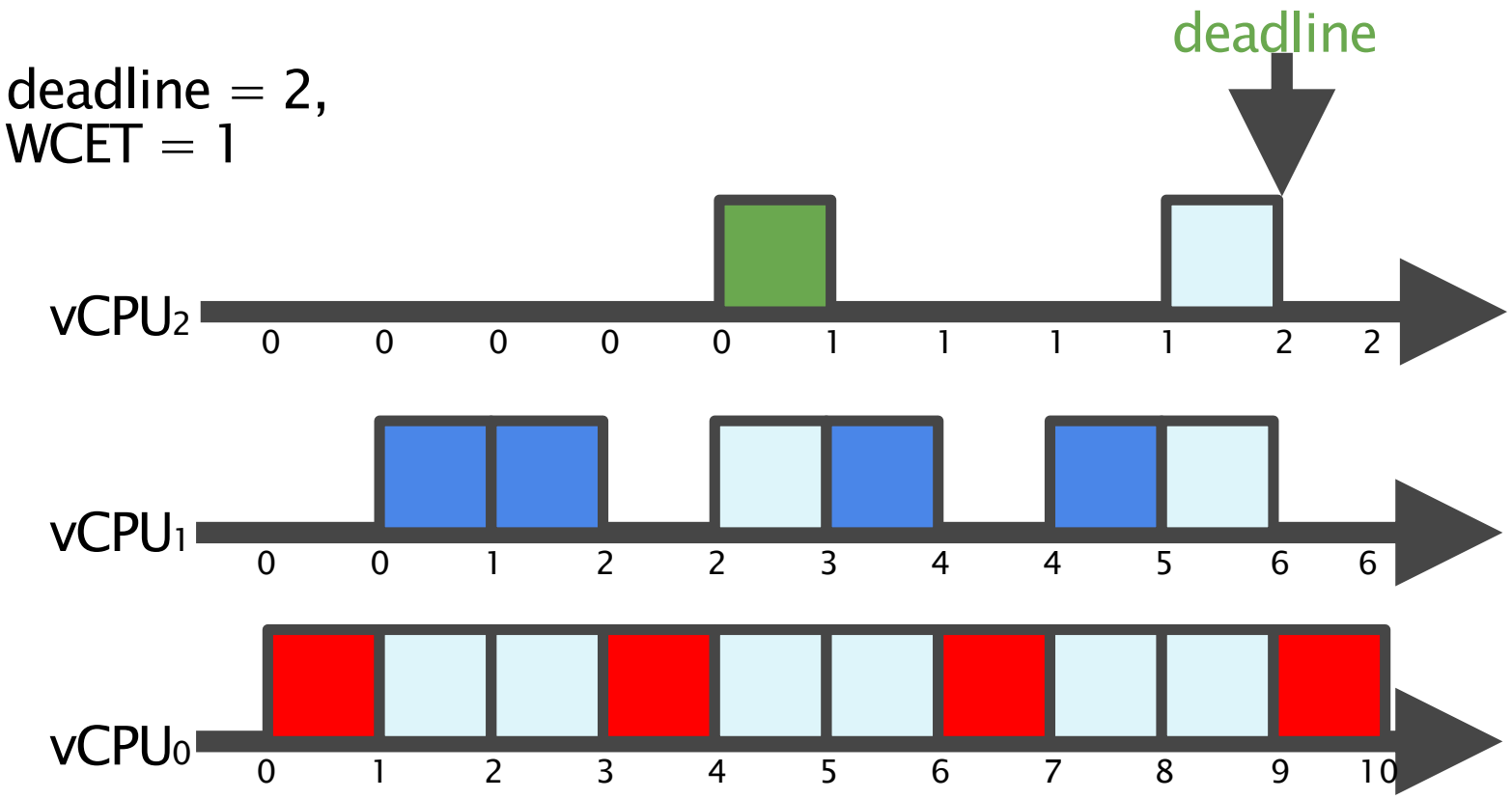
deadline = 3,
WCET = 2

priority	period	WCET
1	5	2
2	9	1

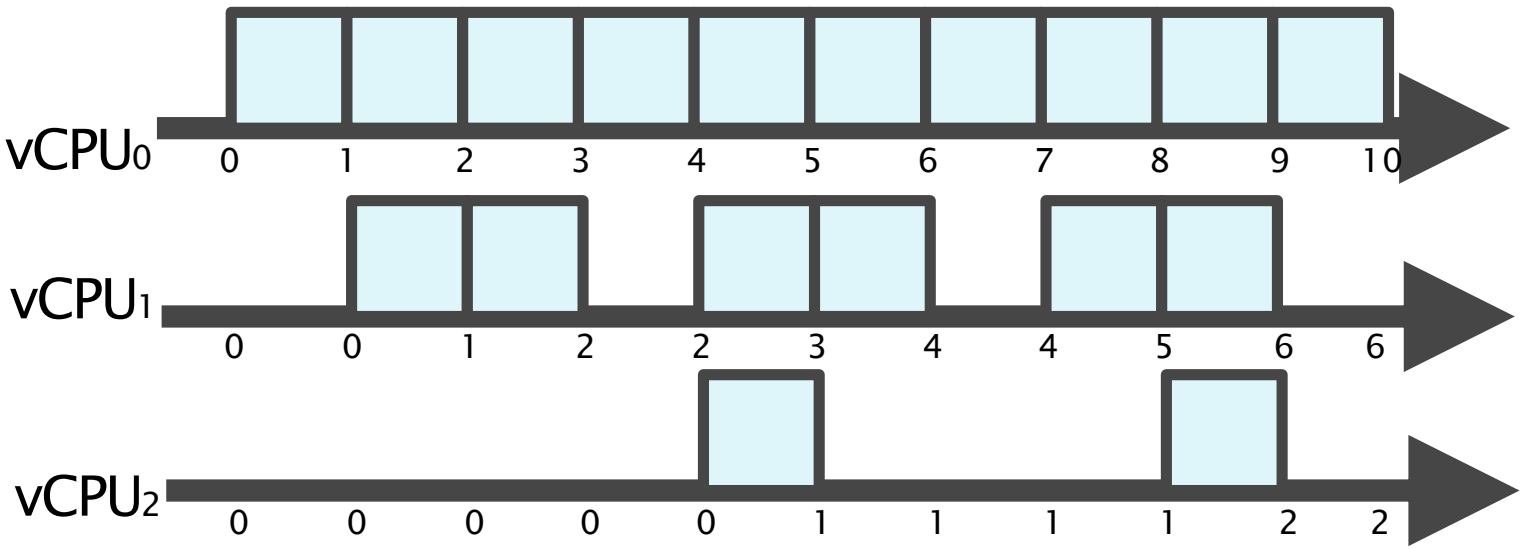


Virtual Timeline

priority	period	WCET
2	9	1



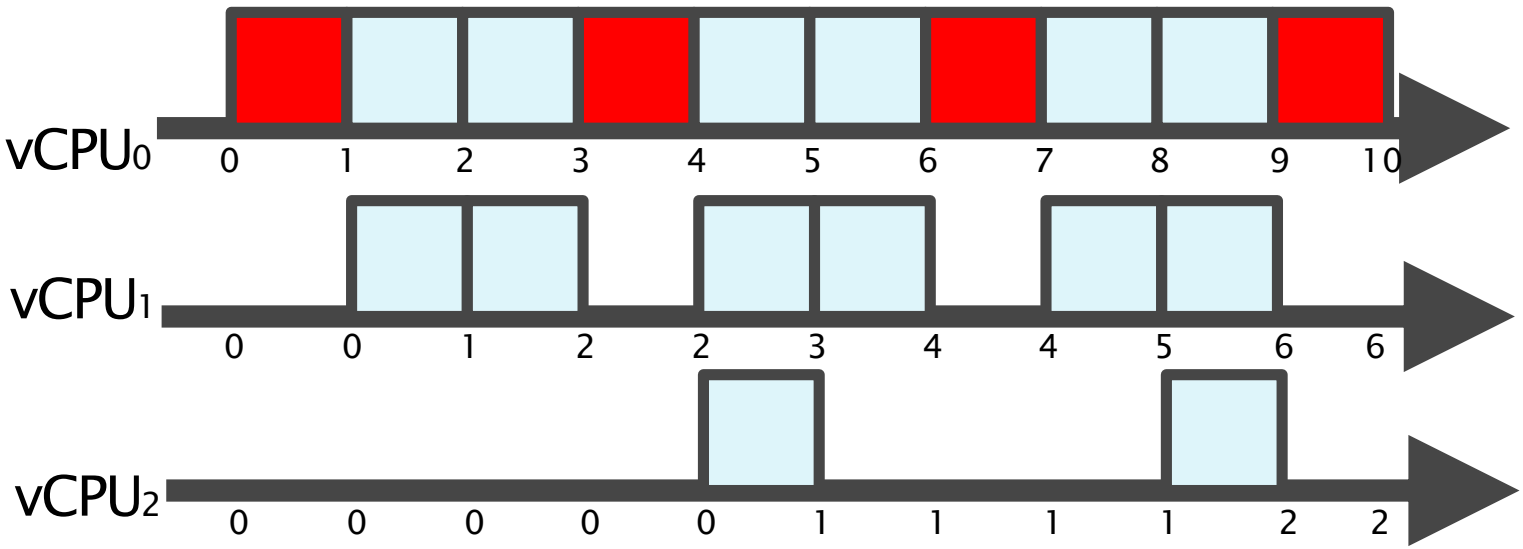
Computation of The Time Map



Time Map:

0	1	2	3	4	5	6	7	8	9	10

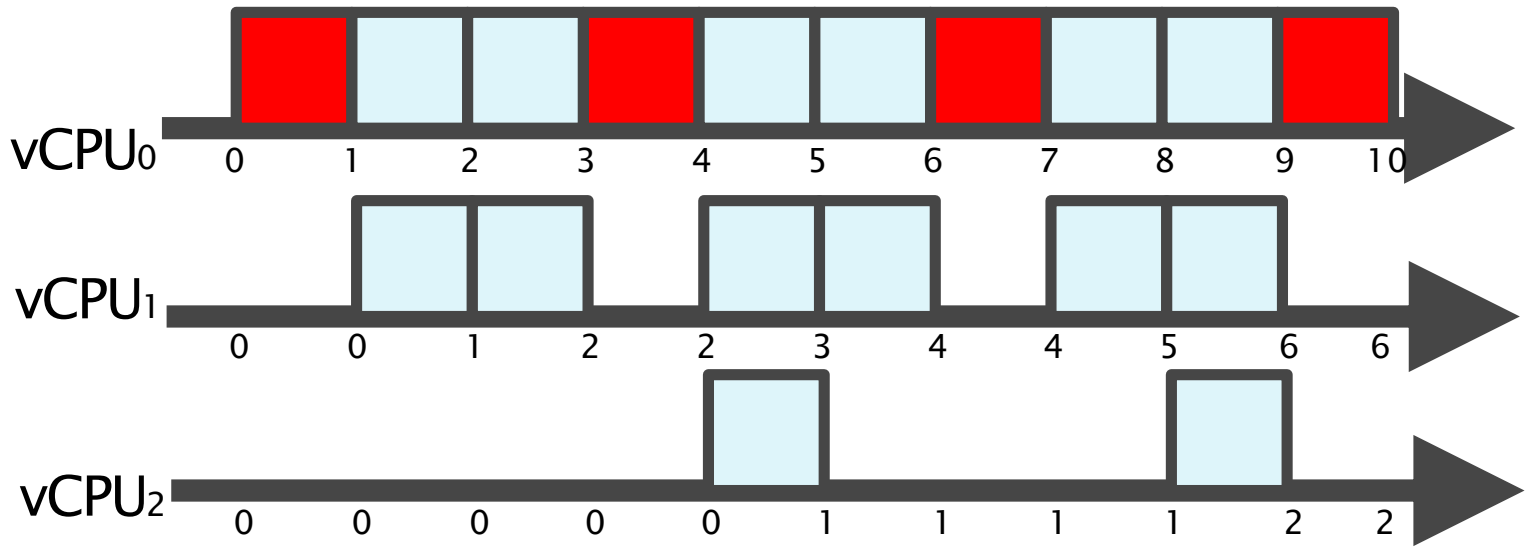
Computation of The Time Map



Time Map:

0	1	2	3	4	5	6	7	8	9	10

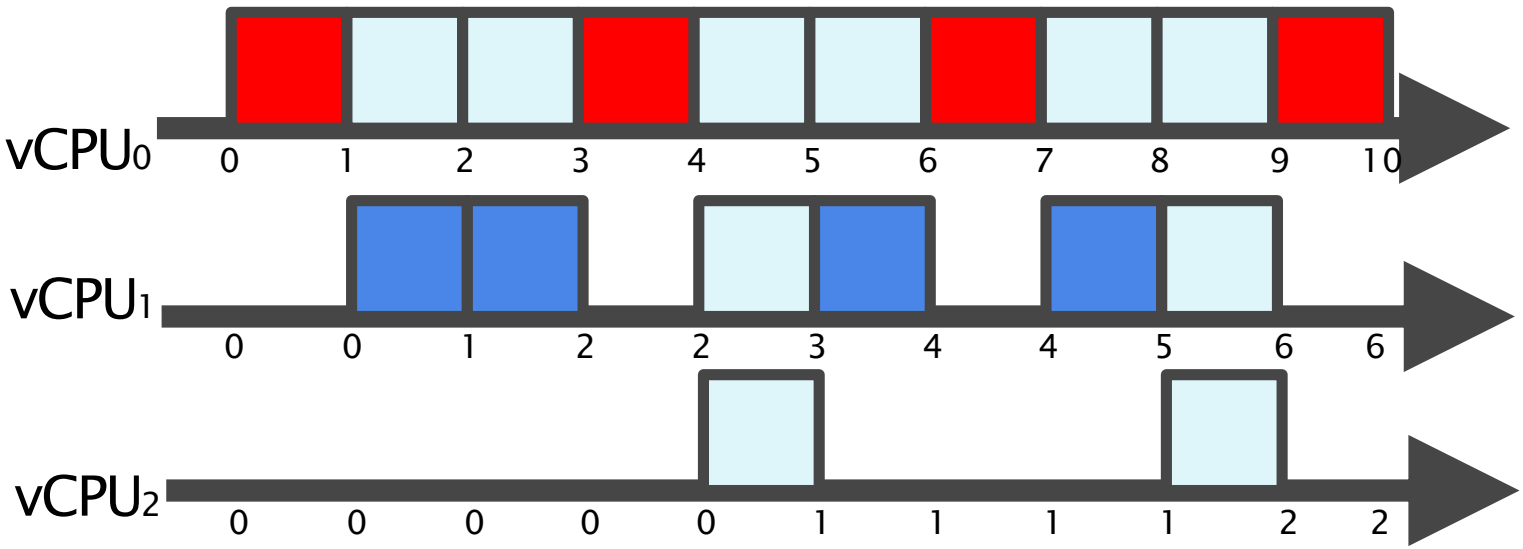
Computation of The Time Map



Time Map:

0	1	2	3	4	5	6	7	8	9	10
0	0	1	2	2	3	4	4	5	6	6

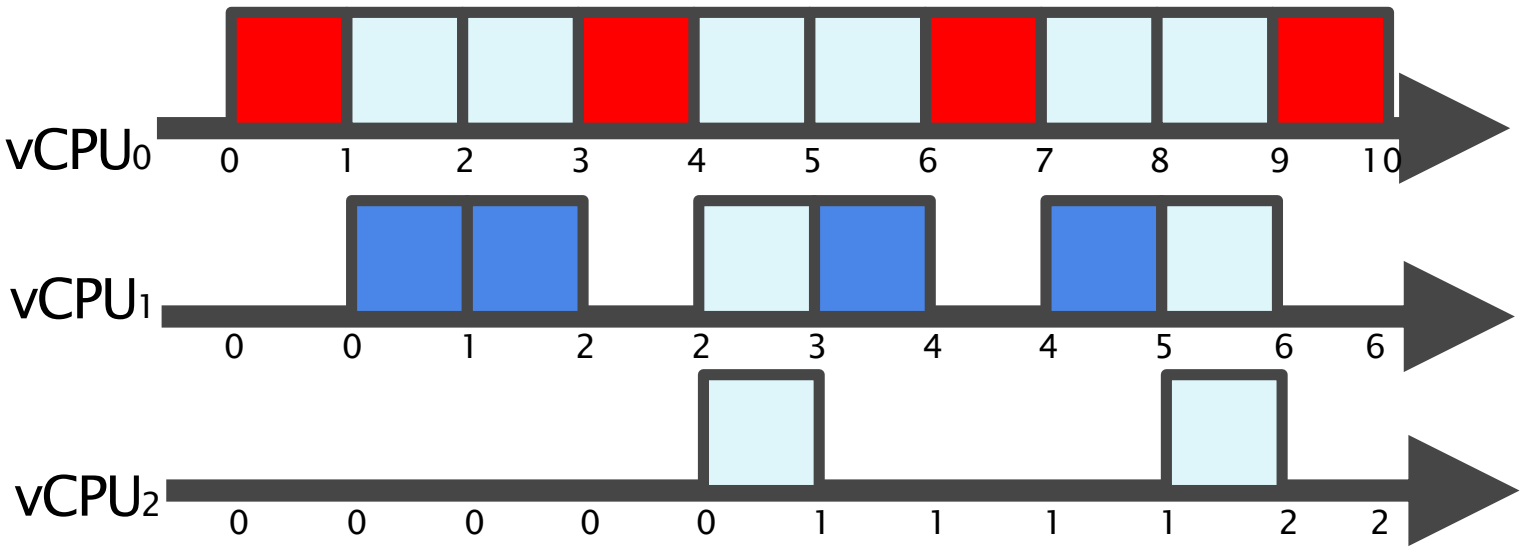
Computation of The Time Map



Time Map:

0	1	2	3	4	5	6	7	8	9	10
0	0	1	2	2	3	4	4	5	6	6

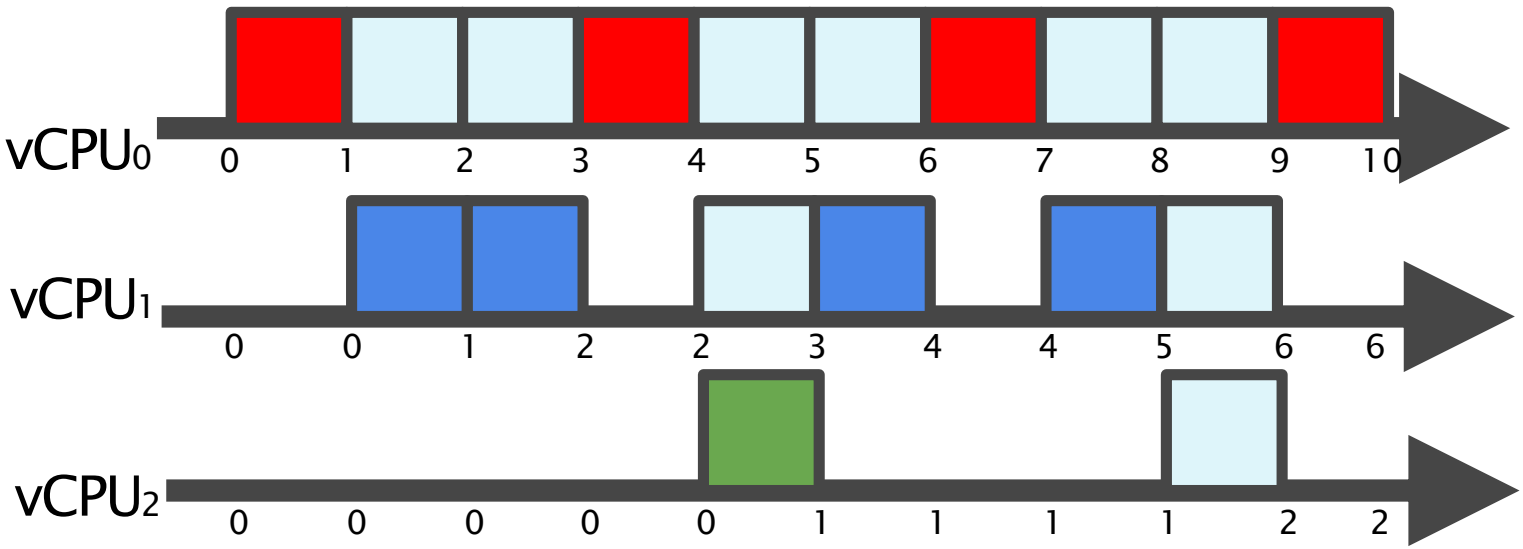
Computation of The Time Map



Time Map:

0	1	2	3	4	5	6	7	8	9	10
0	0	1	2	2	3	4	4	5	6	6
0	0	0	0	0	1	1	1	1	2	2

Computation of The Time Map

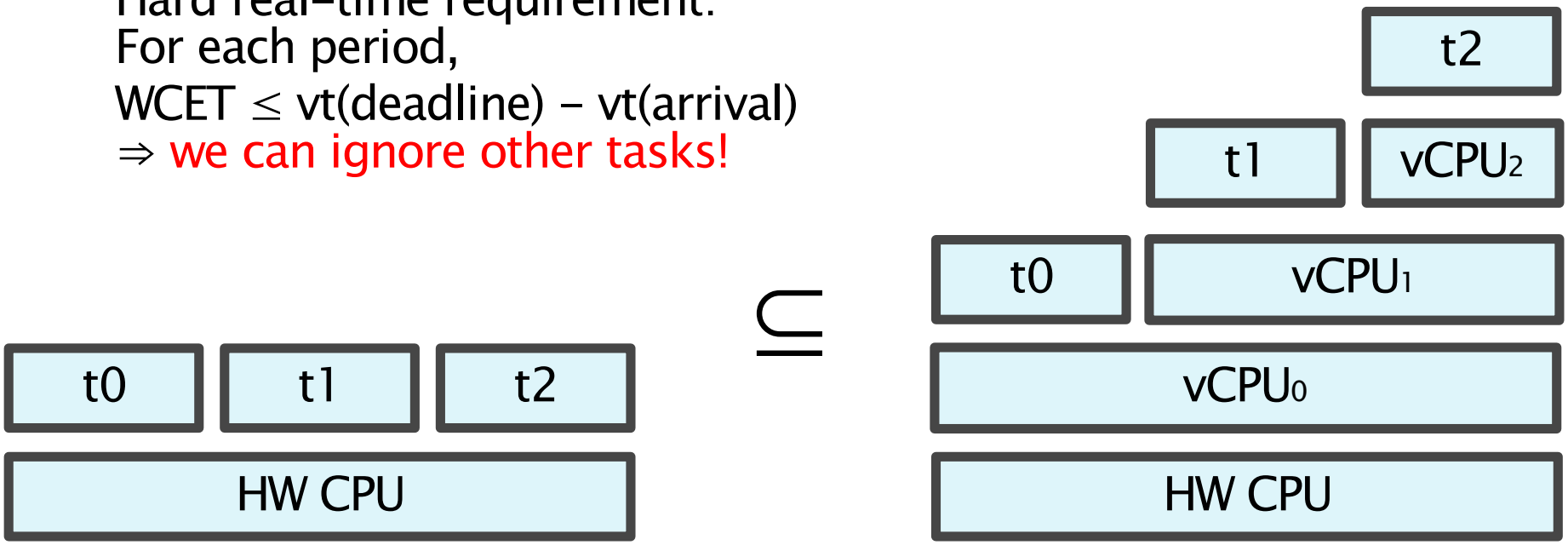


Time Map:

0	1	2	3	4	5	6	7	8	9	10
0	0	1	2	2	3	4	4	5	6	6
0	0	0	0	0	1	1	1	1	2	2

Virtual Timeline

Hard real-time requirement:
For each period,
WCET ≤ vt(deadline) – vt(arrival)
⇒ we can ignore other tasks!



Connecting Virtual Time with the Kernel Code

```
int concrete_sched(int t){
  tick_quanta();

  int pid =
    highest_pos_quantum();
  if (pid < N){
    quantum[pid]--;
  }
  return pid;
}
```

$\subseteq$

```
int highest_vtime_available(t){
  for(int i = 0; i < N; i++){
    if (vtime_available(i, t))
      return i;
  }
  return N;
}

int vt_sched(int t){
  return highest_vtime_available(t)
}
```

Connecting Virtual Time with the Kernel Code

```
int concrete_sched(int t){
  tick_quanta();

  int pid =
    highest_pos_quantum();
  if (pid < N){
    quantum[pid]--;
  }
  return pid;
}
```

Update quantums when new periods arrive

```
for(int i = 0; i < N; i++){
  if (t % T[i] == 0)
    quantum[i] = WCET[i];
}
```

Return the highest priority process with positive remaining quantum

```
for(int i = 0; i < N; i++){
  if (quantum[i] > 0)
    return i;
}
return N;
```

Connecting Virtual Time with the Kernel Code

Return the highest priority process
1> whose virtual time line could progress:
`vt(t + 1) = vt(t) + 1`
2> which has not exhausted all budget in the current period
`vt(t) - vt(t / T * T) < WCET`

```
int highest_vtime_available(t){
  for(int i = 0; i < N; i++){
    if (vtime_available(i, t))
      return i;
  }
  return N;
}

int vt_sched(int t){
  return highest_vtime_available(t)
}
```

Connecting Virtual Time with the Kernel Code

```
int concrete_sched(int t){
  tick_quanta();

  int pid =
    highest_pos_quantum();
  if (pid < N){
    quantum[pid]--;
  }
  return pid;
}
```

```
int highest_vtime_available(t){
  for(int i = 0; i < N; i++){
    if (vtime_available(i, t))
      return i;
  }
  return N;
}
```

⊆

```
int vt_sched(int t){
  return highest_vtime_available(t)
}
```

highest_pos_quantum() = highest_vtime_available(t)

$\forall t, p, quantum[p] = WCET_p - \min(WCET_p, vt_p(t+1) - vt_p(t / T_p * T_p))$

Connecting Virtual Time with the Kernel Code: INV on concrete_sched

INV: $\forall t, p$, quantum[p] = $WCET_p$ – min($WCET_p$, $vt_p(t+1) - vt_p(t / T_p * T_p)$)

```
pre: INV(t)
int concrete_sched(int t){
  tick_quanta();

  int pid =
    highest_pos_quantum();
  if (pid < N){
    quantum[pid]--;
  }
  return pid;
}
post: INV(t+1)
```

Lower priority i > k:
(1) Virtual time does not progress
(2) quantum[i] refilled only when (t+1)%T[i] = 0

Priority k, case analysis on two situations
(1) There is remaining budget (quantum[k] > 0), and virtual time progresses
(2) A new period arrives ((t+1)%T[k] = 0), and quantum[k] is refilled

Higher priority i < k:
(1) Must have already finished: quantum[i] = 0 /\ (t+1)%T[i] <> 0
(2) Virtual time keeps progressing

Assuming highest_pos_quantum() = k < N

Process-Local View

```
int vt_sched_p(int t){
  if (vtime_available(p, t)){
    return p;
  }else{
    return N;  // other tasks
  }
}
```

```
int highest_vtime_available(t){
  for(int i = 0; i < N; i++){
    if (vtime_available(i, t))
      return i;
  }
  return N;
}
```

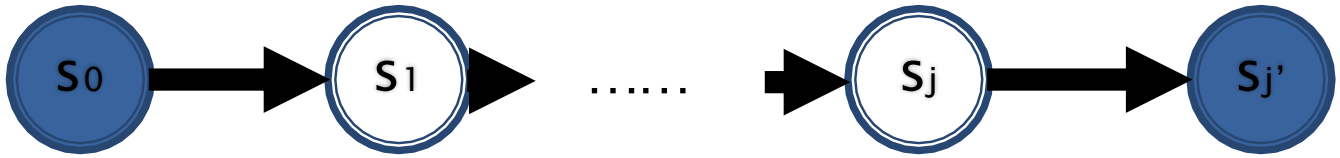
```
int vt_sched(int t){
  return highest_vtime_available(t)
}
```

vtime_available(p, t) = true→highest_vtime_available(t) = p

vtime_available(p, t) implies all higher priority processes have finished

Liveness

Liveness, an essential part of the non-interference proof, is then inferred by schedulability



tick = t
pid = k

tick = t'
pid = _

tick = t' + 1
pid = k

```
int vt_sched_p(int t){
  if (vtime_available(p, t)){
    return p;
  }else{
    return N;  // other tasks
  }
}
```

There exists t' > t, s.t. vtime_available(p, t') is true

Task p is schedulable

Proof Effort

- Changes against existing work

Feature	LoC	Files Touched
Interrupts/Preemption	10533	161
Real-Time Scheduler	26142	22
Schedulability Analysis	1398	4
get_time system call	4401	28
Non-Interference	28458	28
Total	**70932**	**243**

Conclusion: Real-Time CertiKOS

- New machine model supporting interrupts and timing registers
- New framework for non-interference of a preemptive OS kernel
- Formal description of real-time properties, and the *virtual timeline* framework for hard real-time reasoning
- Future work: shared memory IPC, porting more complex cost models, interference from other cores

Resource-Adaptive CertiKOS

- A general notion of resource objects
  - Model both raw HW objects or their SW abstraction
  - Each layer has its own set of resource objects
  - A generic resource-safety property (e.g., schedulability)
  - Local reasoning and frame rules apply everywhere
- Resource-aware abstraction layers
  - All programs preserve "resource-safety" properties
- Resource- & security-preserving simulation
  - CertiKOS layer refinement is now "resource-preserving contextual refinement"
  - CompCertX can be made resource-aware

Thank You!