Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads

Zhong Shao
Yale University

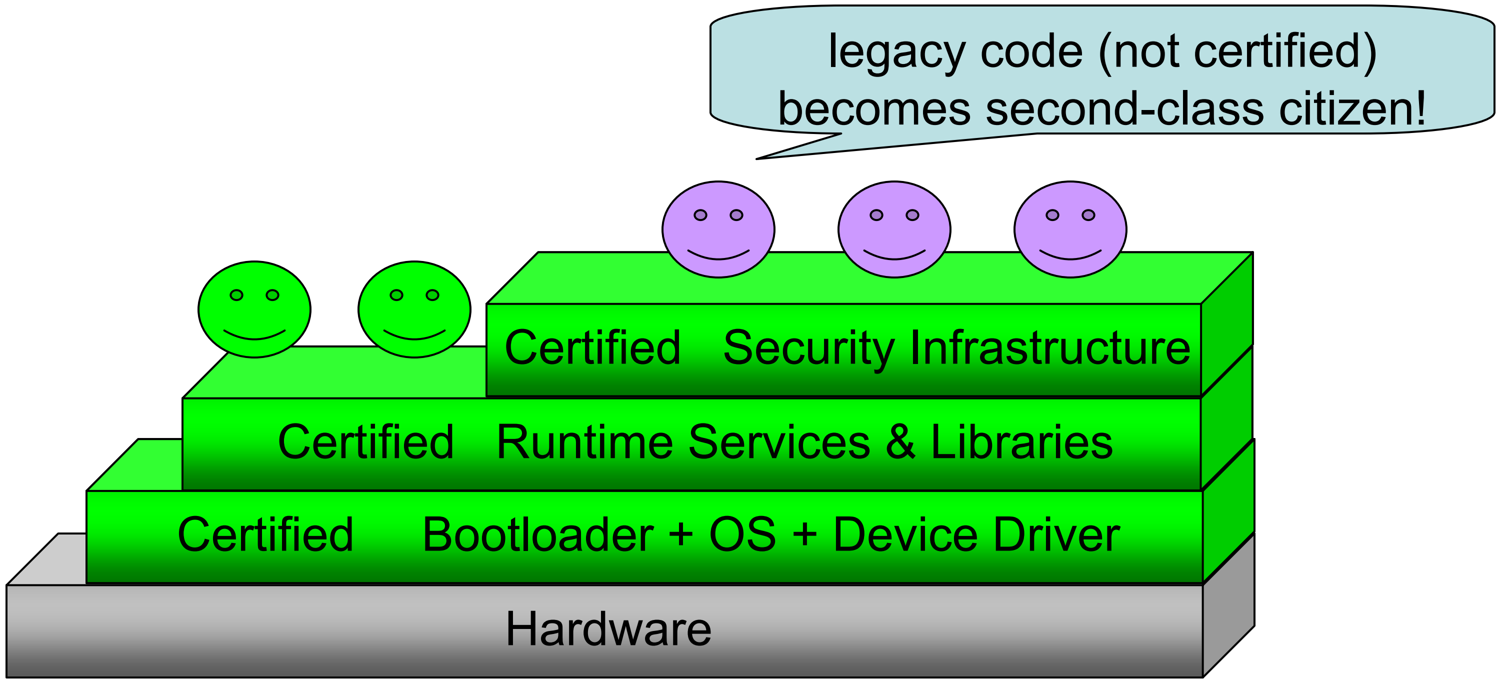
Joint work with Xinyu Feng (TTI Chicago) and Yu Guo (USTC) and Yuan Dong (Tsinghua)

HCSS challenges: my take

- How to measure software dependability?
  - How to make progress if you don’t know what is better?
  - ***Main confusion***: dependability of software vs. that of its execution environment & human operator
- System software is not dependable
  - How much can you do if the OS kernel is buggy?
- The “last-mile problem” for verification
  - High-level model vs. actual executable code

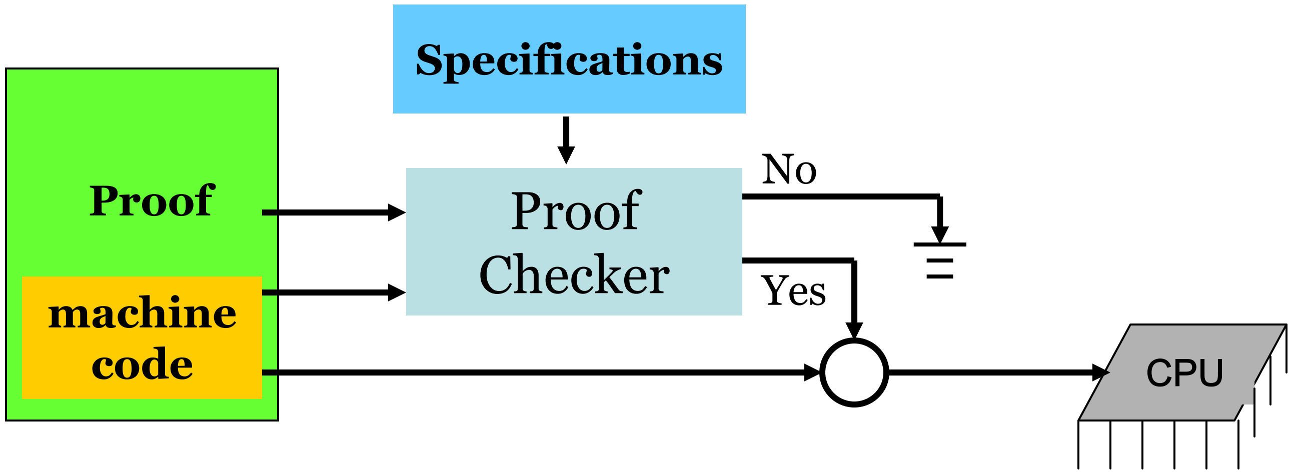
Certified Software

We need a "certified" computing platform!


legacy code (not certified) becomes second-class citizen!
Certified Security Infrastructure
Certified Runtime Services & Libraries
Certified Bootloader + OS + Device Driver
Hardware

We need firm control of the lowest-level software!

Components of a certified framework

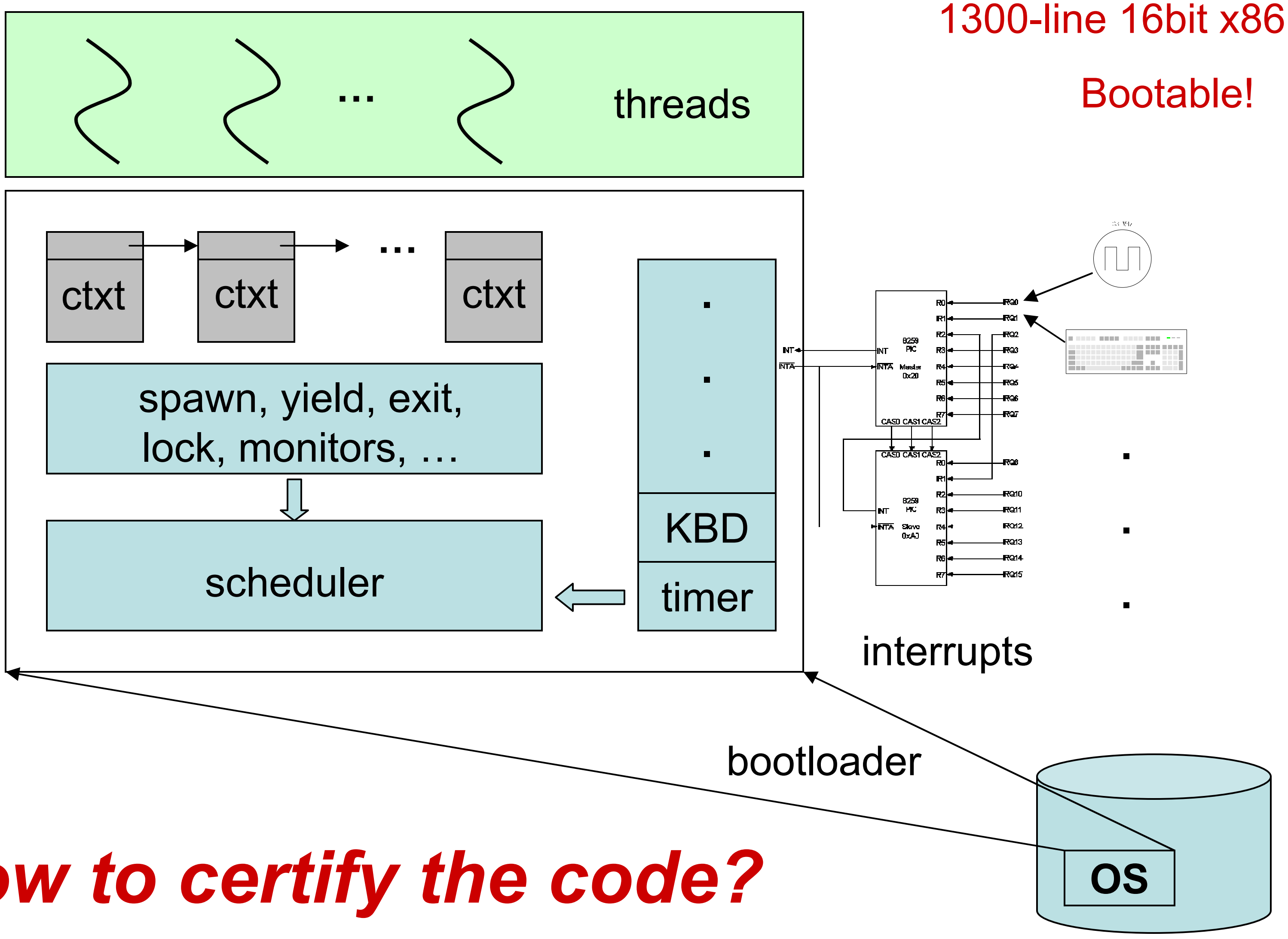


- certified code (proof + machine code)
- machine model
- specifications
- mechanized meta-logic
- proof checker

Case study: a Mini-OS

1300-line 16bit x86 code,

Bootable!



How to certify the code?

Certifying the Mini-OS

1300 lines of code

- bootloader
- scheduler
- timer int. handler
- thread lib: spawn, exit, yield, …
- sync. lib: locks and monitors
- keyboard driver
- keyboard int. handler
- …

Many challenges:

- **Code loading**
- **Low-level code: C/Assembly**
- **Concurrency**
- **Interrupts**
- **Device drivers / IO**
- **Certifying the whole system**
  - **Many different features**
  - **Different abstraction levels**

Important questions to answer

1. Can we do it?

2. Can we do it in a clean way?

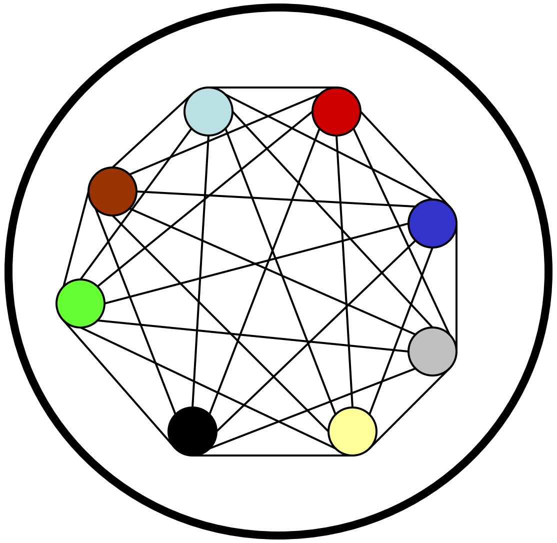
   (modularity, local reasoning, reusable, general, …)

3. Does it scale?

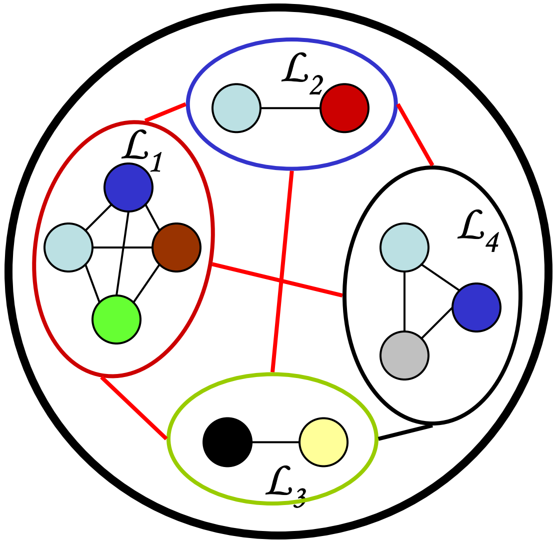
   (usability, support of automation, tool support)

Building fully certified systems

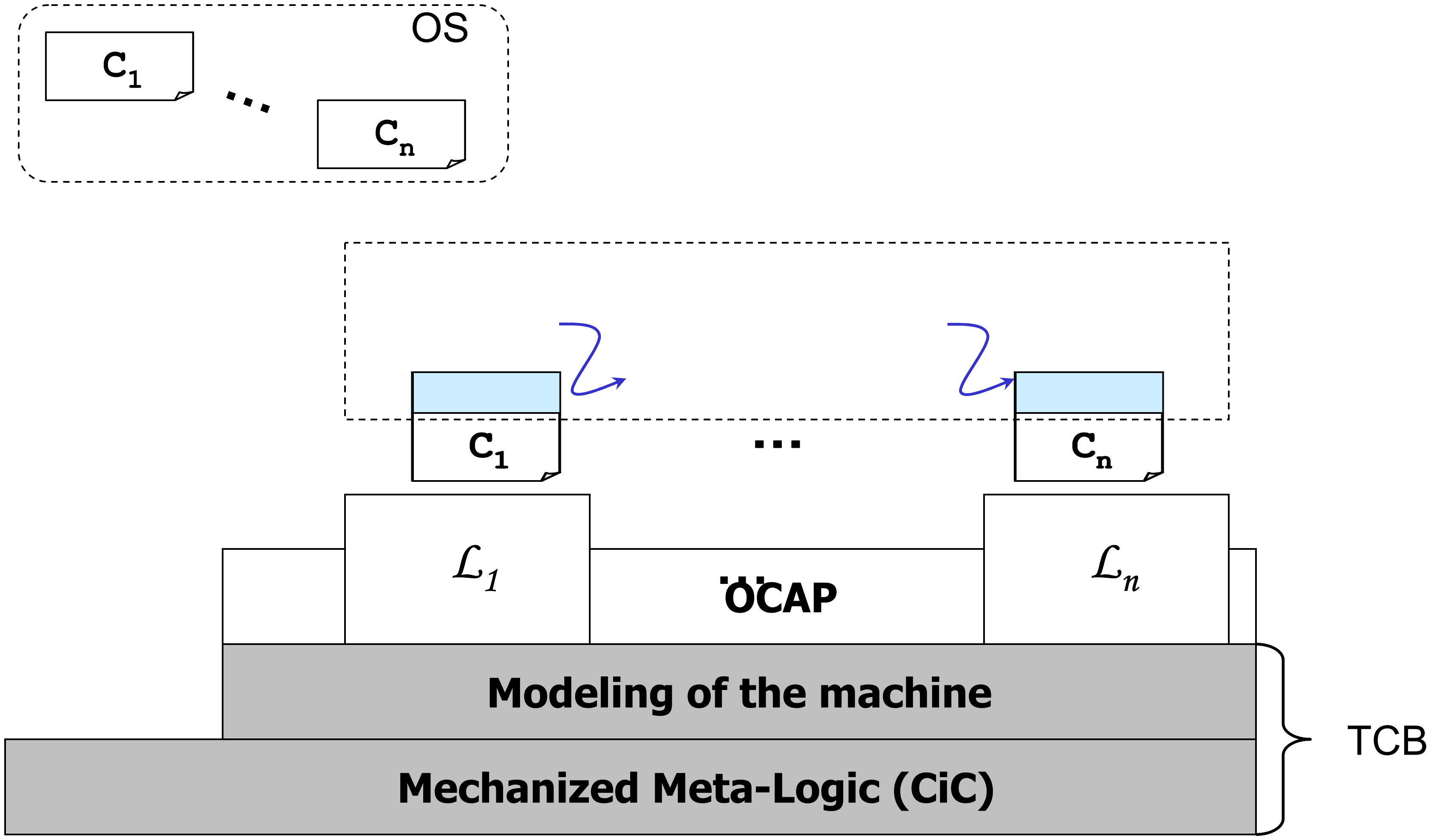
- One logic for all code
  - Consider all possible interactions.
  - Very difficult!



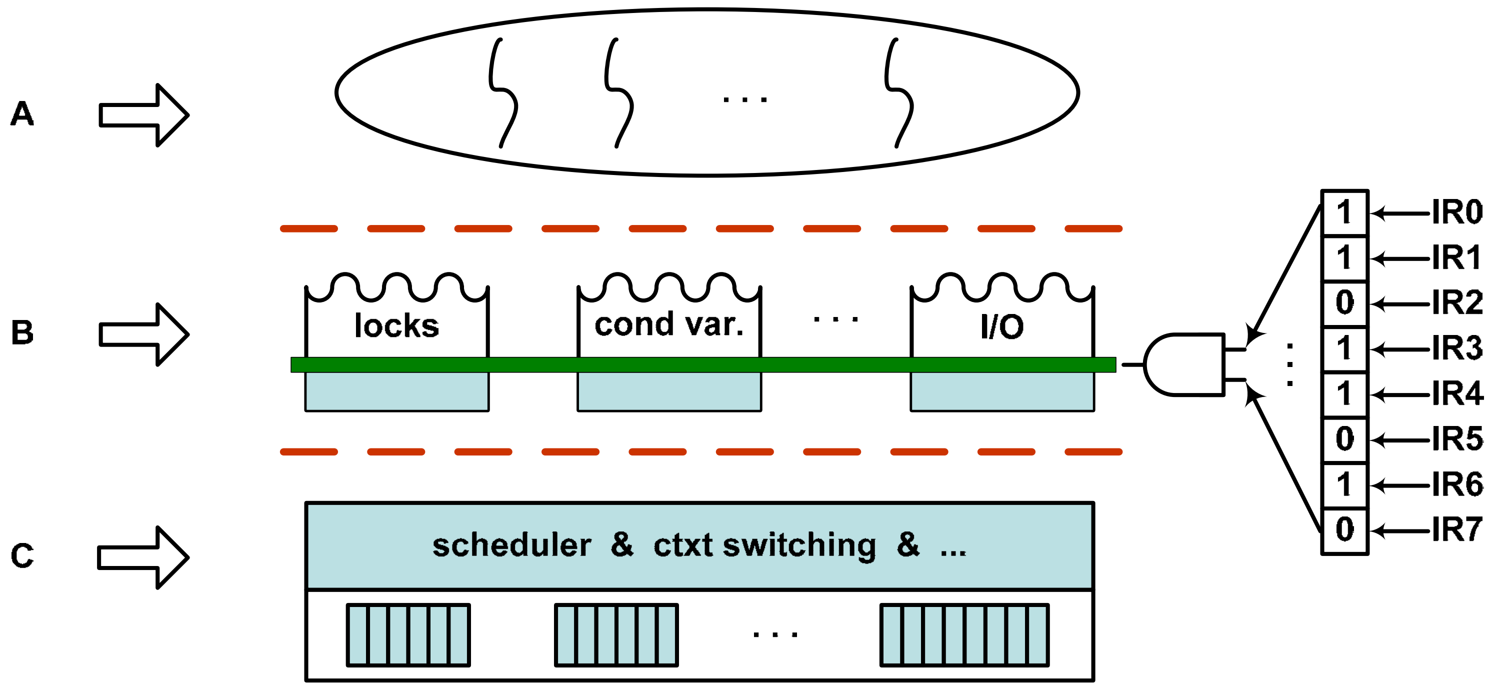
- Reality
  - Only limited combinations of features are used.
  - It's simpler to use a specialized logic for each combination.
  - Interoperability between logics



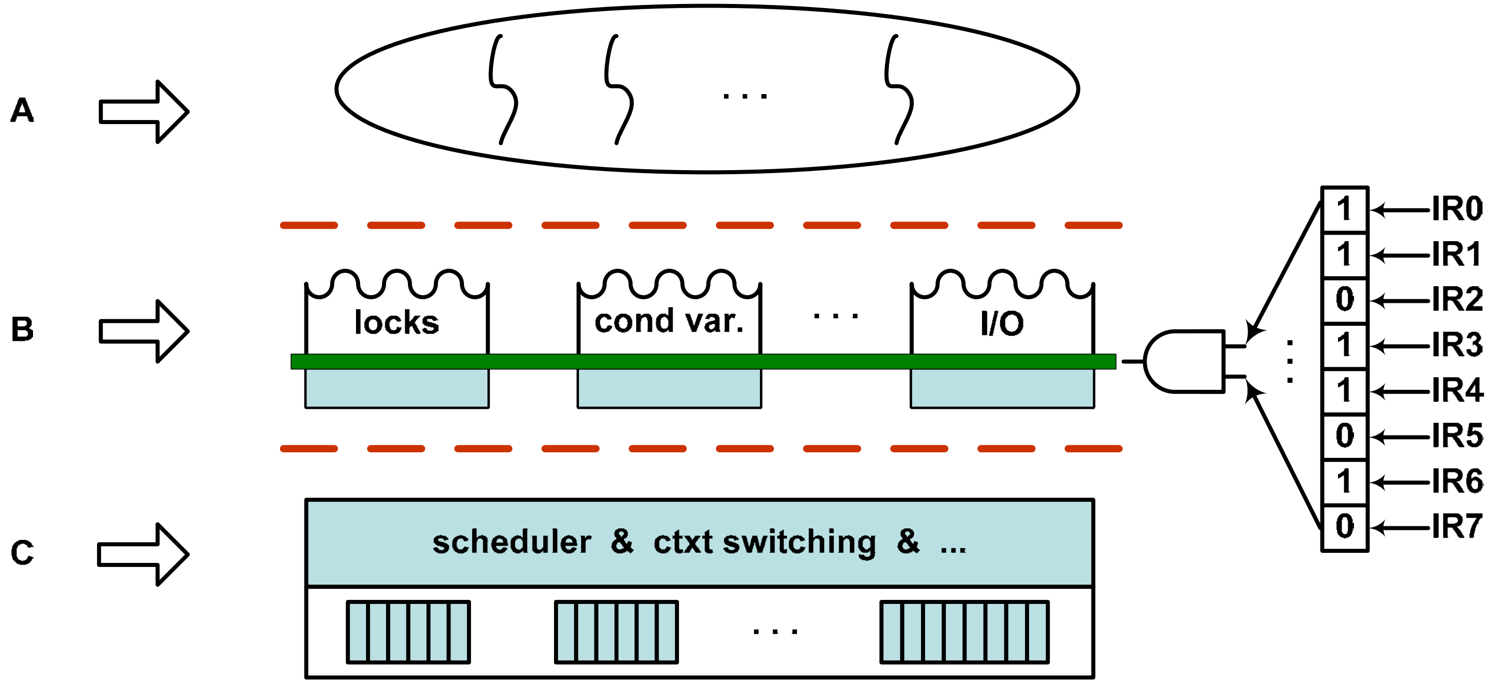
Our solution



Layering of miniOS Code



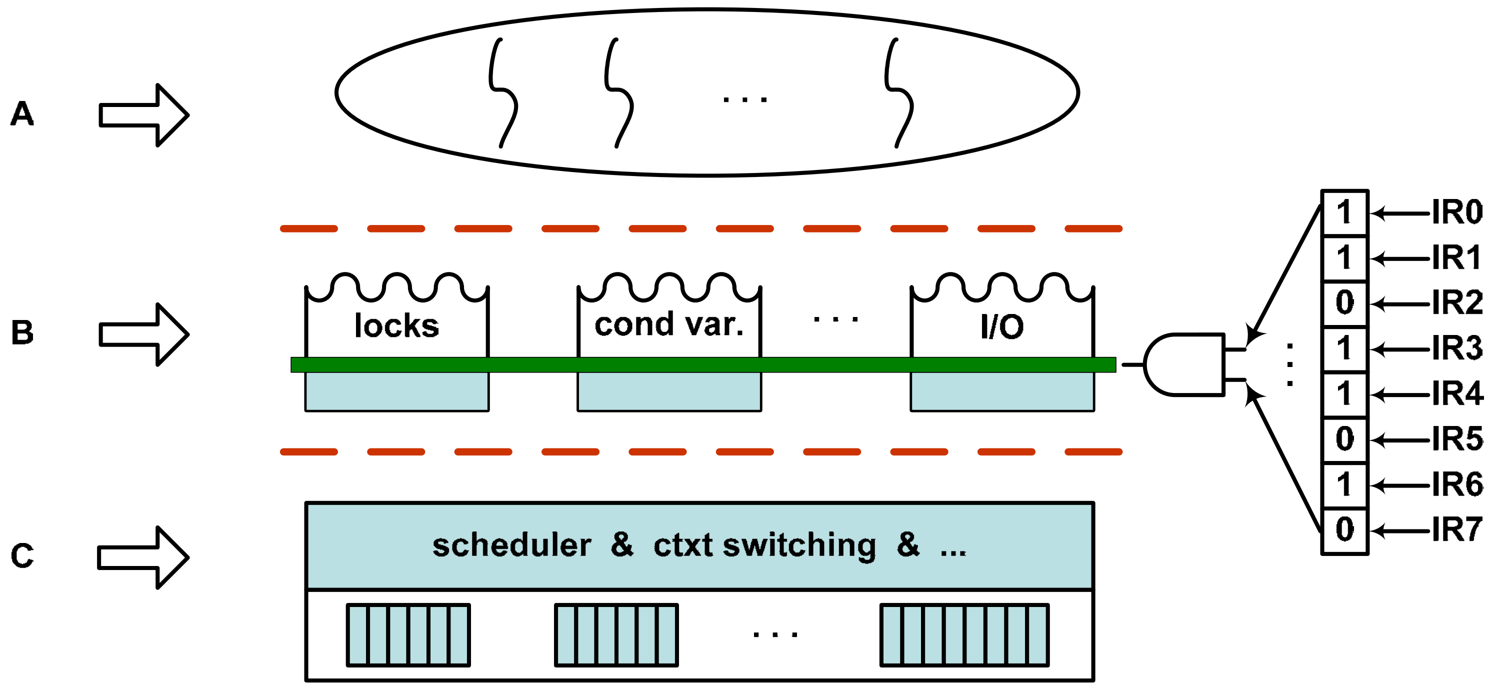
Layering of miniOS Code



C: low-level sequential code, might be tricky (e.g. context switching)

[Feng et al. PLDI'06, Feng et al. TLDI'07, Ni et al. TPHOLs'07]

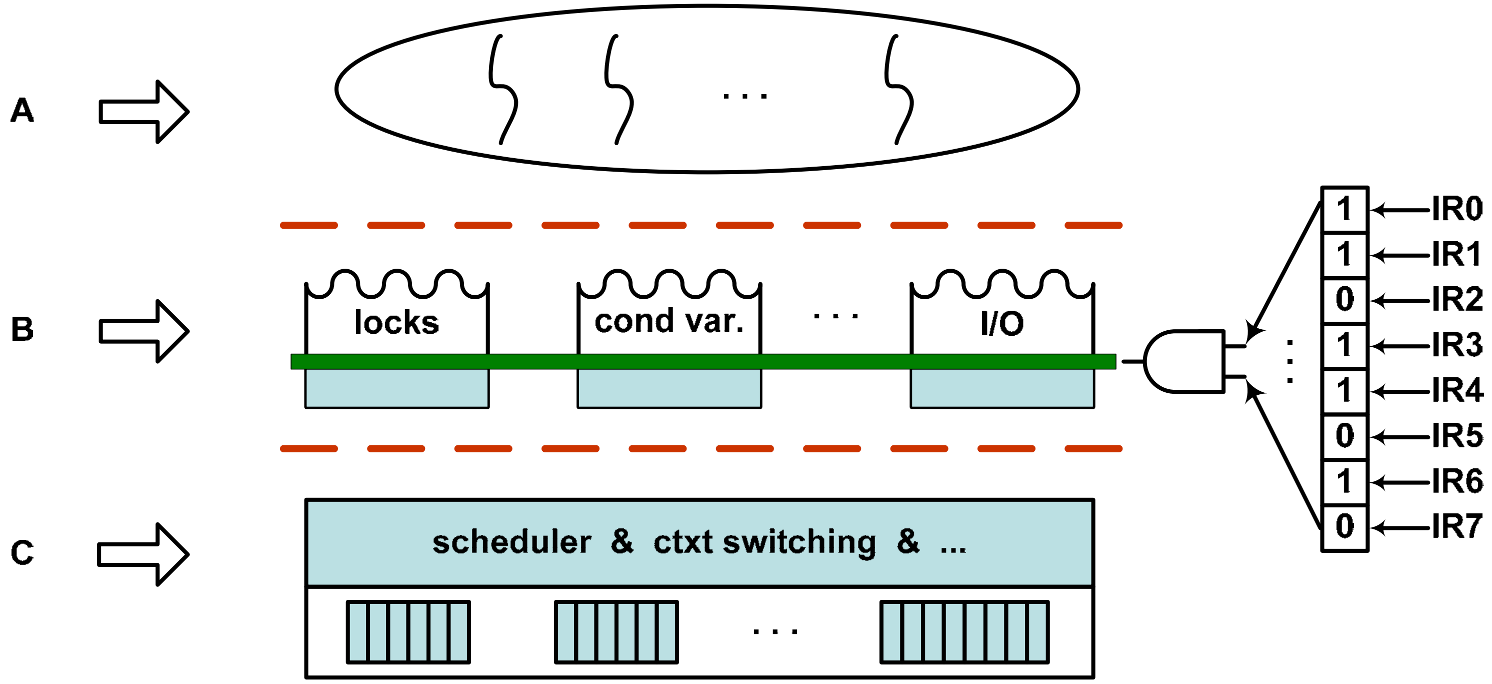
Layering of miniOS Code



A: concurrent code, Rely-Guarantee, CSL, …

PCC: [Yu&Shao ICFP’04, Feng&Shao ICFP’05, Feng et al. ESOP’07]

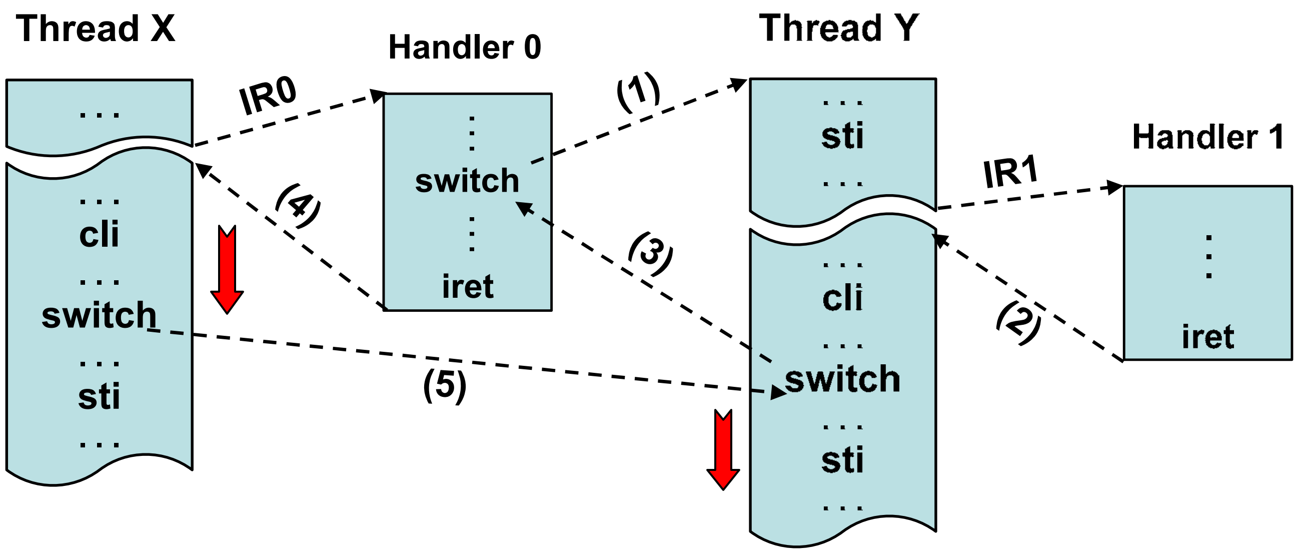
Layering of miniOS Code



B: concurrent code with explicit interrupts

How to certify ???

Concurrency with Interrupts: Challenges



Asymmetric preemption between handlers and non-handler code

Intertwining between threads and handlers

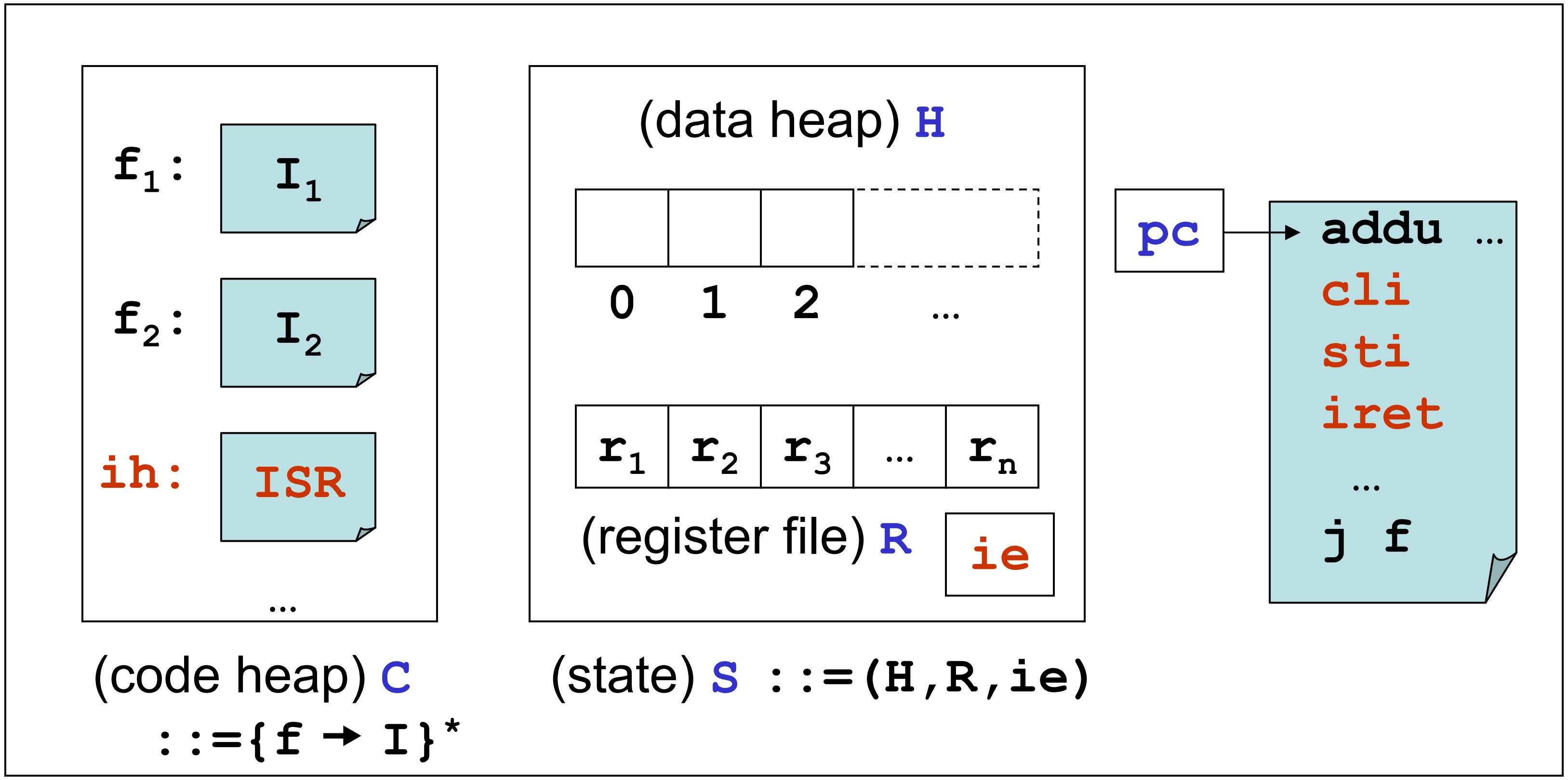
Asymmetric synchronization: cli/sti are different from locks

Nesting of handlers: may not respect priorities

Our contributions [PLDI'08]

- An abstract machine and operational semantics
  - Interrupts, switch, block/unblock
- Program logics
  - Modeling concurrency/interrupt primitives in terms of memory ownership transfer
- Certifying implementation of sync. primitives
  - Locks
  - Condition variables
    - Hoare style, Brinch-Hanson style, Mesa style

AIM – I : the machine



(program) P::=(C,S,pc)

AIM – I : operational semantics

Using non-determinism to model interrupts:

$$\frac{\mathbf{NextS}_{C(pc)}(S, S') \qquad \mathbf{NextPC}_{C(pc)}(pc, pc')}{(C, S, pc) \rightarrow (C, S', pc')} \quad \textbf{(seq)}$$

$$\frac{S.ie = 1 \qquad S' = S|_{ie=0}}{(C, S, pc) \Rightarrow (C, S', ih)} \quad \textbf{(irq)}$$

$$\frac{P \rightarrow P' \quad \vee \quad P \Rightarrow P'}{P \rightarrow P'} \quad \textbf{(step)}$$

Example: Teeter-Totter

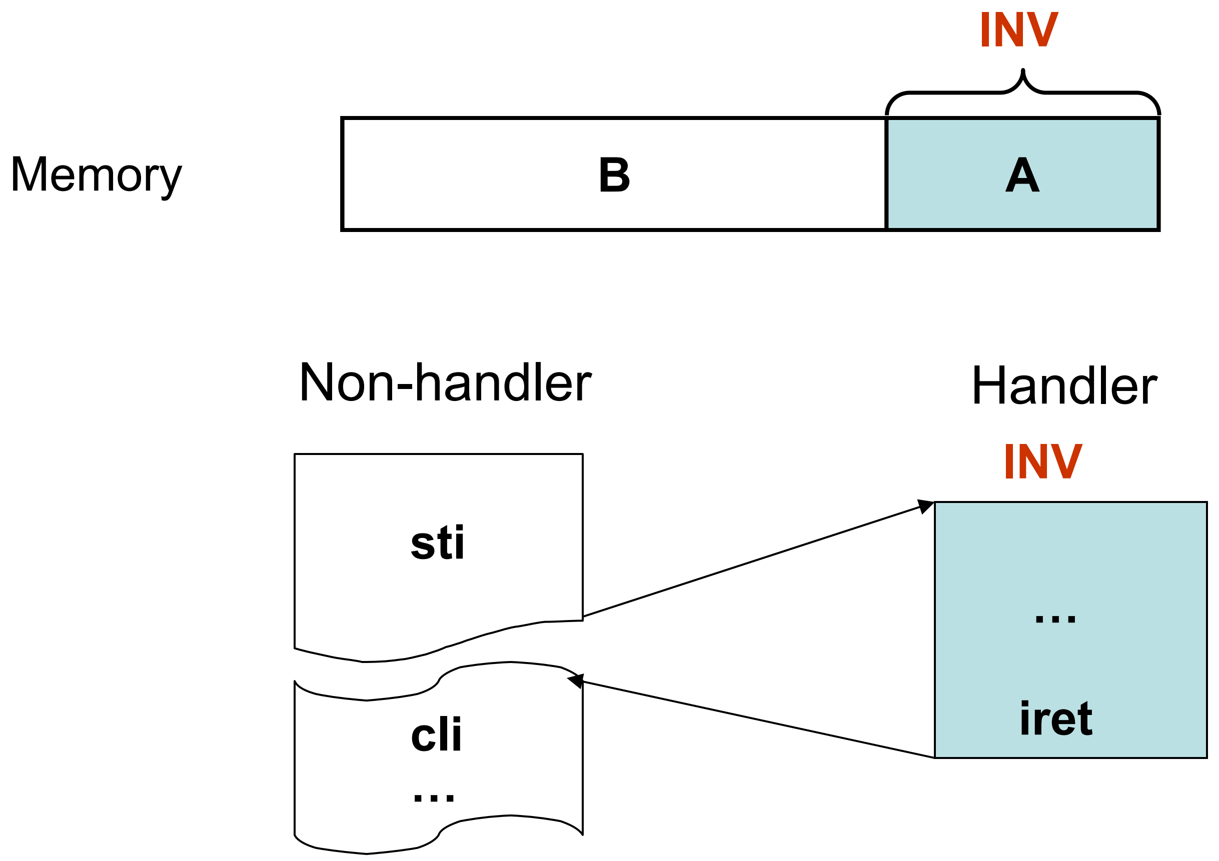
left	right
50	50

```
while(true){
  cli;
  if([right] == 0){
      sti;
      break;
  }
  [right] := [right]-1;
  [left]  := [left]+1;
  sti;
}
print("left wins!");
```

```
timer:
  if([left] == 0){
    print("right wins!");
    iret;
  }
  [left]  := [left]-1;
  [right] := [right]+1;
  iret;
```

How to guarantee non-interference?

AIM – I : the memory model

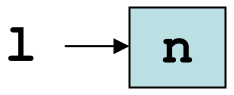


The memory partition is logical, not physical!

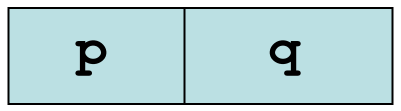
Separation logic to enforce partition

`emp` empty heap

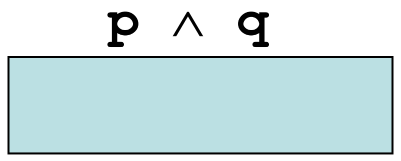
`l` $\rightarrow$ **`n`**



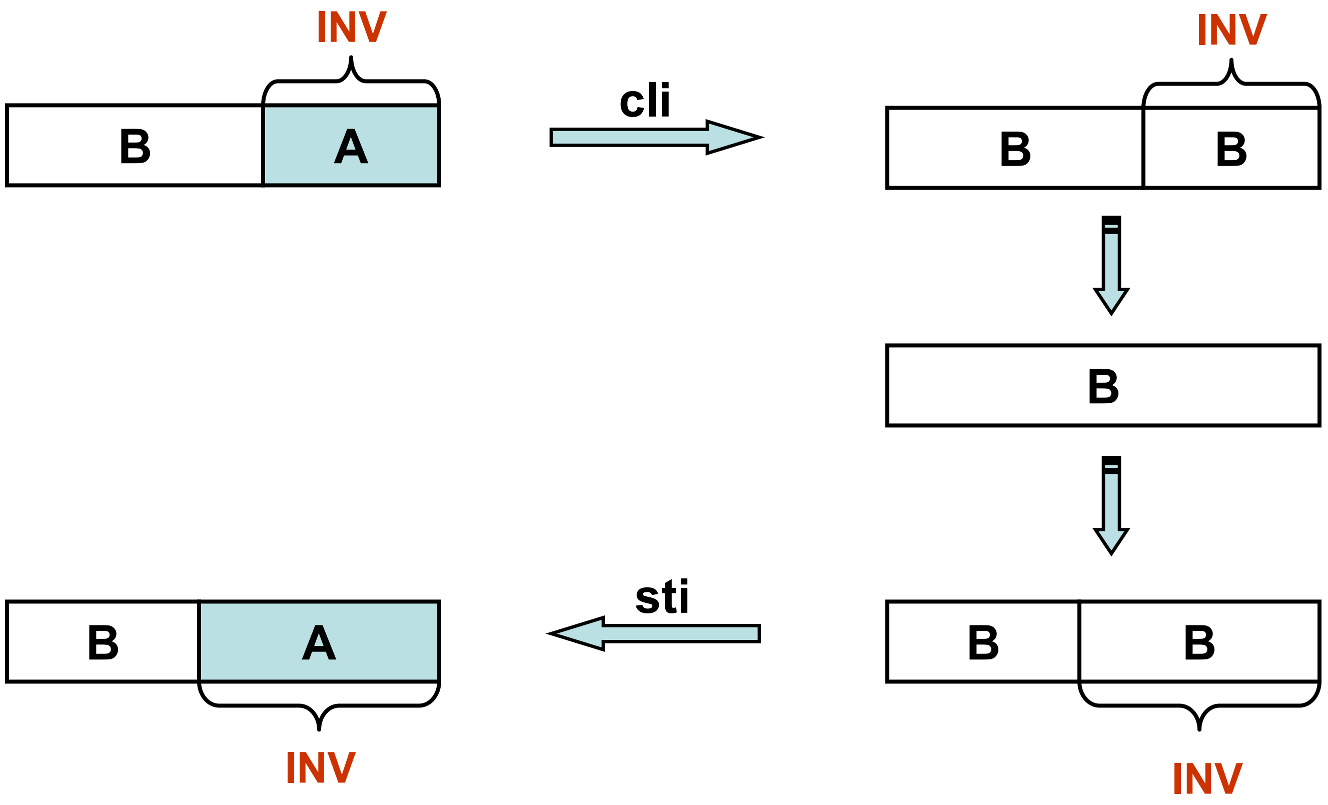
`p * q`



`p` $\wedge$ **`q`**



AIM – I : `cli/sti`



ie = 1

ie = 0

Example: Teeter-Totter

INV: $\exists m, n.\ (\text{left} \rightarrow m) * (\text{right} \rightarrow n) \land (m+n = 100)$

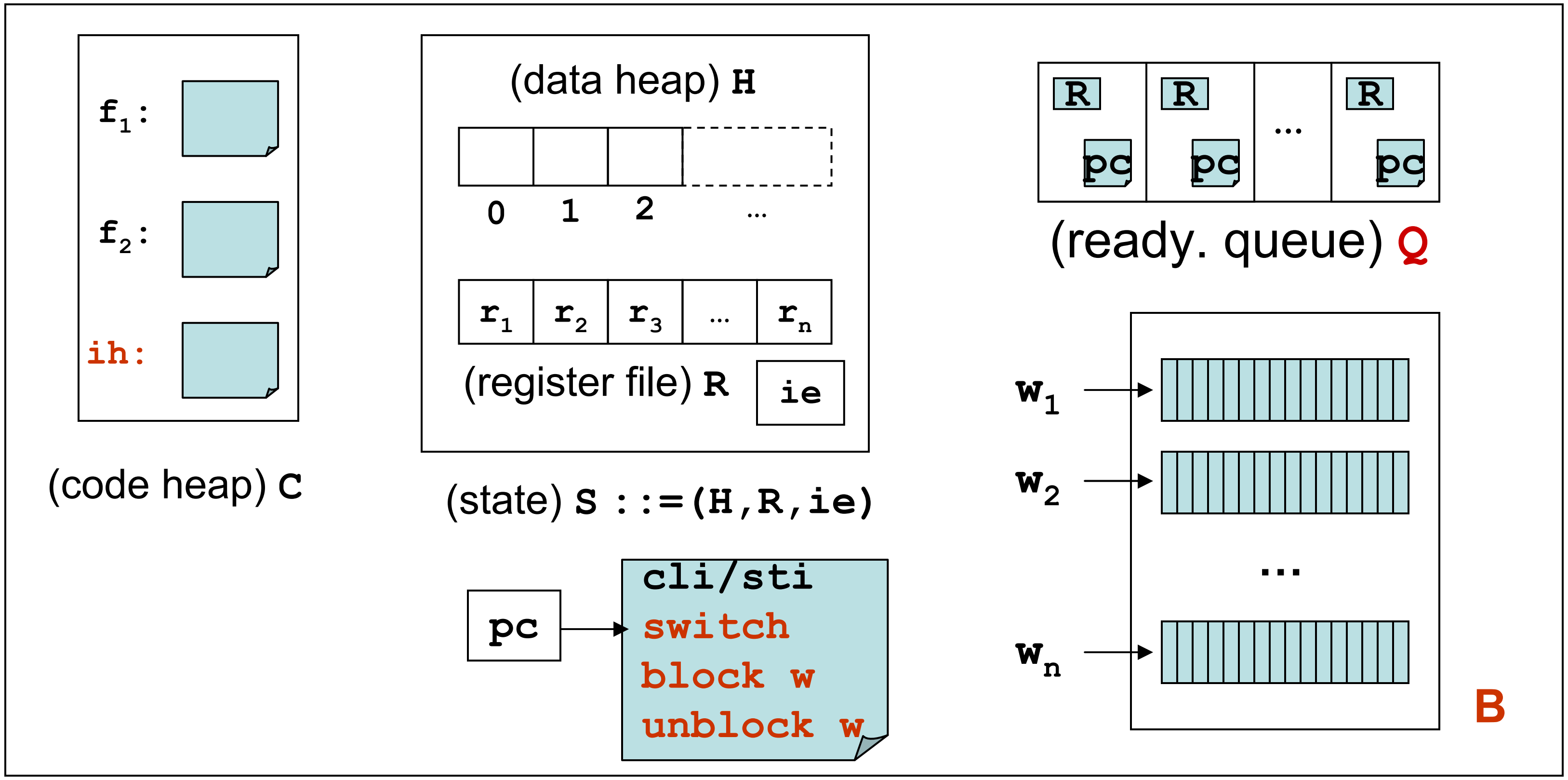
left	right
50	50

```
while(true){

  ~~cli~~;

  ...
  [right] := [right]-1;
  [left]  := [left]+1;

  sti;

}
```

```
sti
cli
```

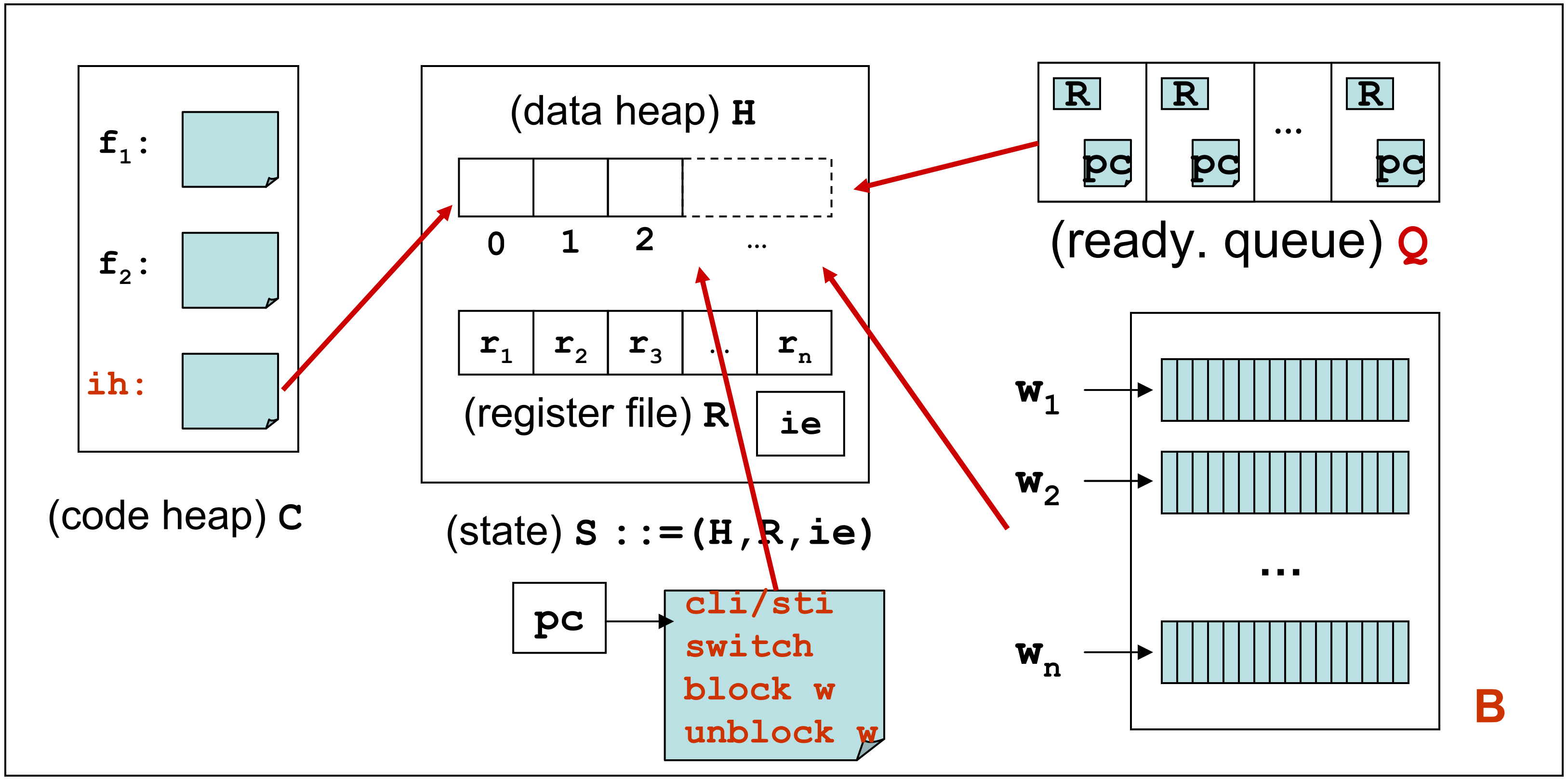
```
timer:

  if([left] == 0){
    print("right wins!");

    iret;
  }
  [left]  := [left]-1;
  [right] := [right]+1;

  iret;
```

AIM-II : Multi-threaded code with interrupts



(program) P ::=(C,S,B,Q,pc)

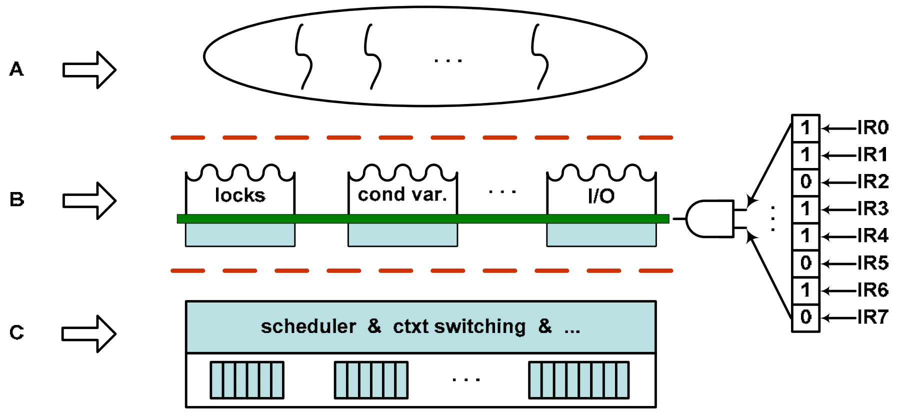
Non-interference?



Use memory partition to control interference!

AIM – II : `switch`

- A primitive implemented at layer C
  - Interrupt must be disabled before executing switch
- Operational semantics:
  - Put the current thread into ready Q
  - Resume a thread in Q (non-deterministic op)



Examples

```
timer_0:
    ...
    switch
    ...
    iret
```

```
yield:
    cli
    switch
    sti
    ret
```

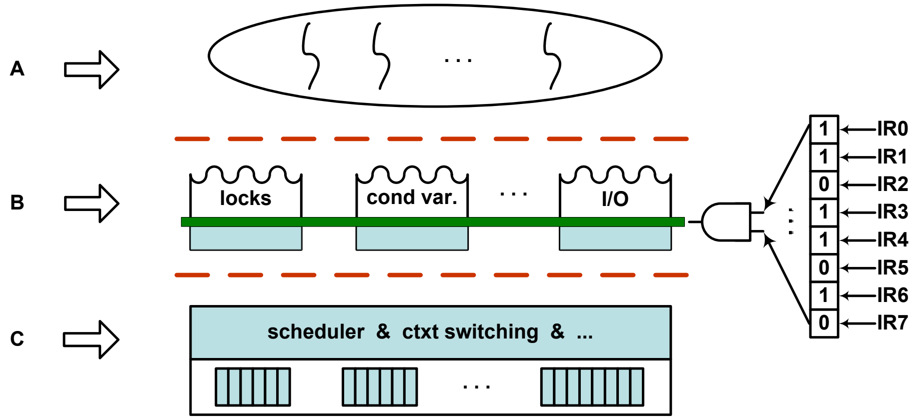
```
timer_1:
    iret
```

Layer A:

ie = 0: non-preemptive threads

ie = 1 & timer_1:
non-preemptive threads

ie = 1 & timer_0:
preemptive threads

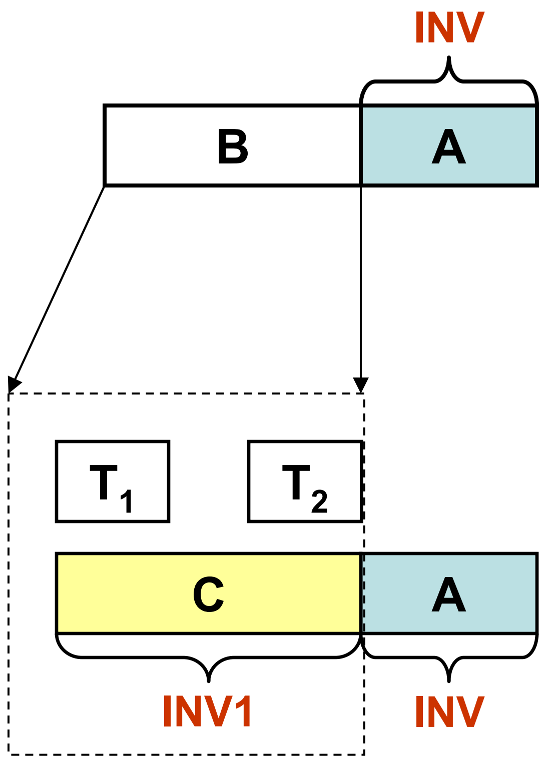


Example: spin locks

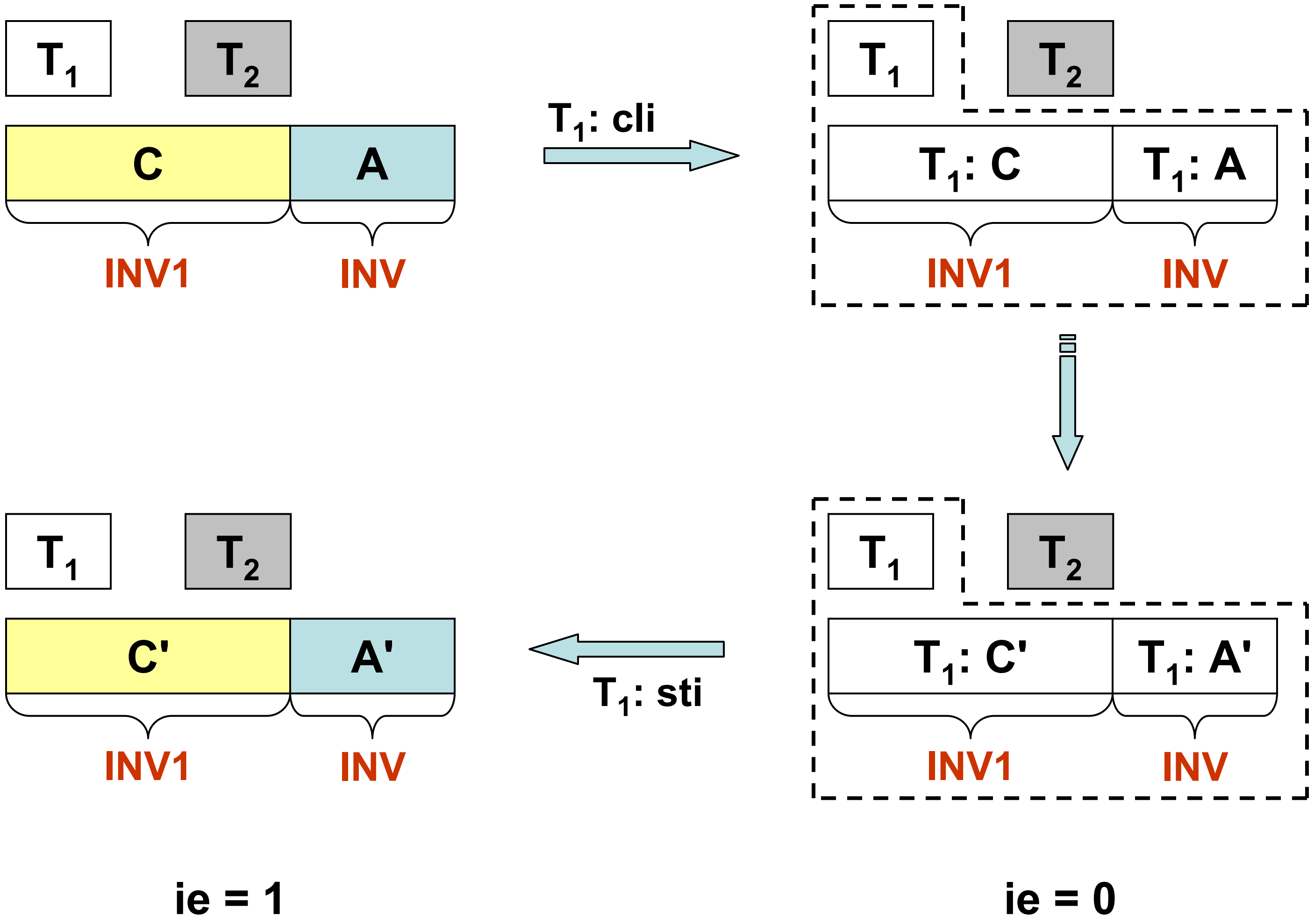
```
acquire (l):
    cli;
    while([l] == 0){
        switch
    }
    [l] := 0;
    sti;
    return;
```

```
release (l):
    cli;
    [l] := 1;
    sti;
    return;
```

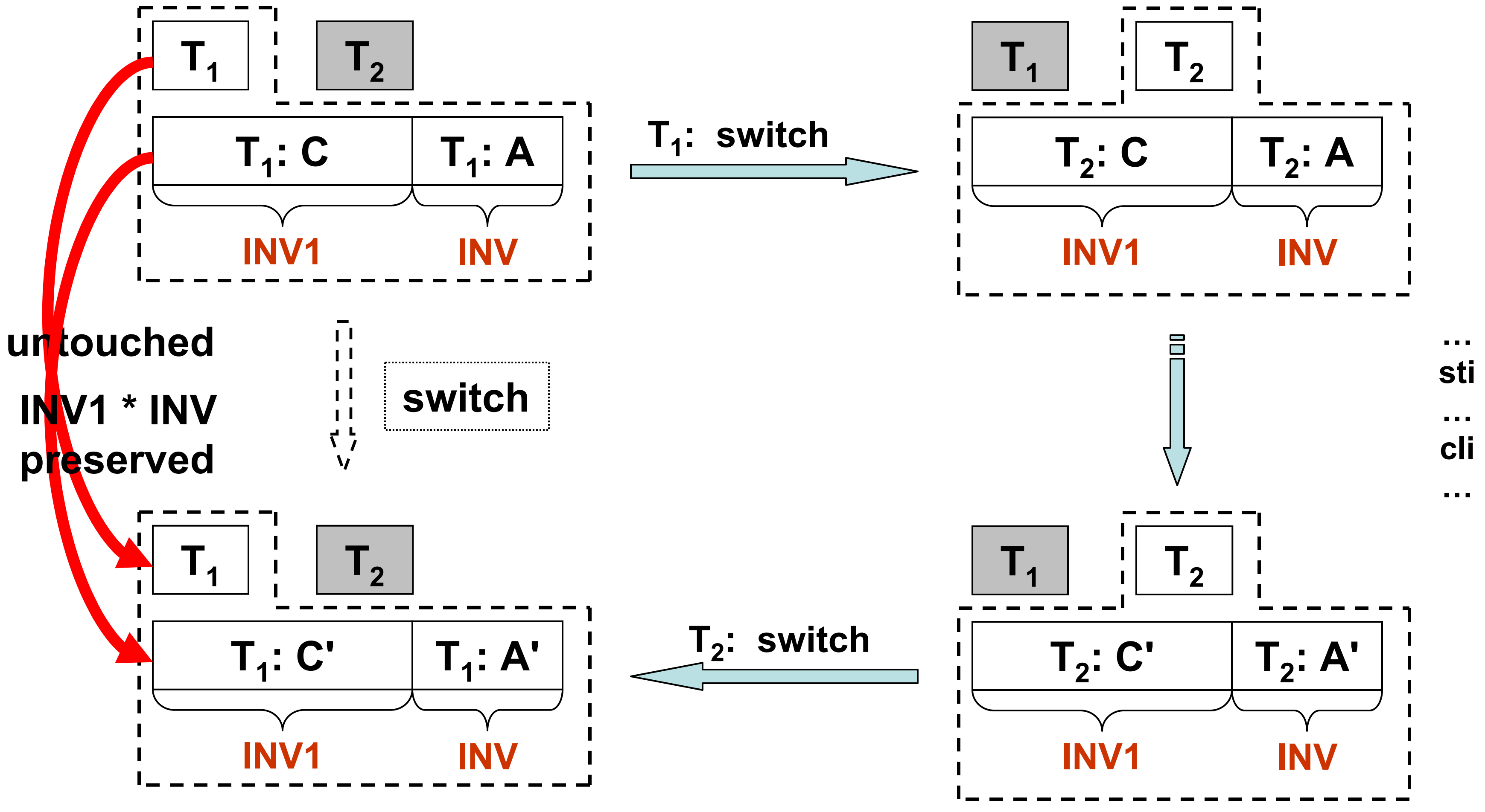
AIM – II : memory model



AIM – II : `cli/sti`

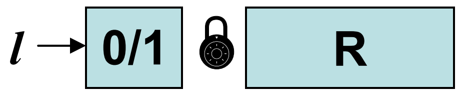


AIM – II : `switch`



ie = 0

Example: spin locks

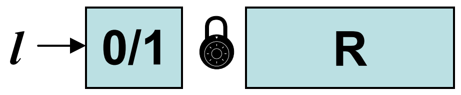


INV1: $\exists b.\ (l \mapsto b) * (b=0 \wedge \mathbf{emp} \vee b=1 \wedge R)$

```
-{ emp }
acquire (l):
    cli;
    while([l] == 0){
        sti;
        cli;
    }
    [l] := 0;
    sti;
    return;
-{ R }
```

```
-{ R }
release (l):
    cli;
    [l] := 1;
    sti;
    return;
-{ emp }
```

Example: spin locks



INV1: $\exists b. (l \mapsto b) * (b=0 \wedge emp \vee b=1 \wedge R)$

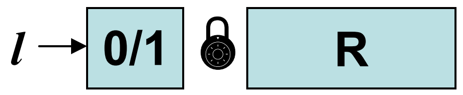
```
-{ emp }
acquire (l):
    cli;
                -{ emp * INV1 * INV0}
    while([l] == 0){
                -{ emp * INV1 * INV0}
        sti;
                -{ emp }
        cli;
                -{ emp * INV1 * INV0}
    }
                -{ (l ↦ 1) * R * INV0}
    [l] := 0;
                -{ (l ↦ 0) * R * INV0}
                -{ INV1 * INV0 * R}
    sti;
                -{ R}
    return;
-{ R }
```

AIM – II : block and unblock

- **`block rs`**
  - put current thread into the block queue **B(rs)**
  - pick a thread from ready queue to execute

- **`unblock rs, rd`**
  - move a thread from **B(rs)** to the ready queue
  - put the thread id into **rd**, put 0 if **B(rs)** empty
  - no context switching!

Examples: locks



$B(l) = Q_l$

```
acquire_0(l):
    cli;
    if ([l] == 0)
        block l;
    else
        [l] := 0;
    sti;
    return;
```

```
acquire_1(l):
    cli;
    while ([l] == 0)
        block l;
    [l] := 0;
    sti;
    return;
```

```
release_0(l):
    local x;
    cli;
    unblock l x;
    if (x == 0)
        [l] := 1;
    sti;
    return;
```

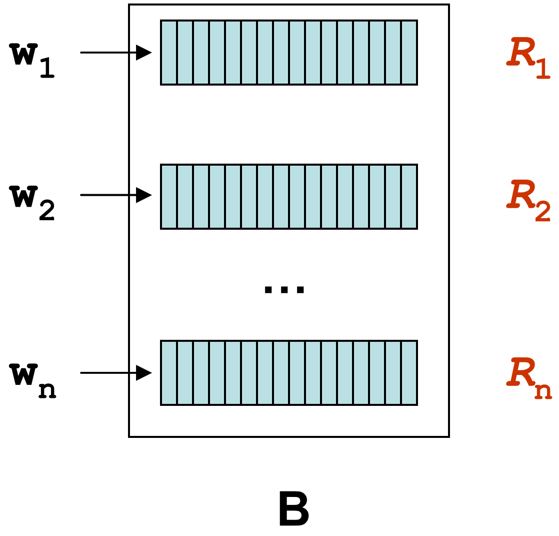
```
release_1(l):
    local x;
    cli;
    unblock l x;
    [l] := 1;
    sti;
    return;
```

How to interpret `block/unblock`

Threads block themselves to wait for resources.

locks: wait for resources protected by locks

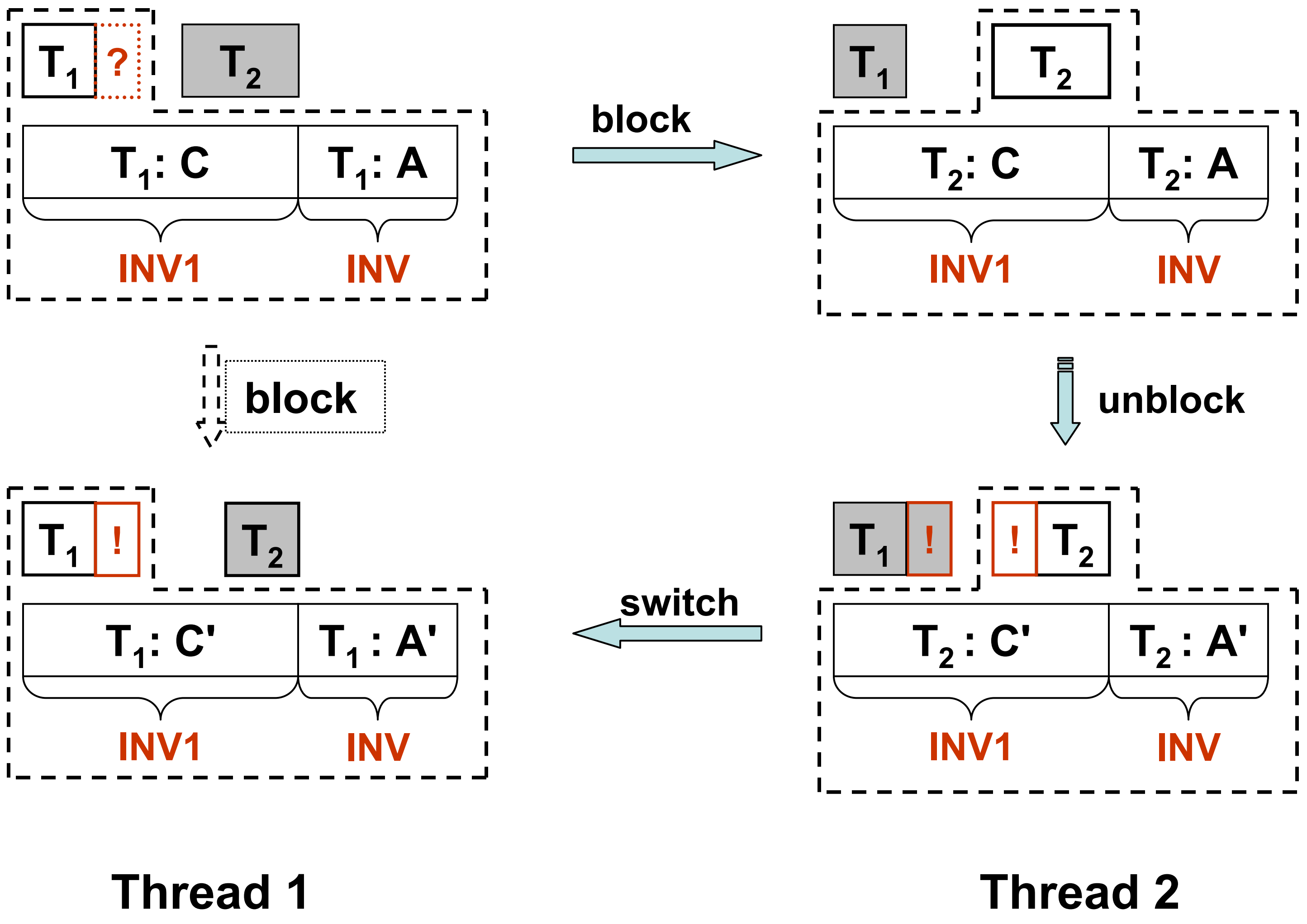
condition variables:
wait for resources over which the condition holds



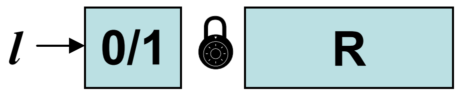
$\Delta ::= \{w_0 \to R_0, \ldots, w_n \to R_n\}$

R_i can be emp !

AIM-II: `block/unblock`



Examples: locks

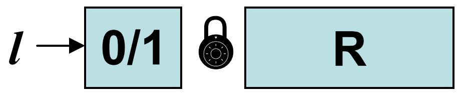


$B(l) = Q_l$ $\Delta(l) = R$

INV1: $\exists b, (l \mapsto b) * (b=0 \wedge emp \vee b=1 \wedge R)$

```
-{ emp }
acquire_0(l):
                        -{ emp }
    cli;
                        -{ emp * INV1 * INV0}
    if ([l] == 0)
                        -{ emp * INV1 * INV0}
        block l;
                        -{ emp * INV1 * INV0 * Δ(l) }
    else
                        -{ emp * (l ➡ 1) * R * INV0}
        [l] := 0;
                        -{ emp * (l ➡ 0) * R * INV0}
    sti;
                        -{ emp * INV1 * INV0 * R}
    return;
                        -{ emp *                R}
-{ R }
```

Examples: locks



$B(l) = Q_l$ $\Delta(l) = R$

INV1: $\exists b, (l \mapsto b) * (b=0 \wedge emp \vee b=1 \wedge R)$

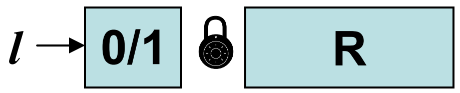
-{ R }

release_0(l):

```
    local x;
                         -{ R }
    cli;
                         -{ R * INV1 * INV0}
    unblock l x;
                         -{(x=0 ∧ R ∨ x ≠0 ∧ emp) * INV1 * INV0}
    if (x == 0)
                         -{ (l ↦ 0) * R * INV0}
      [l] := 1;
                         -{ (l ↦ 1) * R * INV0}
    sti;
                         -{ INV1 * INV0}
    return;
                         -{ emp }
```

-{ emp }

Examples: locks



$\mathbf{B}(l) = \mathbf{Q}_l$ $\quad$ ~~$\Delta(l) = \mathbf{R}$~~ $\quad$ $\Delta(l) = \mathbf{emp}$

INV1: $\exists b, (l \mapsto b) * (b{=}0 \wedge \mathbf{emp} \vee b{=}1 \wedge \mathbf{R})$

```
-{ emp }
acquire_1(l):
    cli;
    while ([l] == 0)
        block l;
    [l] := 0;
    sti;
    return;
-{ R }
```

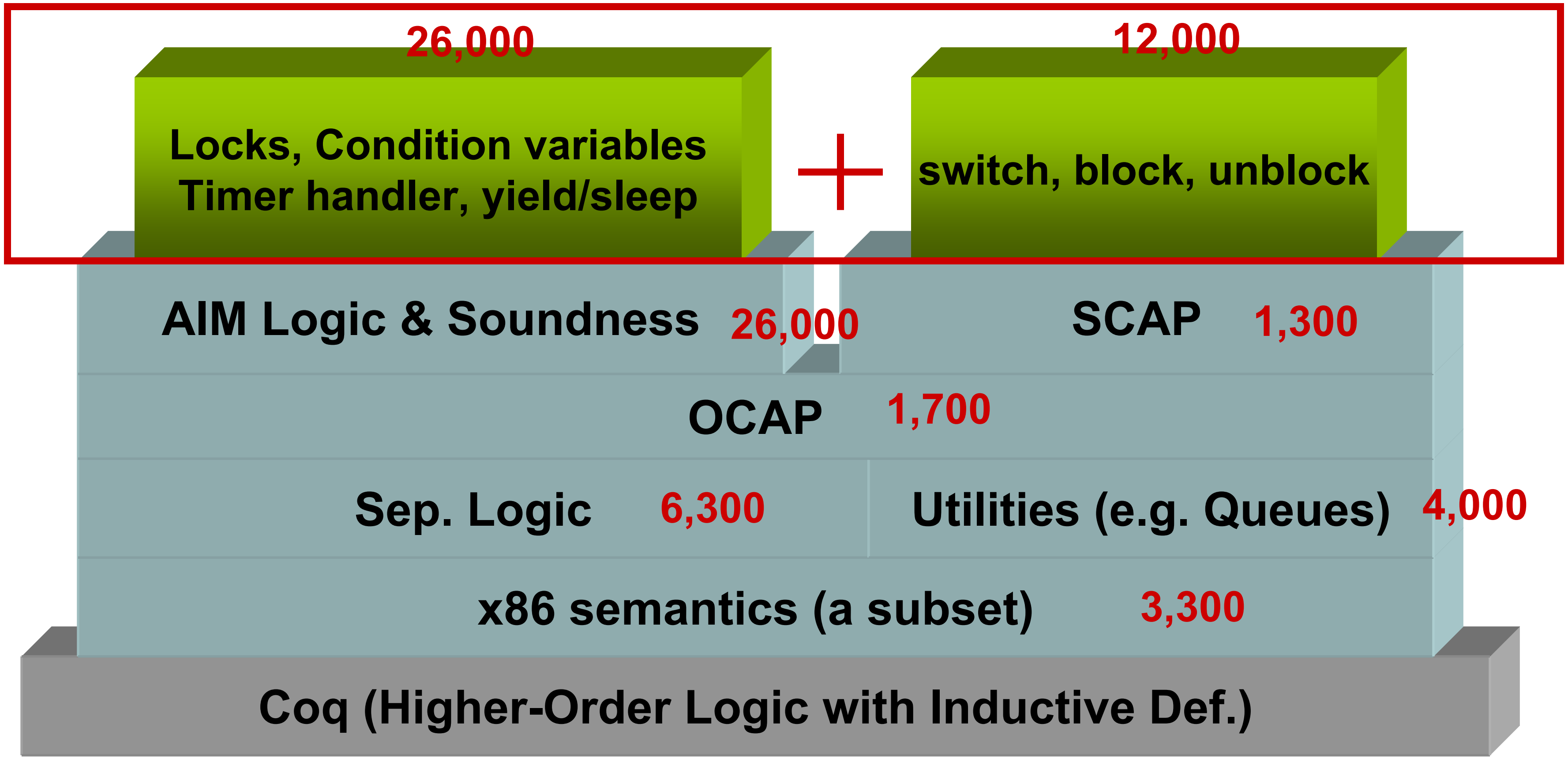
```
-{ R }
release_1(l):
    local x;
    cli;
    unblock l x;
    [l] := 1;
    sti;
    return;
-{ emp }
```

Examples : condition variables

- **`wait()/notify()`**
- Hoare Style / Brinch-Hanson Style
- Mesa style
- See the full paper:

  http://flint.cs.yale.edu/publications/aim.html

Implementations in Coq



Around 82,000 lines of Coq code

See http://flint.cs.yale.edu/publications/aim.html

Other work from my group

(see http://flint.cs.yale.edu)

- Specialized program logics
  - SCAP: stack-based control abstractions **[PLDI'06]**
  - SAGL: modular concurrency verification **[ESOP'07]**
  - CMAP: dynamic thread creation **[ICFP'05]**
  - GCAP: dynamic loading & self-modifying code **[PLDI'07]**
  - Certified garbage collectors & mutators **[PLDI'07, TASE'07]**
  - Certified context switch libraries **[POPL'06, TPHOLs'07]**
  - Preemptive concurrency with interrupts **[PLDI'08, VSTTE'08]**

- New framework for end-to-end verification/linking
  - OCAP: embedding and interoperation between different verification systems **[TLDI'07]**
  - interoperability based on semantic models **[ongoing]**

Open research problems

- Building practical certified system software
  - OS kernels, device drivers, hypervisor
  - System software for future ubiquitous computing devices
- Developing formal layers of abstractions for future computing
  - Concurrency?
  - Information flow?
  - Safety and liveness properties?
  - Key design patterns?
- Developing new languages & tools for certified programming
  - New programming languages for constructing proofs
  - New certified/certifying programming tools and compilers

What programming will be like if our very initial goal is to build certified software?