Typed Assembly Language for Implementing OS Kernels in SMP/Multi-Core Environments with Interrupts

Toshiyuki Maeda and Akinori Yonezawa

University of Tokyo

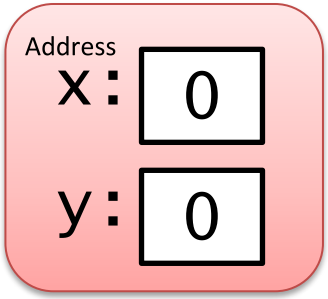
Quiz

Q. Can execution of the following 2 threads yield the result "`r1 = 0` and `r2 = 0`"?

```
Thread1:
        st [x] ← 1
        ld r1 ← [y]
```

```
Thread2:
        st [y] ← 1
        ld r2 ← [x]
```

Initial state of shared memory



[Environment]
CPU: Intel Xeon X5570 (2.93GHz) x 8
OS: Linux

Q. Can execution of the following 2 threads yield the result "`r1 = 0` and `r2 = 0`"?

```
Thread1:
        st [x] ← 1
        ld r1 ← [y]
```

```
Thread2:
        st [y] ← 1
        ld r2 ← [x]
```

A. Yes

[Environment]
CPU: Intel Xeon X5570 (2.93GHz) x 8
OS: Linux

Quiz 2

Q. How often does it occur?

- 1. Once per second or more
- 2. Once a minute
- 3. Once an hour
- 4. Once a day
- 5. Once a month
- 6. Once a year (or less)

[Environment]
CPU: Intel Xeon X5570 (2.93GHz) x 8
OS: Linux

Q. How often does it occur?

A.

- 1. Once per second or more
- 2. Once a minute
- 3. Once an hour
- 4. Once a day
- 5. Once a month
- 6. Once a year (or less)

[Environment]
CPU: Intel Xeon X5570 (2.93GHz) x 8
OS: Linux

Typed Assembly Language for Implementing OS Kernels in SMP/Multi-Core Environments with Interrupts

or

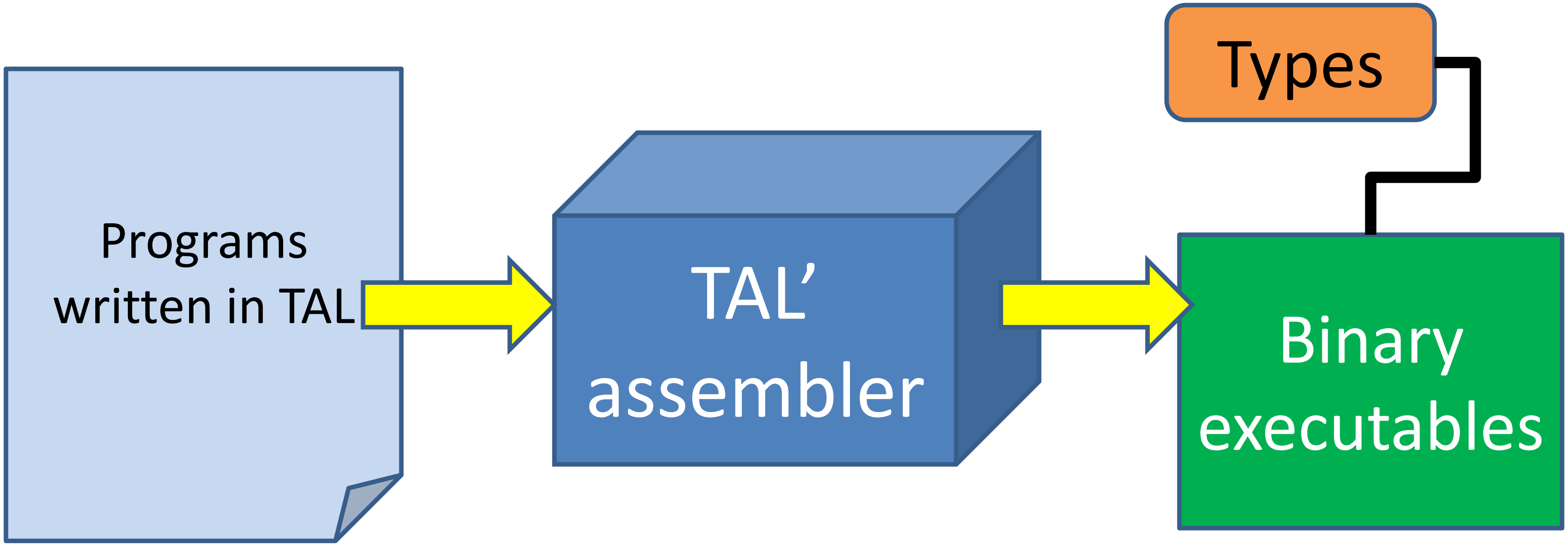
Typed Assembly Language for Implementing Ad Hoc Synchronization Correctly

What is Typed Assembly Language (TAL)?

- "Strongly-typed" assembly language
  - Its type-checking ensures two safety
    - Memory safety
    - Control-flow safety
  - Except for being typed,
    it is an ordinary assembly language
    - It was first introduced in the field of
      type-preserving compilation [Morrisett et al. 1999]

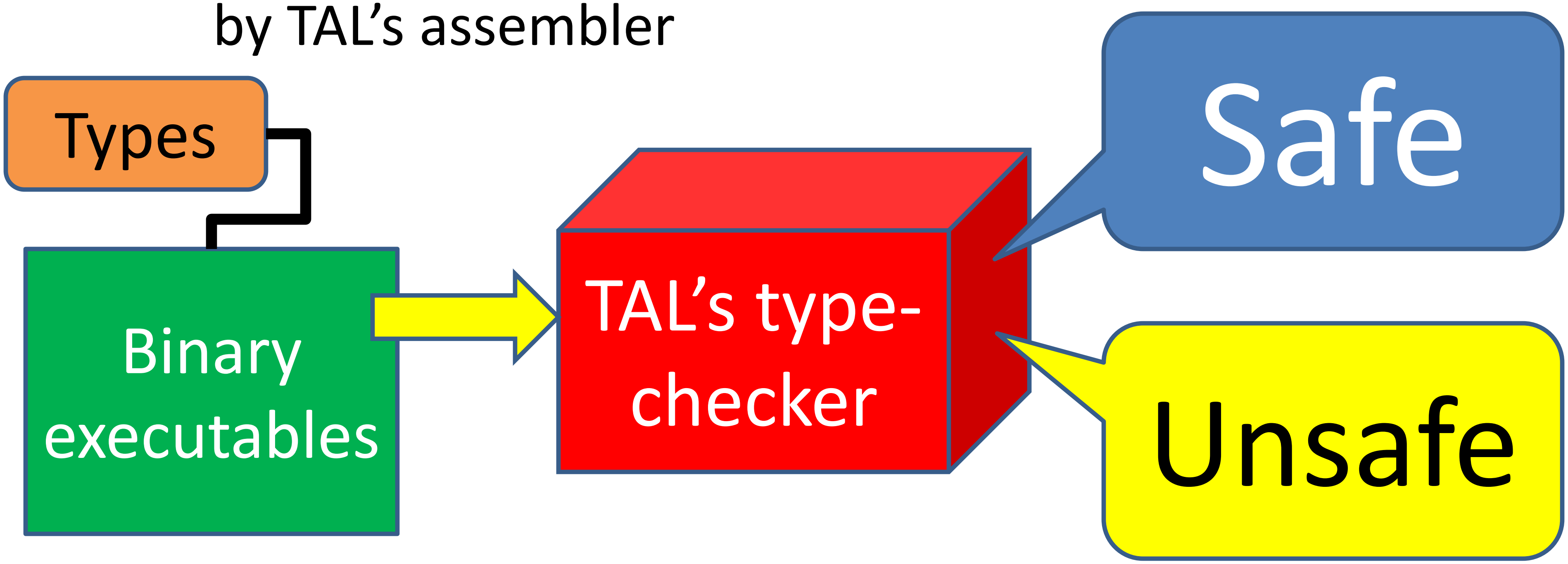
Overview of TAL's framework: generating binary executables

- TAL's assembler generates not only binary executables, but also their type information



Overview of TAL's framework: type-checking binary executables

- TAL's type-checker can type-check binary executables
  - utilizing type information generated by TAL's assembler



TALK: TAL for Kernel [Maeda et al. 06, 08]

- TAL whose type system is extended in order to implement OS kernels
  - Memory management (malloc/free) and multi-thread management mechanisms can be written in TALK
    - It is impossible in conventional TALs
      - because they rely on external memory management (= GC)

Brief overview of TALK's type system

- Supports variable-length arrays as a language primitive
  - in order to represent memory regions
- Keeps track of integer constraints
  (in the same way as dependent type [Xi et al. 99])
  - in order to perform array-bound checking statically
- Keeps track of pointer aliases
  (in the same way as alias type [Walker et al. 00])
  - in order to realize safe strong update (explained in the next slide)
- Introduces notion of split/concatenation of arrays
  - in order to integrate variable-length arrays and alias type

What is strong update?

- Memory operation
  that modifies types of memory regions
  - Memory management (e.g., malloc/free)
    can be viewed as strong updates

Memory management as strong update

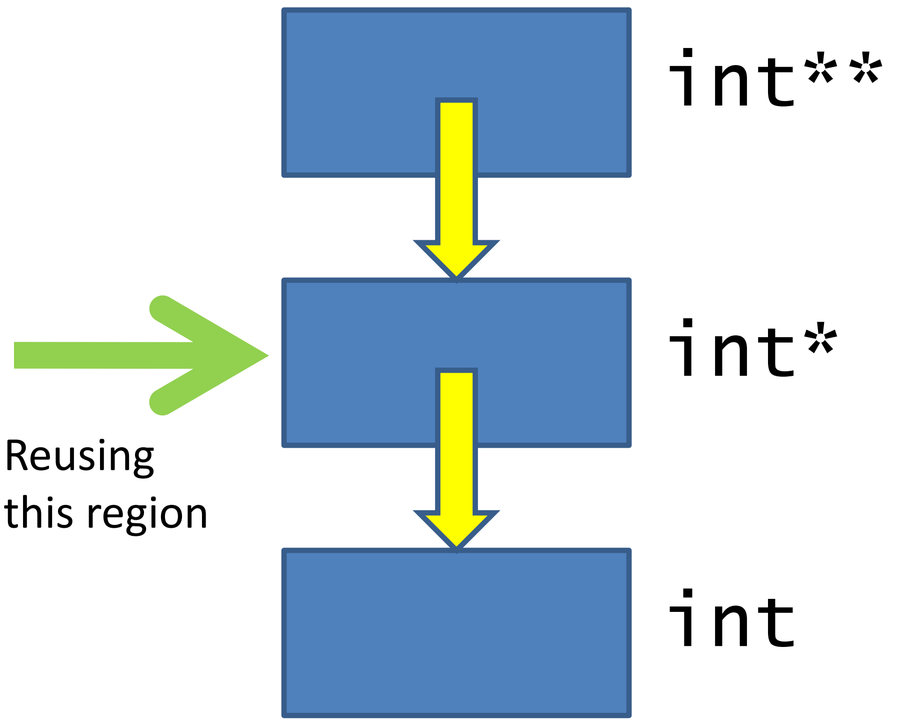
- Example of memory reuse
  - Reusing “`int*`” as “`int`”

```
int* reuse(int** o)
{
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

Memory management as strong update

- Example of memory reuse
  - Reusing "`int*`" as "`int`"

```
int* reuse(int** o)
{
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

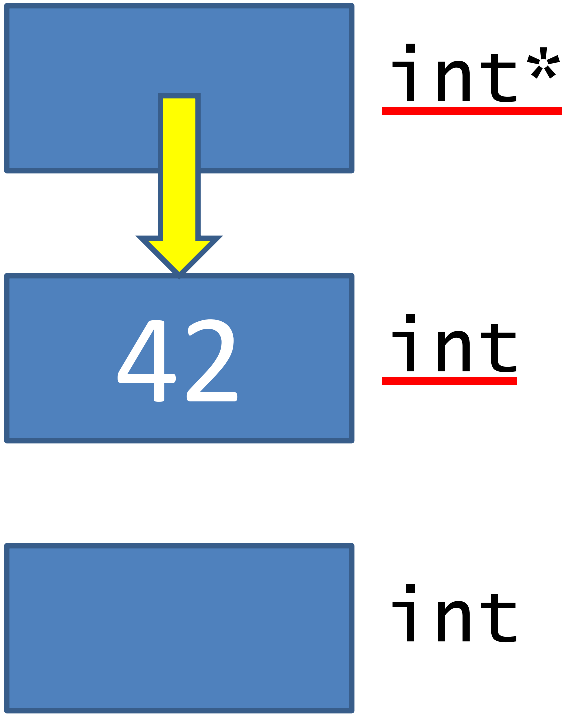


Memory management as strong update

- Example of memory reuse
  - Reusing “`int*`” as “`int`”

```
int* reuse(int** o)
{
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

Strong update

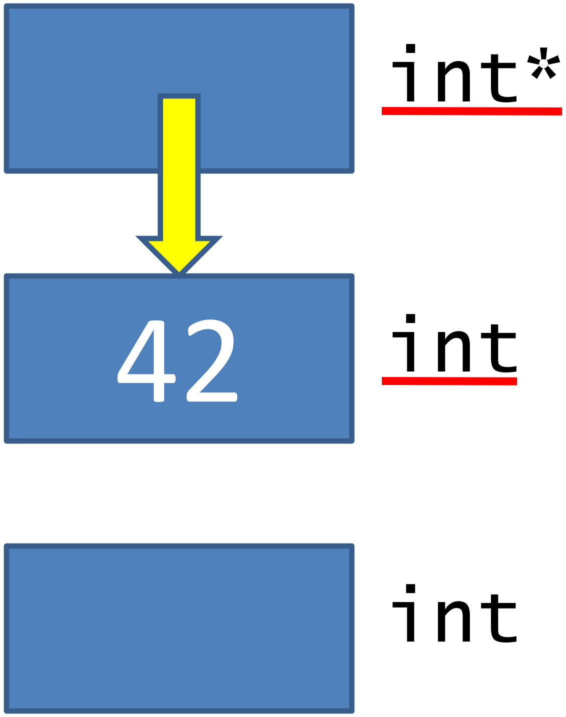


Memory management as strong update

- Example of memory reuse
  - Reusing "int*" as "int"

```
int* reuse(int** o)
{
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

Strong update



In general, strong updates are not always safe
because pointer o may be used in other locations

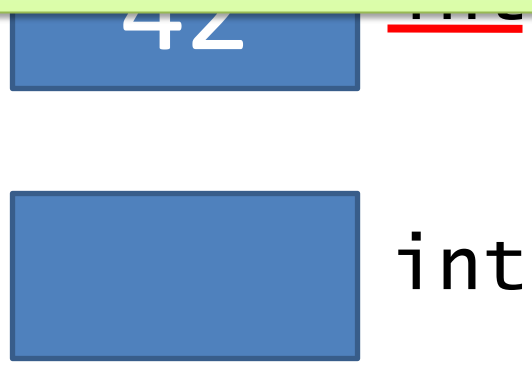
Memory management as strong update

- Example of memory reuse
  - Reusing “`int*`” as “`int`”

```
int* reuse(int** o)
{
    int* p = (int*)o;
    *p = 42;
    return p;
}
```

Strong update

Alias type system ensures that this strong update is safe
by ensuring that pointer o is not aliased with other pointers



In general, strong updates are not always safe
because pointer o may be used in other locations

Problem of the original TALK

- The original alias type system
  becomes unsound
  in SMP/Multi-core environments

Why unsound?

- The original alias type system does not keep track of pointer aliases between threads

Unsafe

if pointer o is being used by other threads

```
int* reuse(int** o)
{
  int* p = (int*)o;
  *p = 42;
  return p;
}
```

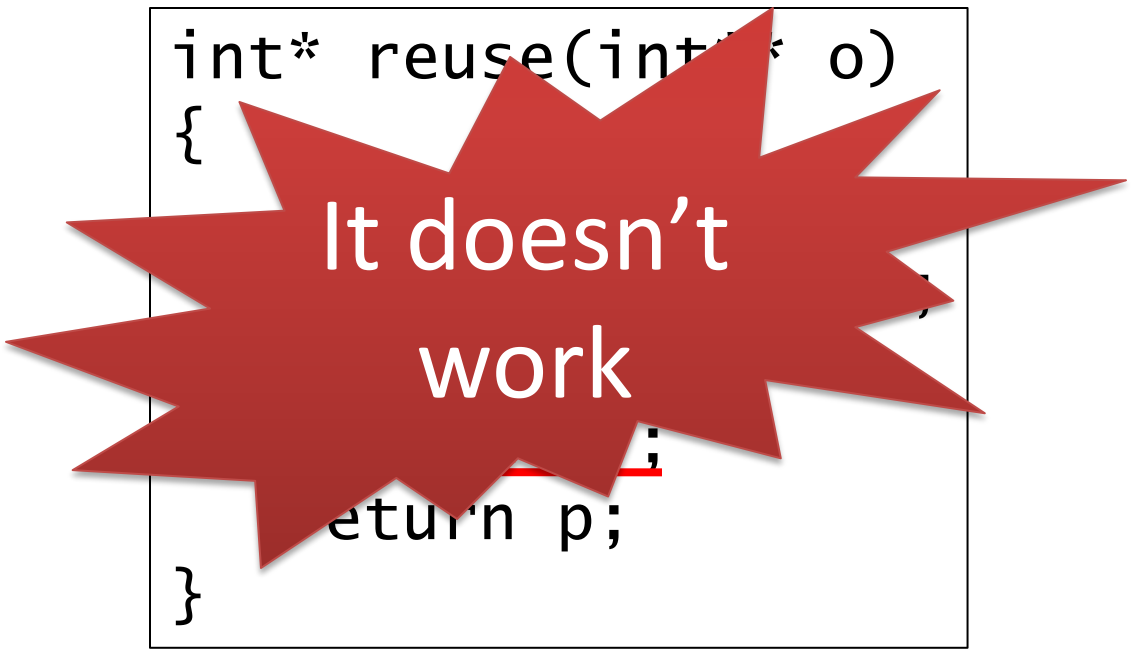
An approach to making it sound

- Introduce synchronization primitives
  - Lock/unlock, synchronized block, atomic block, etc.

```
int* reuse(int** o)
{
    lock(L);
    int* p = (int*)o;
    *p = 42;
    unlock(L);
    return p;
}
```

An approach to making it sound

- Introduce synchronization primitives
  - Lock/unlock, synchronized block, atomic block, etc.



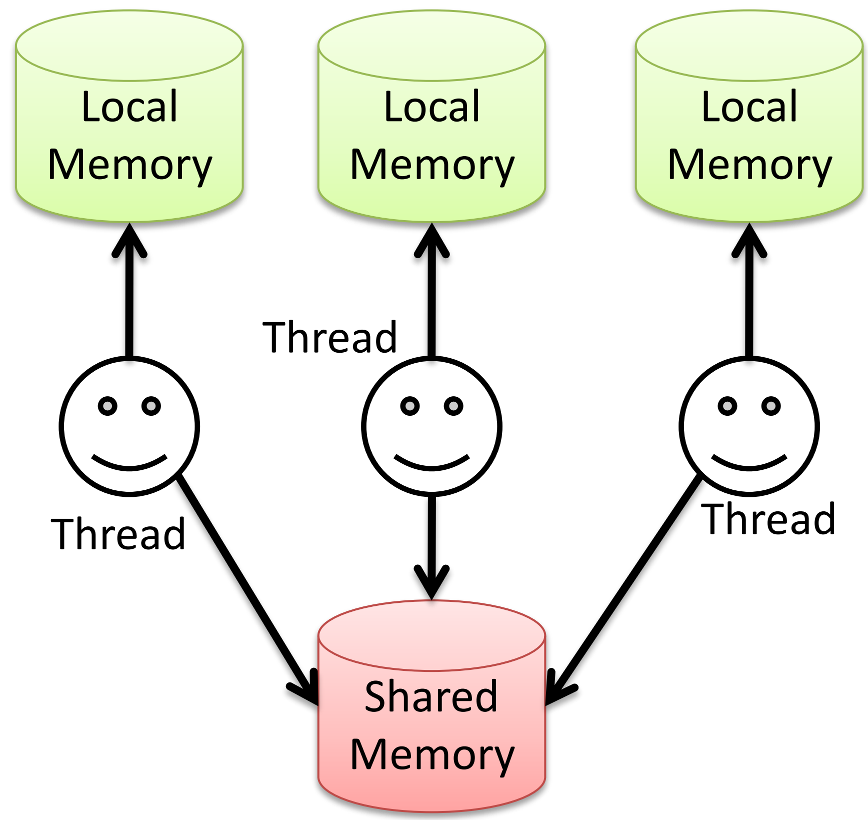
Why doesn’t it work?

- Sync primitives don’t help for safe strong update
  - They can ensure race-freedom etc., but don’t tell whether types are changed or not

- Sync primitives are not available when implementing OS kernels
  - OS kernels should provide them by using low-level CPU instructions

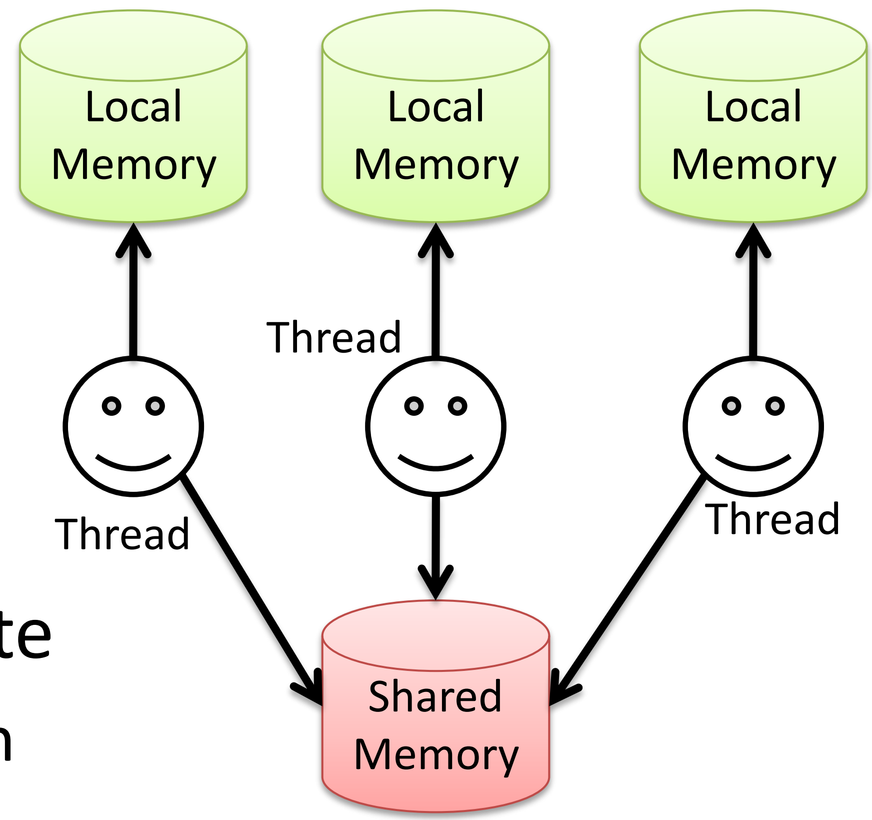
Our approach to safe strong update in SMP/multi-core environments (1 of 2)

- Classify memory types into two kinds:
  - Local memory
    - Only a dedicated thread can access
  - Shared memory
    - Multiple threads can access



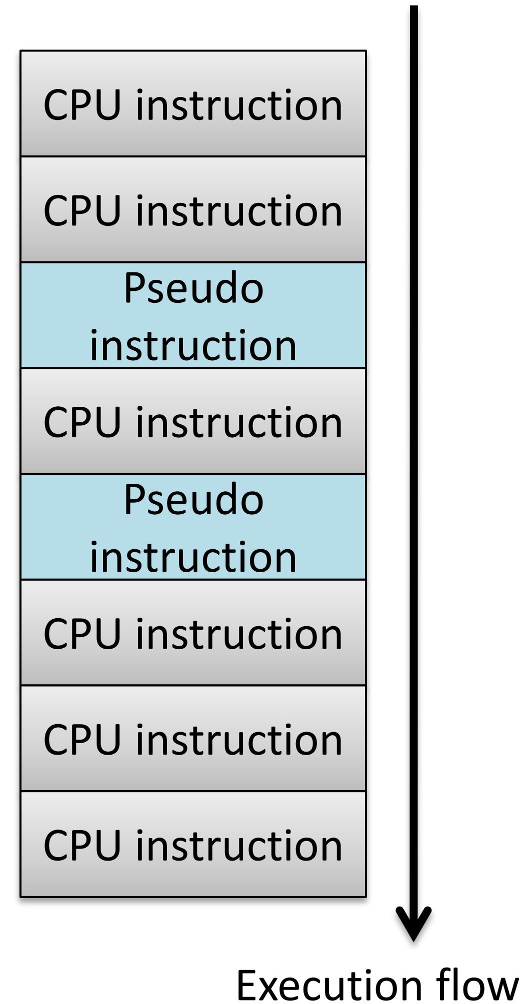
Our approach to safe strong update in SMP/multi-core environments (2 of 2)

- Local memory allows strong update
  - because other threads cannot access it
- Shared memory does not allow strong update
  - Except for a certain condition



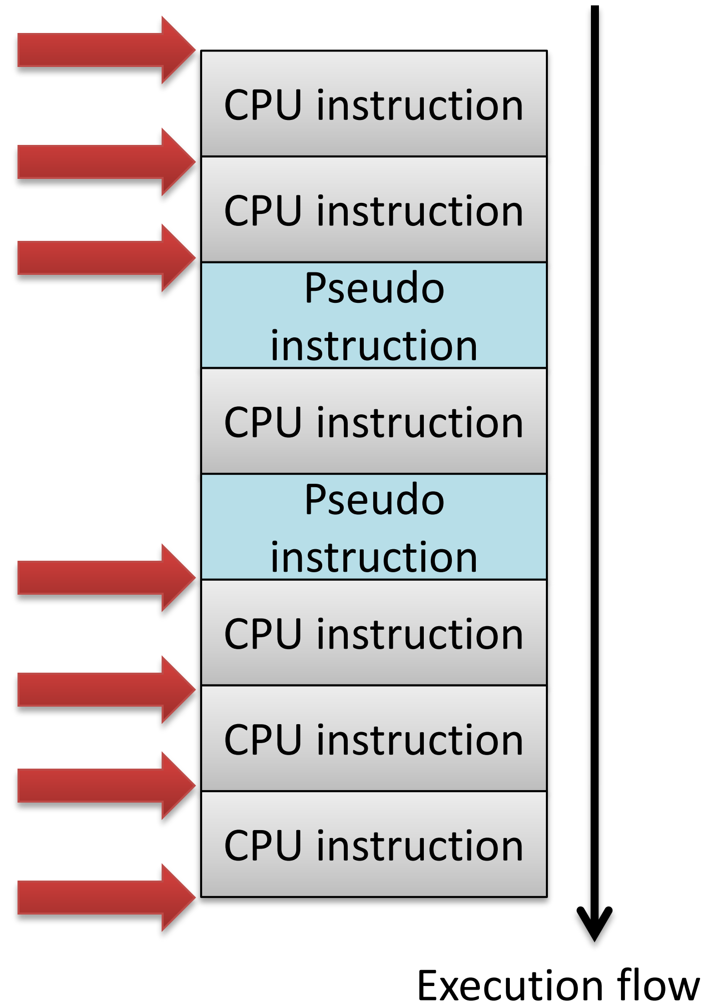
When can we allow strong update of shared memory?

- If types of shared memory are invariant before and after execution of a CPU instruction
  - Strong updates are allowed between 1 CPU instruction + pseudo instructions
    - Pseudo instructions = Instructions that affect types only and have no runtime effects



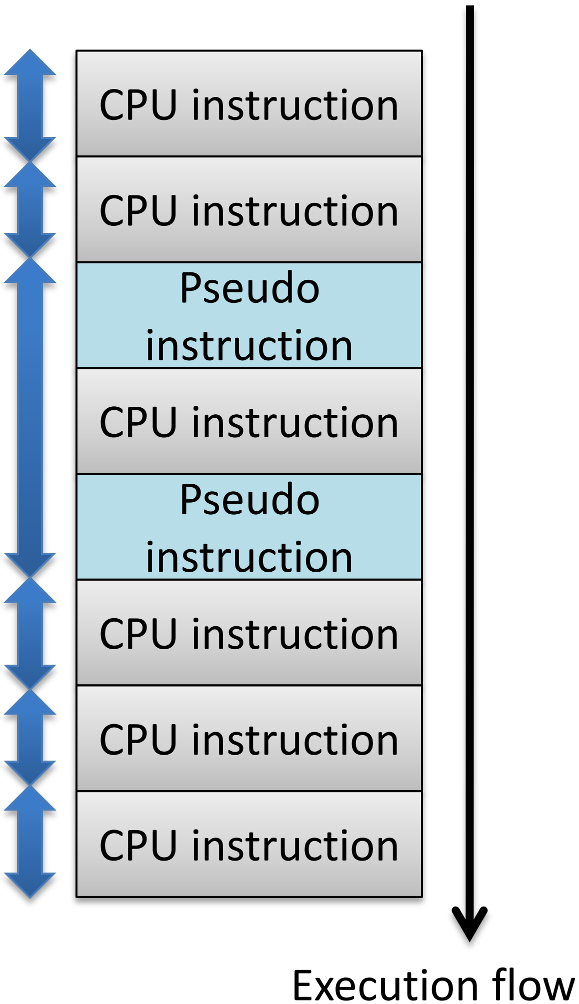
When do we allow strong update of shared memory?

- If types of shared memory are invariant before and after execution of a CPU instruction
  - Strong updates are allowed between 1 CPU instruction + pseudo instructions
    - Pseudo instructions = Instructions that affect types only and have no runtime effects



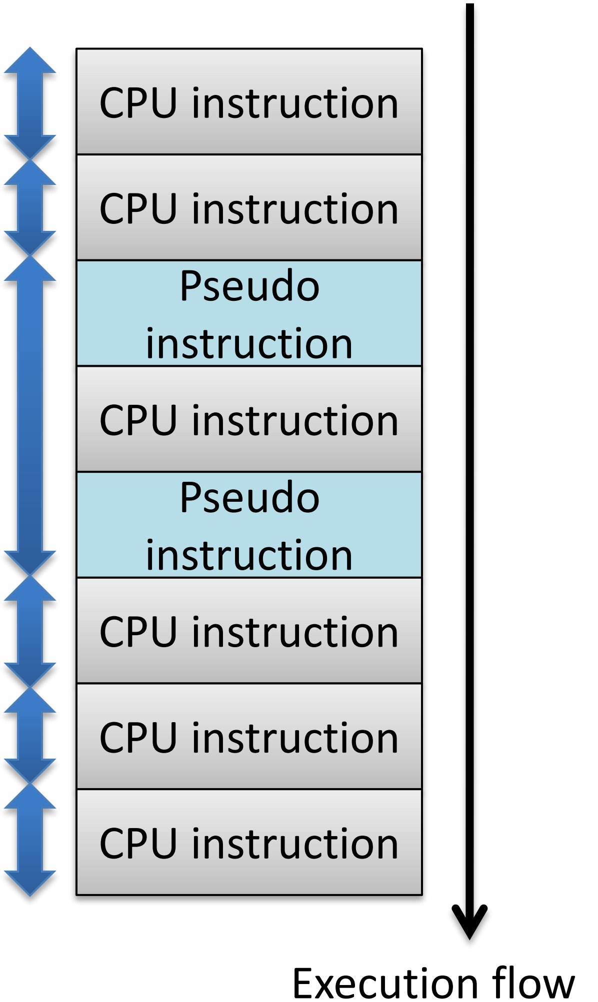
When do we allow strong update of shared memory?

- If types of shared memory are invariant before and after execution of a CPU instruction
  - Strong updates are allowed during 1 CPU instruction + pseudo instructions
    - Pseudo instructions = Instructions that affect types only and have no runtime effects



With this approach, types appear to be invariant from the viewpoint of other threads

- If types of shared memory are invariant before and after execution of a CPU instruction
  - Strong updates are allowed during 1 CPU instruction
    + pseudo instructions
    - Pseudo instructions
      = Instructions that affect types only and have no runtime effects

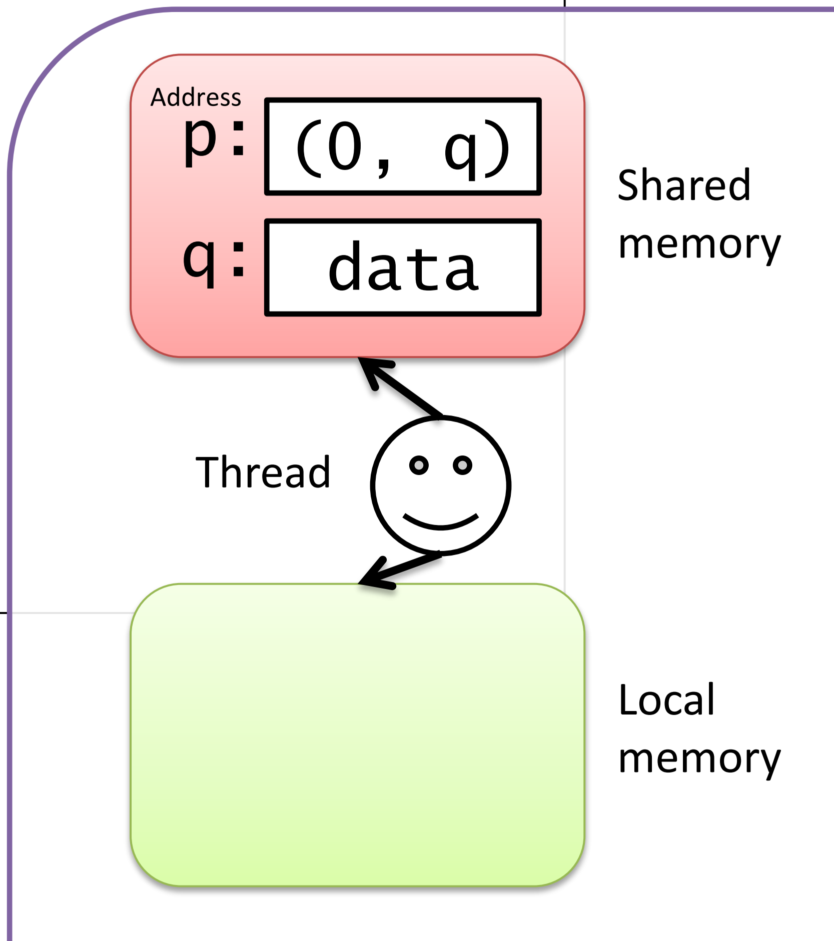


Example: lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```

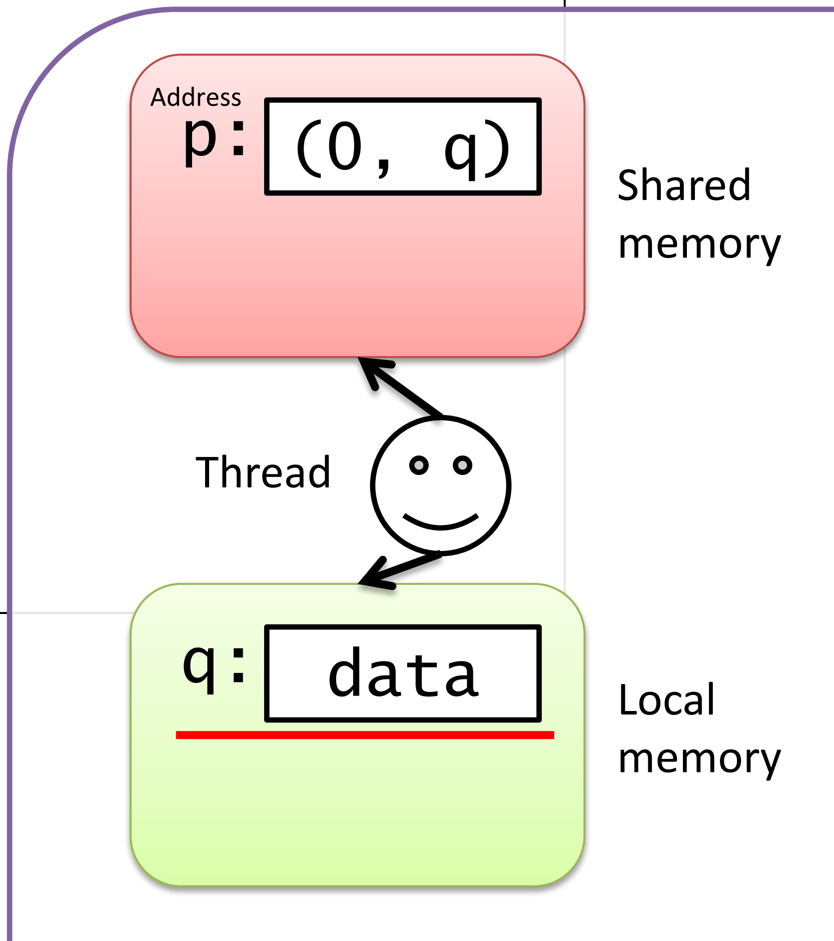
Example: lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```



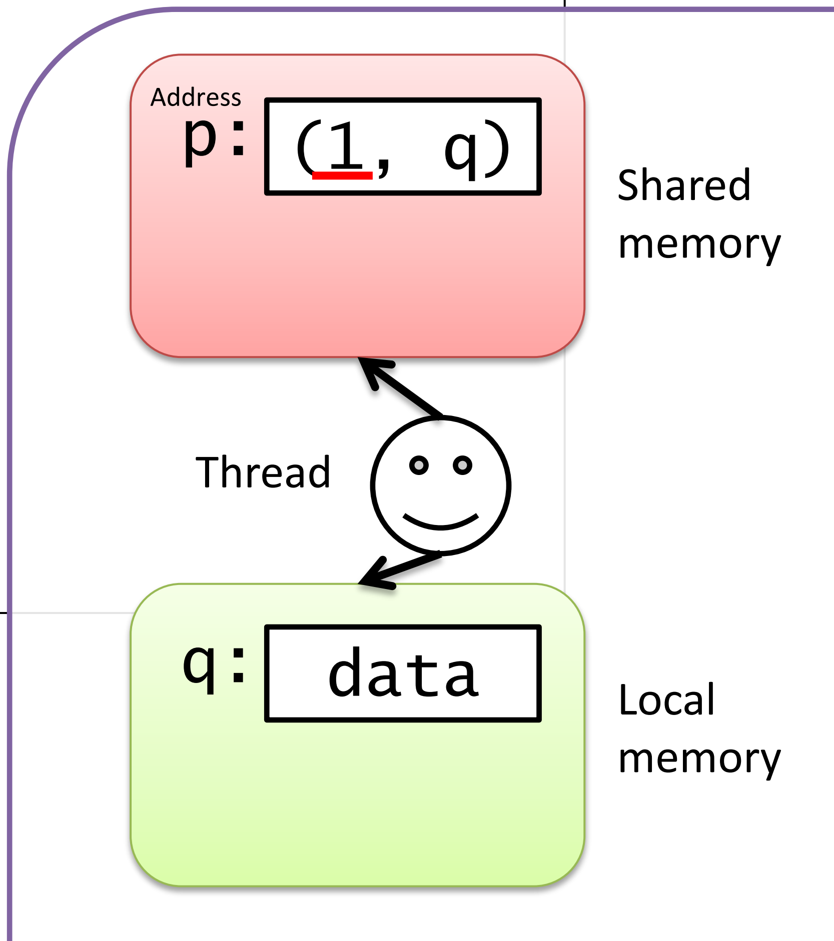
Example: lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
        mov r2 ← 1
        unpack r1
        xchg [r1], r2
        pack r1
        bne r2, 0, lock
        …
```



Example: lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```



Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```

State of the type checker

```
{p → ∃i.….(i, q)}
(r1 : p, r2 : ??)
```

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```

State of the type checker

```
{p → ∃i.….(i, q)}
(r1 : p, r2 : 1)
```

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
        mov r2 ← 1
        unpack r1
        xchg [r1], r2
        pack r1
        bne r2, 0, lock
        …
```

State of the type checker

```
{p → (i, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : 1)
```

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, l…
      …
```

State of the type checker

```
{p → (i, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : 1)
```

Strong update occurred:

The type has to be reverted before executing the CPU instruction after the next

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```

State of the type checker

```
{p → (1, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : i)
```

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
        mov r2 ← 1
        unpack r1
        xchg [r1], r2
        pack r1
        bne r2, 0, lock
        …
```

State of the type checker

```
{p → ∃i.….(i, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : i)
```

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```

State of the type checker

```
{p → ∃i.….(i, q)}
[q → data
     if [i == 0]]
(r1 : p, r2 : i)
```

The type is revered correctly

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```

State of the type checker

```
{p → ∃i.….(i, q)}
[q → data
     if [i == 0]]
(r1 : p, r2 : i)
```

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```

State of the type checker

```
{p → ∃i.….(i, q)}
[q → data]
(r1 : p, r2 : i)
```

Example: type-checking lock acquisition

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      xchg [r1], r2
      pack r1
      bne r2, 0, lock
      …
```

State of the type checker

```
{p → ∃i.….(i, q)}
[q → data]
(r1 : p, r2 : i)
```

Succeed in extracting memory region q protected by a lock

Example: type-checking lock release

```
{p→∃i.{q→data if [i==0]}. (i, q)}
[q→data]
(r1 : p)
unlock:
      unpack r1
      mov [r1] ← 0
      pack r1
      …
```

Example: type-checking lock release

```
{p→∃i.{q→data if [i==0]}. (i, q)}
[q→data]
(r1 : p)
unlock:
      unpack r1
      mov [r1] ← 0
      pack r1
      …
```

State of the type checker

```
{p → ∃i.….(i, q)}
[q → data]
(r1 : p)
```

Example: type-checking lock release

```
{p→∃i.{q→data if [i==0]}. (i, q)}
[q→data]
(r1 : p)
unlock:
        unpack r1
        mov [r1] ← 0
        pack r1
        …
```

State of the type checker

```
{p → (i, q)}
[q → data]
(r1 : p)
```

Example: type-checking lock release

```
{p→∃i.{q→data if [i==0]}. (i, q)}
[q→data]
(r1 : p)
unlock:
     unpack r1
     mov [r1] ← 0
     pack r1
     …
```

State of the type checker

```
{p → (i, q)}
[q → data]
(r1 : p)
```

Strong update occurred:

The type has to be reverted before executing the CPU instruction after the next

Example: type-checking lock release

```
{p→∃i.{q→data if [i==0]}. (i, q)}
[q→data]
(r1 : p)
unlock:
      unpack r1
      mov [r1] ← 0
      pack r1
      …
```

State of the type checker

```
{p → (0, q)}
[q → data]
(r1 : p)
```

Example: type-checking lock release

```
{p→∃i.{q→data if [i==0]}. (i, q)}
[q→data]
(r1 : p)
unlock:
      unpack r1
      mov [r1] ← 0
      pack r1
      …
```

State of the type checker

```
{p → ∃i.….(i, q)}
(r1 : p)
```

Example: type-checking lock release

```
{p→∃i.{q→data if [i==0]}. (i, q)}
[q→data]
(r1 : p)
unlock:
     unpack r1
     mov [r1] ← 0
     pack r1
     …
```

State of the type checker

```
{p → ∃i.….(i, q)}
(r1 : p)
```

The type is revered correctly and memory region q is successfully returned back to shared memory

About CPU interrupts

- CPU interrupts can be type-checked in a similar way
    - Interrupt handlers and interrupted programs can be viewed as concurrently running threads
  - Strong update is basically not allowed to shared memory between interrupters/interruptees
    - If interrupts are disabled using CPU's interrupt flag, strong updates are allowed on the shared memory

One limitation of our approach explained so far

- Relaxed memory models of today’s CPU are not considered
  - Shared memory of relaxed memory consistency may violate memory safety property

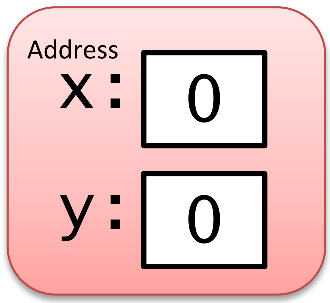
What is relaxed memory consistency?

- In short,
  memory consistency models that allow
  effects of memory operations on one CPU
  to be observed in a different order
  from other CPUs

Example of relaxed memory consistency

- Execution of the following 2 threads can yield the result "`r1 = 0` and `r2 = 0`"

Initial state of shared memory

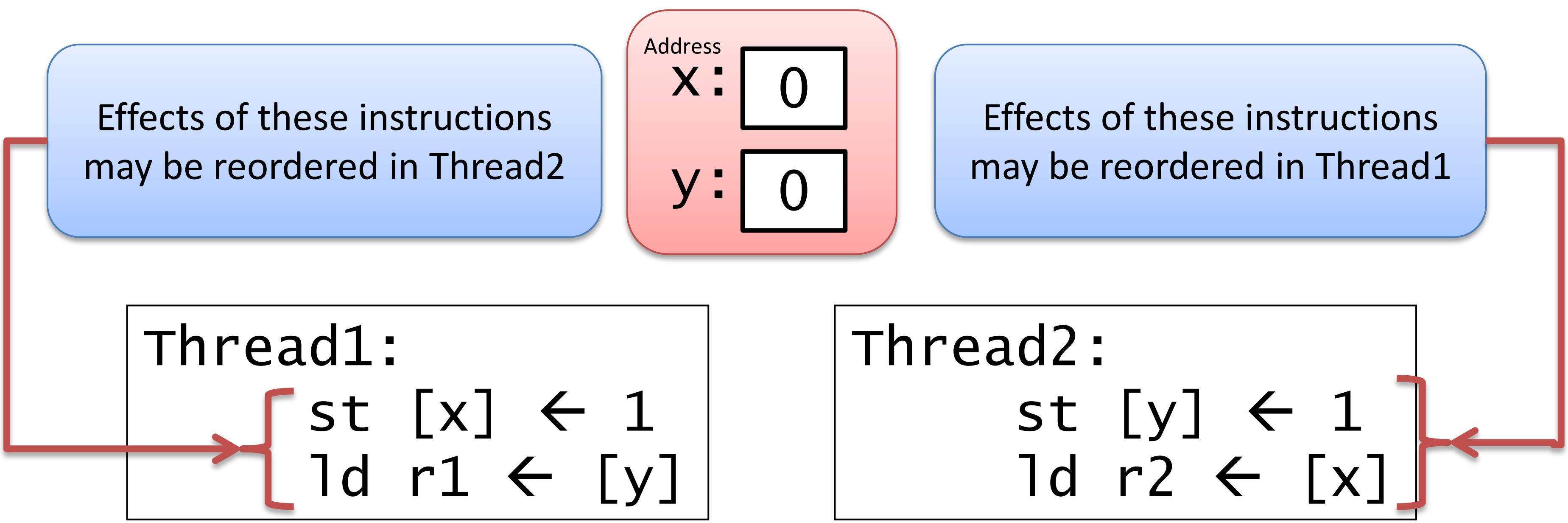



```
Thread1:
        st [x] ← 1
        ld r1 ← [y]
```

```
Thread2:
        st [y] ← 1
        ld r2 ← [x]
```

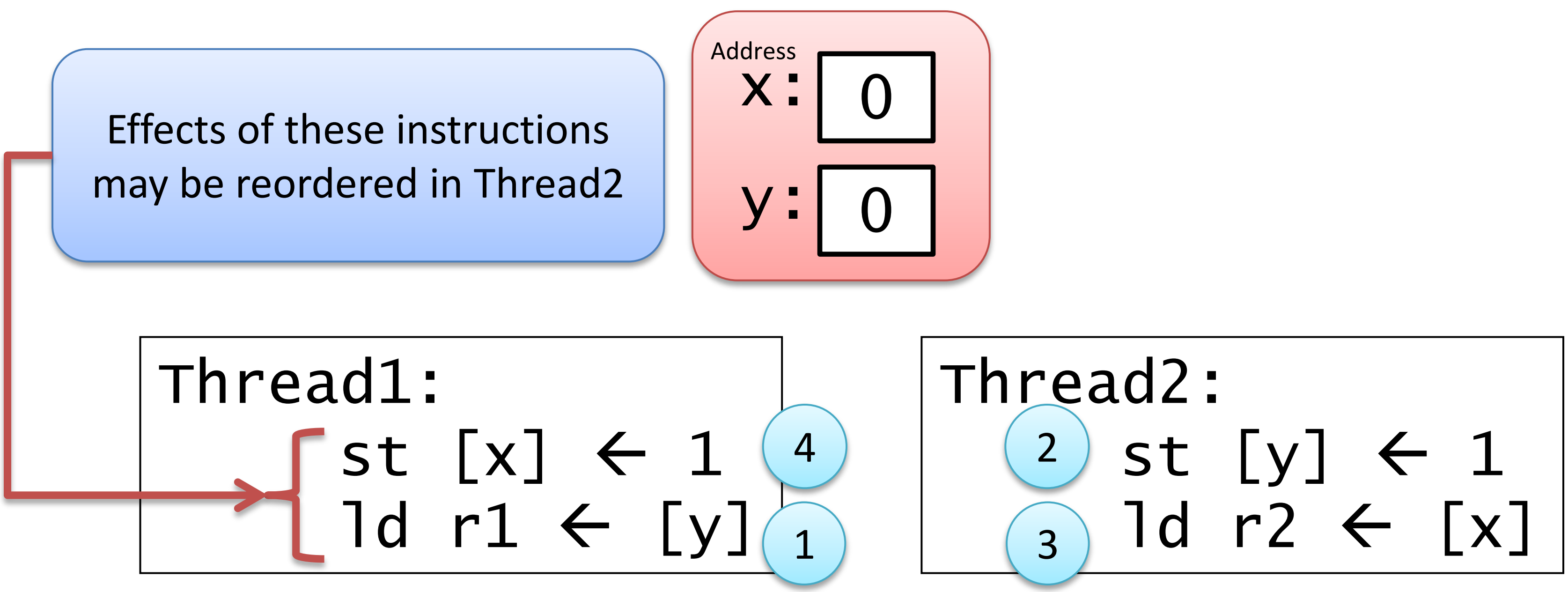
Example of relaxed memory consistency

- Execution of the following 2 threads can yield the result "r1 = 0 and r2 = 0"



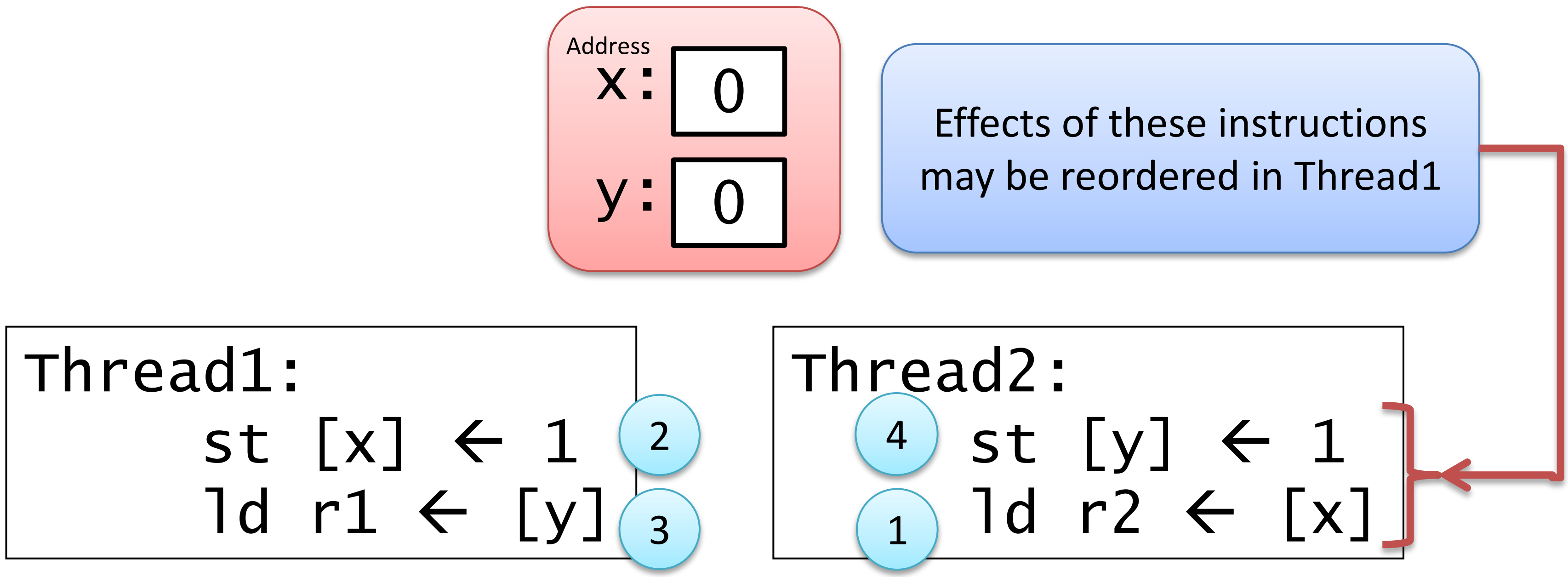
Example of relaxed memory consistency

- Execution of the following 2 threads can yield the result "`r1 = 0` and `r2 = 0`"



Example of relaxed memory consistency

- Execution of the following 2 threads can yield the result "r1 = 0 and r2 = 0"



How to control memory reordering in relaxed memory consistency models?

- Typically, utilize two mechanisms provided by today’s CPUs
  - Atomic memory operation mechanism
    - E.g., “lock” prefix on Intel Architecture
  - Memory ordering control mechanism
    - E.g., acquire/release

Atomic memory operation

- Memory operation whose effect is observed in an “all-or-nothing” way by other threads

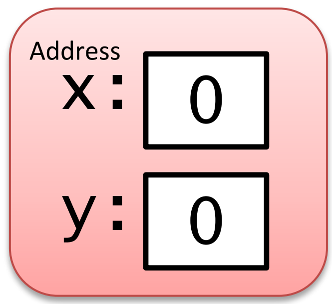
Memory ordering control

- Acquire operation
  - Operation whose effect becomes observable from other threads before any succeeding operation
- Release operation
  - Operation whose effect becomes observable from other threads after any preceding operation

Example of memory ordering control

- Execution of the following 2 threads never yields the result “`r1 = 0` and `r2 = 0`”

Initial state of shared memory

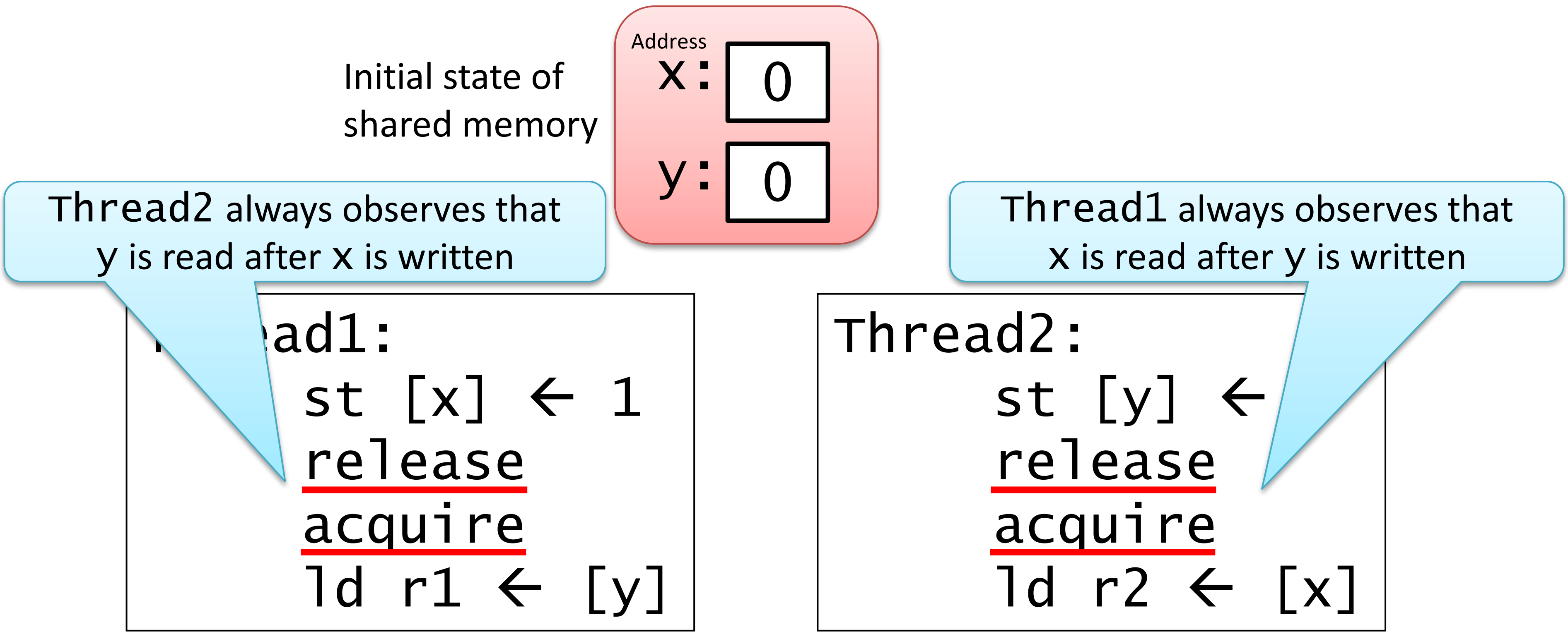



```
Thread1:
        st [x] ← 1
        release
        acquire
        ld r1 ← [y]
```

```
Thread2:
        st [y] ← 1
        release
        acquire
        ld r2 ← [x]
```

Example of memory ordering control

- Execution of the following 2 threads never yields the result "`r1 = 0` and `r2 = 0`"



Our type-checking approach in order to support relaxed memory consistency (just an idea)

- Check the following 2 constraints with type system
  - Only atomic memory operations are able to perform strong update on shared memory
  - Memory ordering control mechanisms are used properly when moving memory regions between shared memory and local memory
    - Shared memory → local memory: use `acquire`
    - Local memory → shared memory: use `release`

Example of lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
     mov r2 ← 1
     unpack r1
     atomic xchg [r1], r2
     acquire
     pack r1
     bne r2, 0, lock
     …
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      atomic xchg [r1], r2
      acquire
      pack r1
      bne r2, 0, lock
      …
```

State of the type-checker

```
{p → ∃i.….(i, q)}
(r1 : p, r2 : ??)
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      atomic xchg [r1], r2
      acquire
      pack r1
      bne r2, 0, lock
      …
```

State of the type-checker

```
{p → ∃i.….(i, q)}
(r1 : p, r2 : 1)
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
    mov r2 ← 1
    unpack r1
    atomic xchg [r1], r2
    acquire
    pack r1
    bne r2, 0, lock
    …
```

State of the type-checker

```
{p → (i, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : 1)
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
     mov r2 ← 1
     unpack r1
     atomic xchg [r1], r2
     acquire
     pack r1
     bne r2, 0, lock
     …
```

State of the type-checker

```
{p → (i, q)}
[q → data
      if [i == 0]]
1 : p, r2 : 1)
```

Memory region q is still not accessible because `acquire` is not performed

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      atomic xchg [r1], r2
      acquire
      pack r1
      bne r2, 0, lock
      …
```

State of the type-checker

```
{p → (1, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : i)
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      atomic xchg [r1], r2
      acquire
         ck r1
        ne r2, 0, lock
```

State of the type-checker

```
{p → (1, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : i)
```

This memory operation on a shared memory region is OK because it is atomic

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
     mov r2 ← 1
     unpack r1
     atomic xchg [r1], r2
     acquire
     pack r1
     bne r2, 0, lock
     …
```

State of the type-checker

```
{p → (1, q)}
[q → data
       if [i == 0]]
(r1 : p, r2 : i)
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
     mov r2 ← 1
     unpack r1
     atomic xchg [r1], r2
     acquire
     pack r1
     bne r2, 0, lock
     …
```

State of the type-checker

```
{p → (1, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : i)
```

Memory region q now becomes accessible because `acquire` is performed

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
    mov r2 ← 1
    unpack r1
    atomic xchg [r1], r2
    acquire
    pack r1
    bne r2, 0, lock
    …
```

State of the type-checker

```
{p → ∃i.….(i, q)}
[q → data
      if [i == 0]]
(r1 : p, r2 : i)
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
      mov r2 ← 1
      unpack r1
      atomic xchg [r1], r2
      acquire
      pack r1
      bne r2, 0, lock
      …
```

State of the type-checker

```
{p → ∃i.….(i, q)}
[q → data
     if [i == 0]]
(r1 : p, r2 : i)
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
     mov r2 ← 1
     unpack r1
     atomic xchg [r1], r2
     acquire
     pack r1
     bne r2, 0, lock
     …
```

State of the type-checker

```
{p → ∃i.….(i, q)}
[q → data]
(r1 : p, r2 : i)
```

Example of type-checking lock acquisition in a relaxed memory consistency model

```
{p→∃i.{q→data if [i==0]}. (i, q)}
(r1 : p)
lock:
     mov r2 ← 1
     unpack r1
     atomic xchg [r1], r2
     acquire
     pack r1
     bne r2, 0, lock
     …
```

State of the type-checker

```
{p → ∃i.….(i, q)}
[q → data]
(r1 : p, r2 : i)
```

Succeed in extracting memory region q protected by a lock

Related work (1/3)

- Type-based approaches
  - A multithreaded typed assembly language [Vasconcelos et al. 2006]
    - It cannot be used to implement synchronization primitives and multi-thread management mechanisms themselves
      - Mutex locks and threading mechanisms are provided as language primitives
  - Type-based analysis of synchronization lock usage [Flanagan et al. 1999, 2007, Iwama et al. 2002, Grossman 2003, etc.]
    - They cannot be used to analyze synchronization primitives and multi-thread management mechanisms themselves
      - Their goals are to ensure race/deadlock- freedom
        - » whereas our goal is limited to ensuring simple type safety

Related work (2/3)

- Separation logic approaches
  - Abstract Interrupt Machine (AIM)
    [Feng et al. 2008]
    - Utilizing separation logic
      in order to verify programs with CPU interrupts
      by maintaining invariants on interrupters/interruptees
      - SMP/multi-core environments are not considered
  - Concurrent Abstract Predicates
    [Dinsdale-Young et al. 2010]
    - Utilizing separation logic in order to handle invariants
      on shared memory between multiple threads
      - Relaxed memory consistency models are not considered

Related work (3/3)

- Program verification
  for relaxed memory consistency models
  - Sober[Burckhardt et al. 2008]
    - A bounded model checker that checks
      whether a program on TSO satisfies SC
  - Boudol et al. 2009, Atig et al. 2010, etc.
    - Define semantics of relaxed memory models
      in operational-semantics styles for program verification

Conclusion and future work

- We presented Typed Assembly Language for SMP/multi-core environments with CPU interrupts
  - Memory and control-flow safety can be verified
  - Sync primitives can be directly written in it
    - We also showed an idea of how to support relaxed memory consistency models
- Future work :
  - Prove the soundness of our type system, particularly for the extension of relaxed memory models
  - Implement an OS kernel with our TAL
  - Extend the type system further in order to support more complex and efficient synchronization primitives