Write once, herd everywhere

Boqun Feng (Microsoft)

Agenda

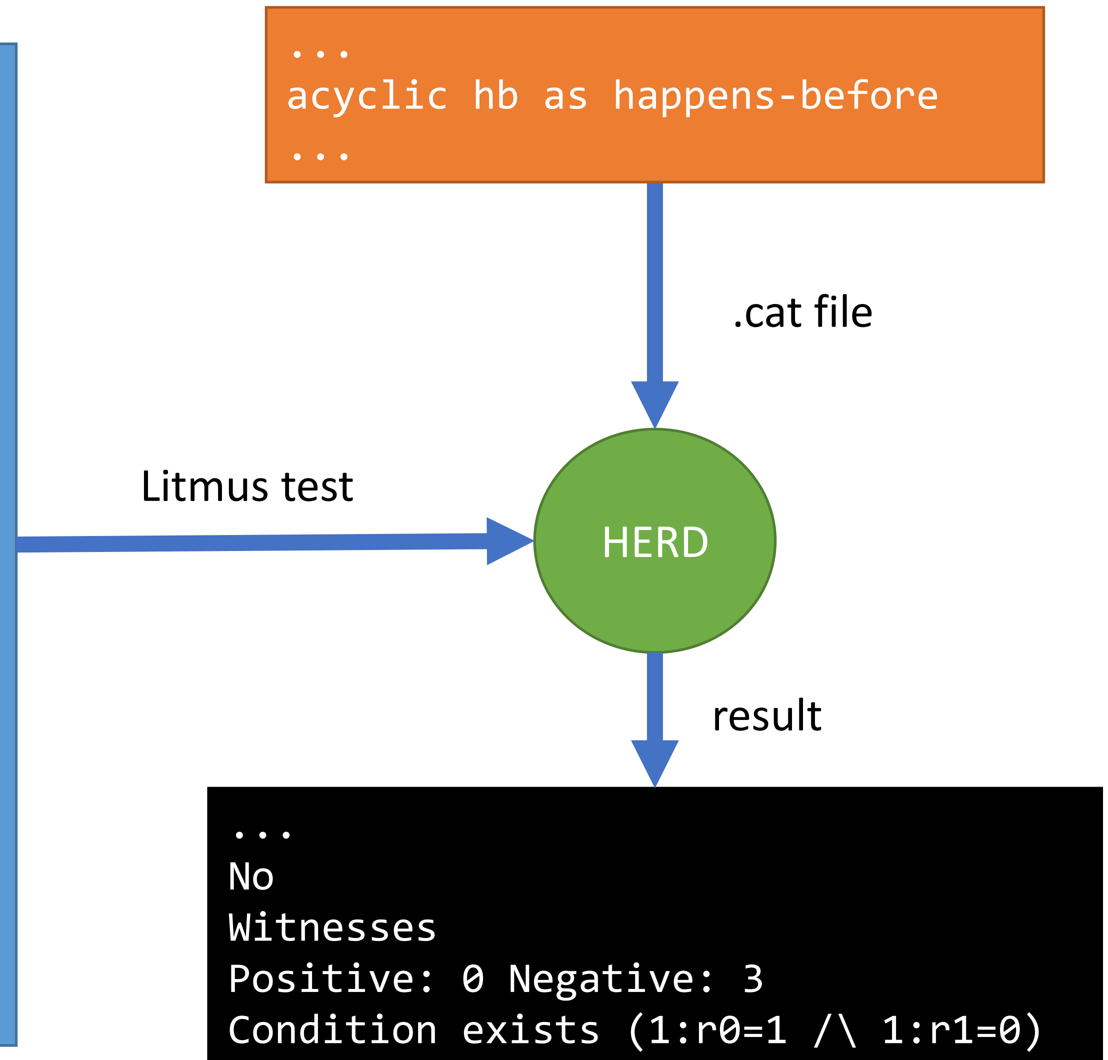
- LKMM and Herdtools
- Litmus translation
- Current status and future work

LKMM and herdtools

- The core of LKMM is linux-kernel.cat file which defines the model in the “cat” language of herdtools.
- Herdtools: A memory model simulator.
  - Users may write simple, single events, axiomatic models of their own
    - Using cat file, e.g. linux-kernel.cat
  - and run litmus tests on top of their model.
    - Like tools/memory-model/litmuts

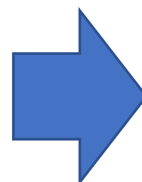
How herd works

```
1:      C MP+pooncerelease+poacquireonce
2:      { x = 0; y = 0 }
3:
4:      P0(int *x, int *y)
5:      {
6:              WRITE_ONCE(*x, 1);
7:              smp_store_release(y, 1);
8:      }
9:
10:     P1(int *x, int *y)
11:     {
12:             int r0;
13:             int r1;
14:
15:             r0 = smp_load_acquire(y);
16:             r1 = READ_ONCE(*x);
17:     }
18:
19:     exists (1:r0=1 /\ 1:r1=0)
```



How herd works: step 1, generate “events”

```
1:      C MP+pooncerelease+poacquireonce
2:      { x = 0; y = 0 }
3:
4:      P0(int *x, int *y)
5:      {
6:              WRITE_ONCE(*x, 1);
7:              smp_store_release(y, 1);
8:      }
9:
10:     P1(int *x, int *y)
11:     {
12:             int r0;
13:             int r1;
14:
15:             r0 = smp_load_acquire(y);
16:             r1 = READ_ONCE(*x);
17:     }
18:
19:     exists (1:r0=1 /\ 1:r1=0)
```



w[] x 0

w[] y 0

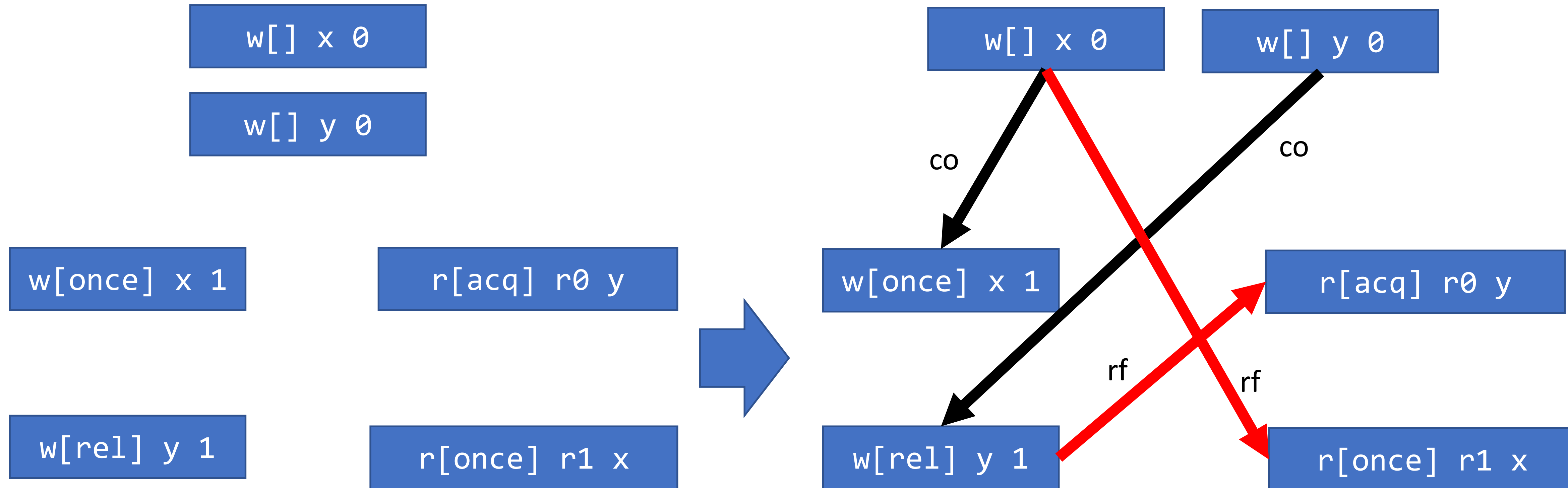
w[once] x 1

r[acq] r0 y

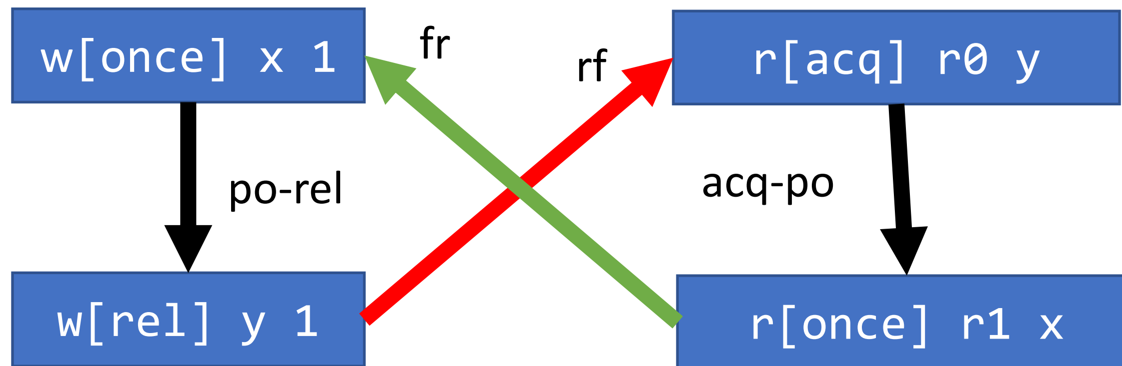
w[rel] y 1

r[once] r1 x

How herd works: step 2, enumerate “com”



How herd works: step 3, check “cycles”



- acyclic hb as happen-before
- prop & int is hb
  - fre ; cumul-fence; rfe is prop
    - po-rel is cumul-fence
- ppo is hb
  - fence is ppo
    - acq-po is fence
- (fre; po-rel; rfe) & int is hb
- acq-po is hb

- hb* form cycle in this execution candidate

C Litmus tests

- tools/memory-model/litmus-tests/*
- Documentation/litmus-tests/*
- https://github.com/paulmckrcu/litmus

Asm litmus tests

```
AArch64 MP+pooncerelease+poacquireonce

{1:X0=y; 1:X2=x; 0:X1=y; 0:X0=x;}

 P0                   | P1                   ;
 MOV X2,#1            | lbl3: LDAR X1,[X0] ;
 lbl1: STR X2,[X0]    | lbl4: LDR X3,[X2]  ;
 MOV X2,#1            |                      ;
 lbl2: STLR X2,[X1]   |                      ;

exists (1:X1=1 /\ 1:X3=0)
```

```
...
No
Witnesses
Positive: 0 Negative: 3
Condition exists (1:r0=1 /\ 1:r1=0)
```

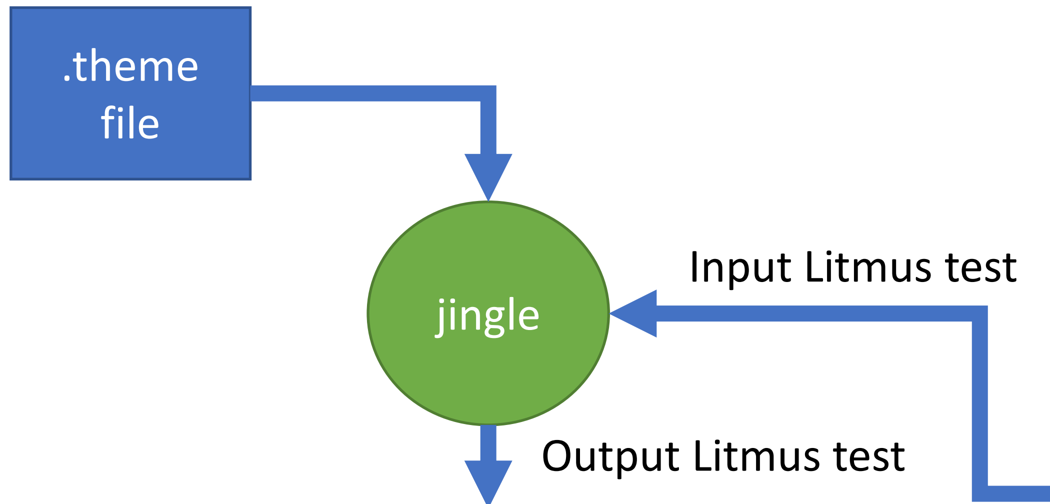
LKMM and herdtools

- With the .cat file of LKMM, developers can use C litmus to understand the model provided by Linux kernel and the semantics of the modeled synchronized primitives (*_ONCE(), smp_*_{store,release}, atomic APIs, etc).
- But how can we know the primitives are implemented correctly?

Translate Litmus tests from C to asm

- Get more litmus tests for free ;-)
- Verify the Linux Kernel Model by comparing the results.
- Tools:
  - jingle and gen_theme

Translate litmus tests using jingle




```
1:      C MP+pooncerelease+poacquireonce
2:      { x = 0; y = 0 }
3:
4:      P0(int *x, int *y)
5:      {
6:              WRITE_ONCE(*x, 1);
7:              smp_store_release(y, 1);
8:      }
9:
10:     P1(int *x, int *y)
11:     {
12:             int r0;
13:             int r1;
14:
15:             r0 = smp_load_acquire(y);
16:             r1 = READ_ONCE(*x);
17:     }
18:
19:     exists (1:r0=1 /\ 1:r1=0)
```

```
AArch64 MP+pooncerelease+poacquireonce

{1:X0=y; 1:X2=x; 0:X1=y; 0:X0=x;}

 P0                   | P1                 ;
 MOV X2,#1            | lbl3: LDAR X1,[X0] ;
 lbl1: STR X2,[X0]    | lbl4: LDR X3,[X2]  ;
 MOV X2,#1            |                    ;
 lbl2: STLR X2,[X1]   |                    ;

exists (1:X1=1 /\ 1:X3=0)
```

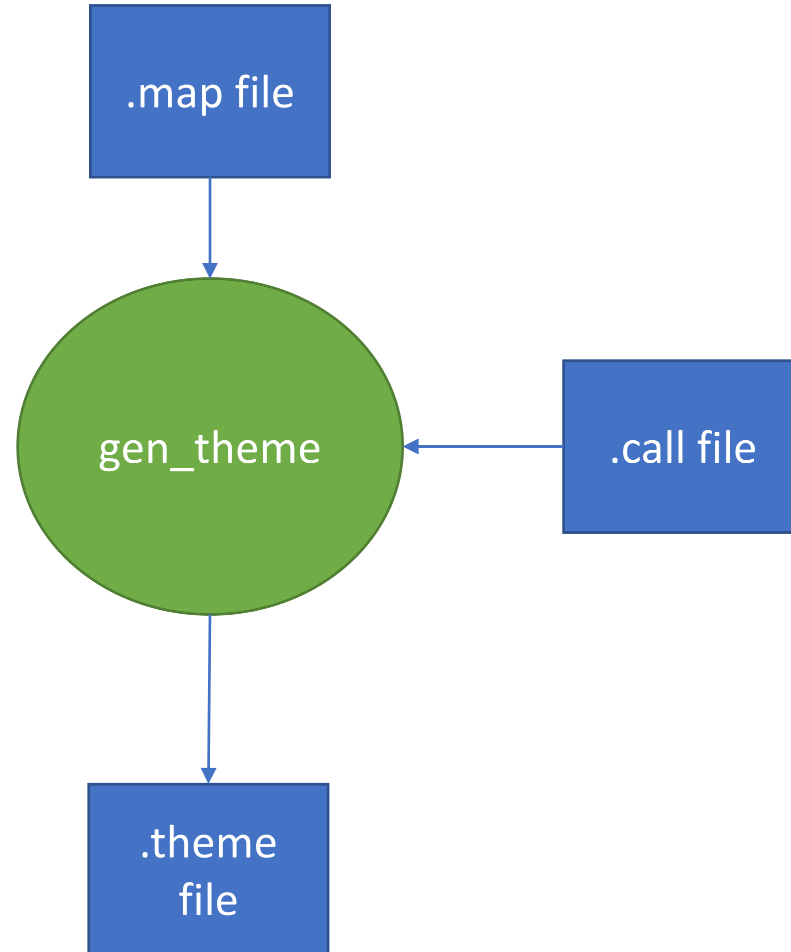
An example of theme file

```
...
"%x = READ_ONCE(*%y);" -> "load:LDR %x,[%y]"
"WRITE_ONCE(*%y, constvar:c);" -> "MOV %tmp,&c;
                                   STR %tmp,[%y]"
...
"%x = smp_load_acquire(%y);" -> "load:LDAR %x,[%y]"
"smp_store_release(%y, constvar:c);" -> "MOV %tmp,&c;
                                         STLR %tmp,[%y]"
```

Ideal approach

- Theme files are maintained by arch maintainers
- Translate and check every time when
  - LKMM changed (adding new api, changing api semantics)
  - Implementation changed (include adding new architecture support)
- But rules (described in .theme file) of translation might be a lot

Generate .theme files



.call file is arch-independent

- Similar as include/atomic.h

```
...
"%x = smp_load_acquire(%y);" -> "@acquire %x = READ_ONCE(*%y);“
"smp_store_release(%y, %x);" -> "@release WRITE_ONCE(*%y, %x);“
"%x = rcu_dereference(*%y);" -> "@id %x = READ_ONCE(*%y);”
"%r = xchg(%x, constvar:c);" -> "@full %r = xchg(%x, constvar:c);“
...
```

.map file is per arch

- Similar as asm/atomic.h

```
...
"%x = READ_ONCE(*%y);" -> "load:LDR %x,[%y]"
"WRITE_ONCE(*%y, constvar:c);" -> "MOV %tmp,&c;
                                    store:STR %tmp,[%y]“
...
"release" : "store:STR" -> "store:STLR"
"release" : "store:STXR" -> "store:STLXR"
"release" : "" -> "DMB ISH;“
"full" : "acquire | release | full_on_acq_rel“

...
```

Current Status

- Support Linux2ARM64 and Linux2PPC translation
- Atomic APIs are partially supported
- RCU APIs are not supported
- Spinlocks are translated as simple spinlock implemention
  - Herd check results may vary between C version and asm version.

Future work

- Support translation for more APIs
- Propose the .call and .map files to Linux mainline
- Try another approach if the previous doesn’t work

Demos

- Thanks!