

Introduction of passive reference

Kengo Nakahara

(knakahara@n.o / k-nakahara@iij.ad.jp)

Internet Initiative Japan Inc.

AsiaBSDCon 2016 ~~NetBSD~~ BoF

March 11 2016

Table of contents

- Overview
- Motivation
- Concept
- Design
- Operation overview
- Summary

Overview

- passive reference (psref) is a synchronization mechanism like reference counting
 - using `pserialize(9)` (*)
 - `pserialize(9)` is one of lockless synchronization mechanisms like RCU of Linux
 - to scale better than reference counting
- psref is under development for NetBSD kernel
 - e.g. packet processing paths
- psref is
 - discussed by `riastradh@n.o`, `rmind@n.o`, and `dyoung@n.o`
 - reviewed by `riz@n.o`
 - being implemented by `riastradh@n.o`
 - <http://mail-index.netbsd.org/tech-net/2016/01/24/msg005507.html>
 - and update <http://mail-index.netbsd.org/tech-net/2016/02/15/msg005621.html>
 - Thanks!
- I am using psref for making gif(4) MP-ify

(*) see “Modernizing NetBSD Networking Facilities and Interrupt Handling” in AsiaBSDCon2015

Motivation

- Network packet processing needs to share resources
 - route, tunnel configuration, and so on
- pserialize(9) can achieve good scalability
 - we verified it with bridge(4) (*)
 - contract: **sleep is prohibited** in reader critical section
- Packet processing **may sleep** even in fast paths
 - e.g. adaptive mutex, rwlock, and rtalloc1
- How to resolve with good scalability?
 - Changing all of these processings not sleeping in that section is hard work because of complex interdependency
 - Reference count decrease scalability because of interprocessor synchronization

(*) see “Modernizing NetBSD Networking Facilities and Interrupt Handling” in AsiaBSDCon2015, too

Concept

- Somehow hold a reference to shared resource without interprocessor synchronization
 - like OpenBSD's SRP(9) (I think. I don't know detail)
- If sleepable processing doesn't migrate between CPUs, interprocessor synchronization is not needed
 - Except for destruction, discuss later
- softint(9) and CPU-bound kthread satisfy this assumption
 - If not satisfied, we have to prevent it somehow during using psref

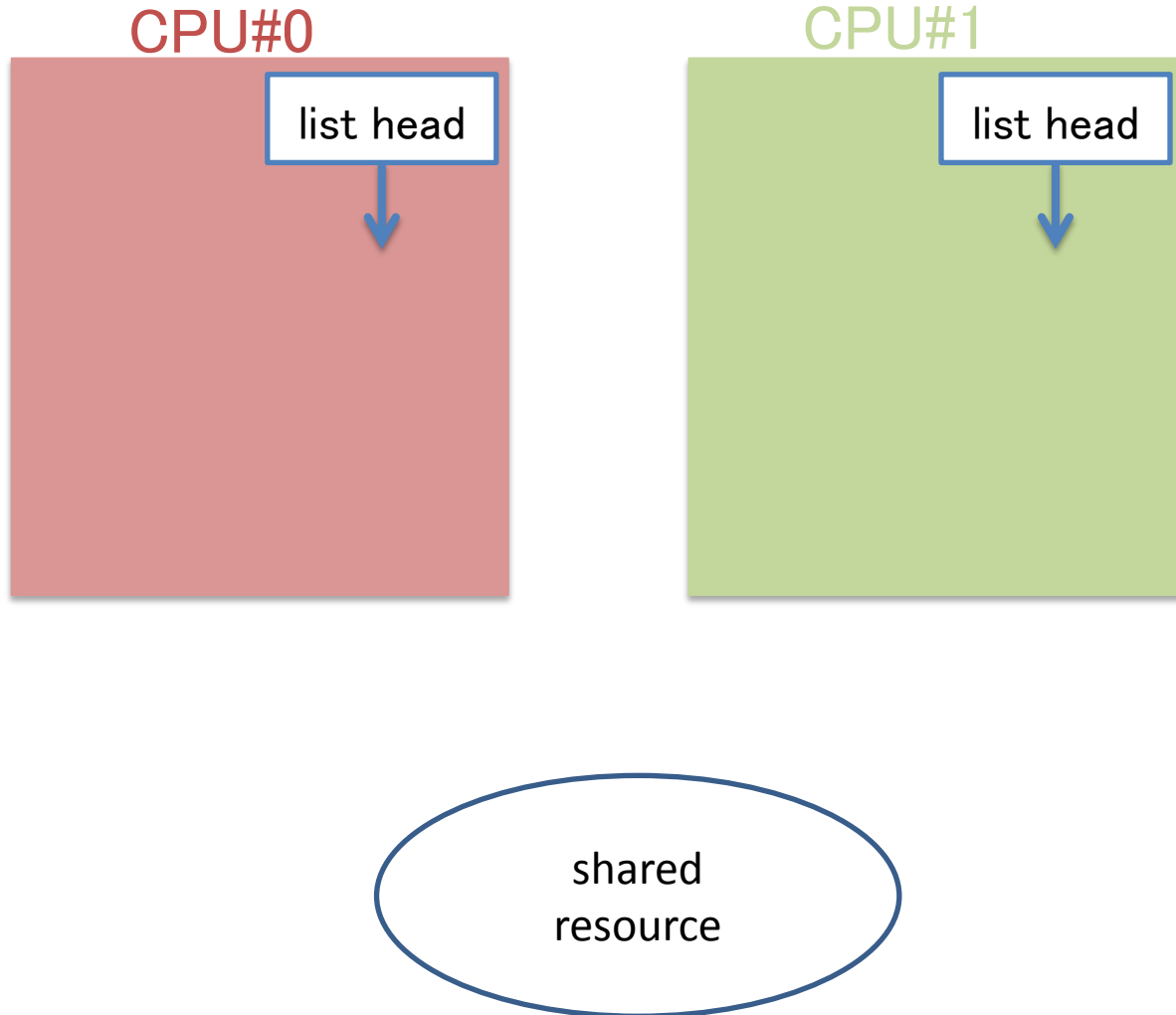
Design

- Add a CPU-local list to each CPU to manage references
- Read side (fast path)
 - Acquiring a reference is represented as adding an entry to the list of the current CPU
 - pserialize(9) guarantees the entry isn't destroyed during the operation
 - Releasing a reference is represented as removing the entry from the list
- Write side (slow path)
 - Before destroying a shared resource, wait for **ALL** CPUs to release their references to it
 - i.e. IPI (*) broadcast is needed

(*) Inter-Processor Interrupt, also called “cross call”, “xcall”

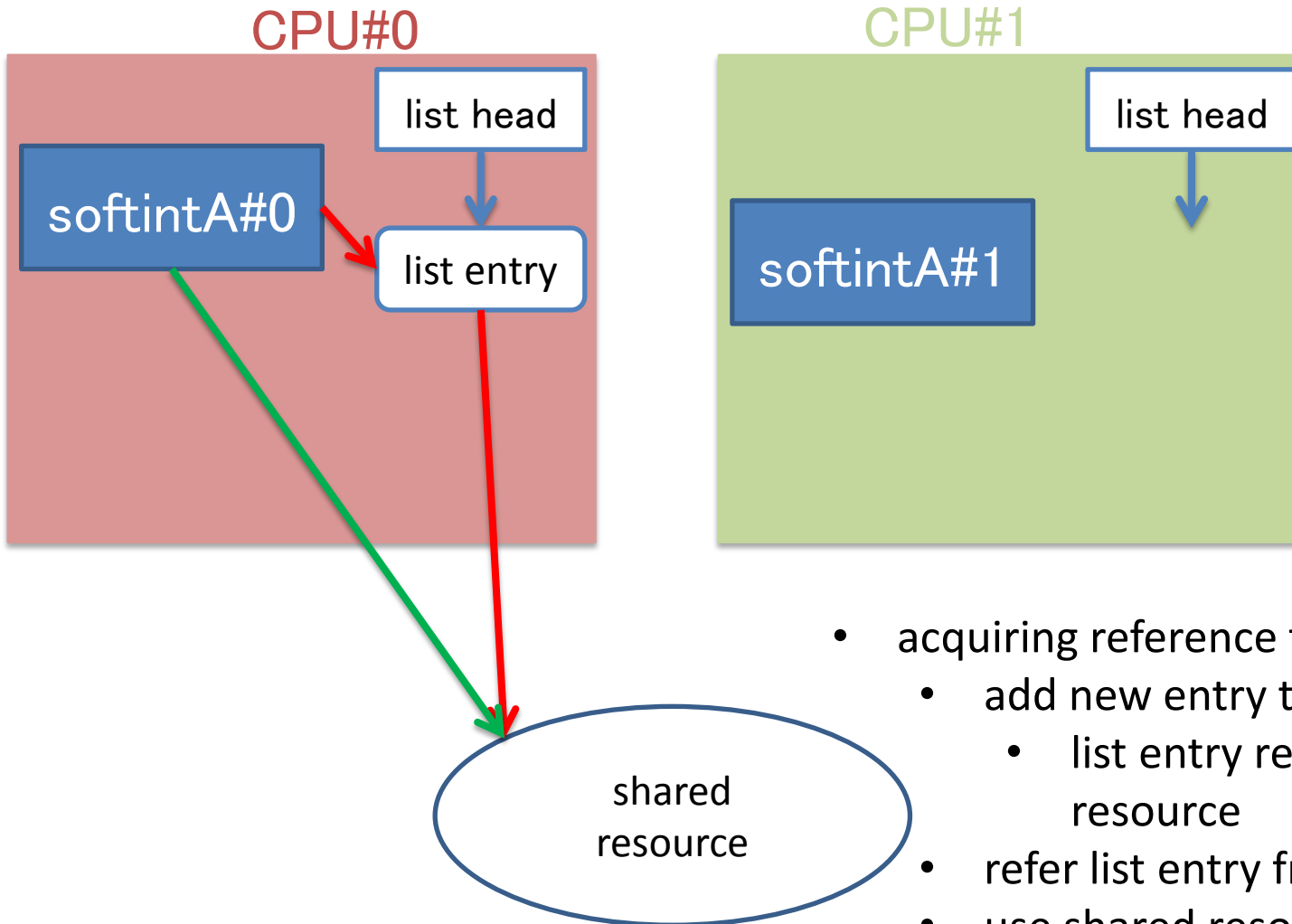
Operation overview (1/7)

initial state



Operation overview (2/7)

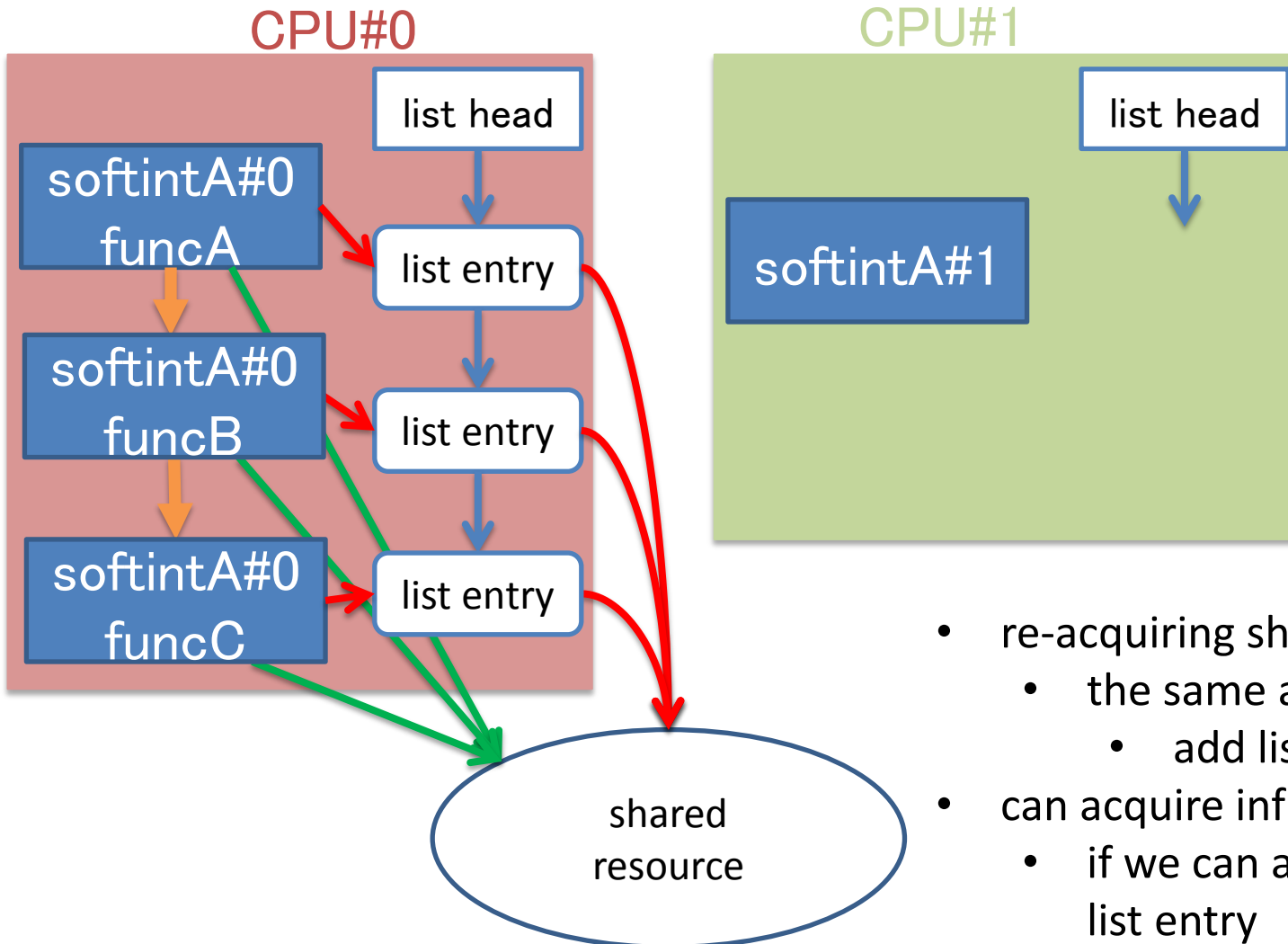
acquiring reference by CPU#0



- acquiring reference to shared resource
 - add new entry to CPU local list
 - list entry refer to shared resource
 - refer list entry from call stack
 - use shared resource

Operation overview (2-b/7)

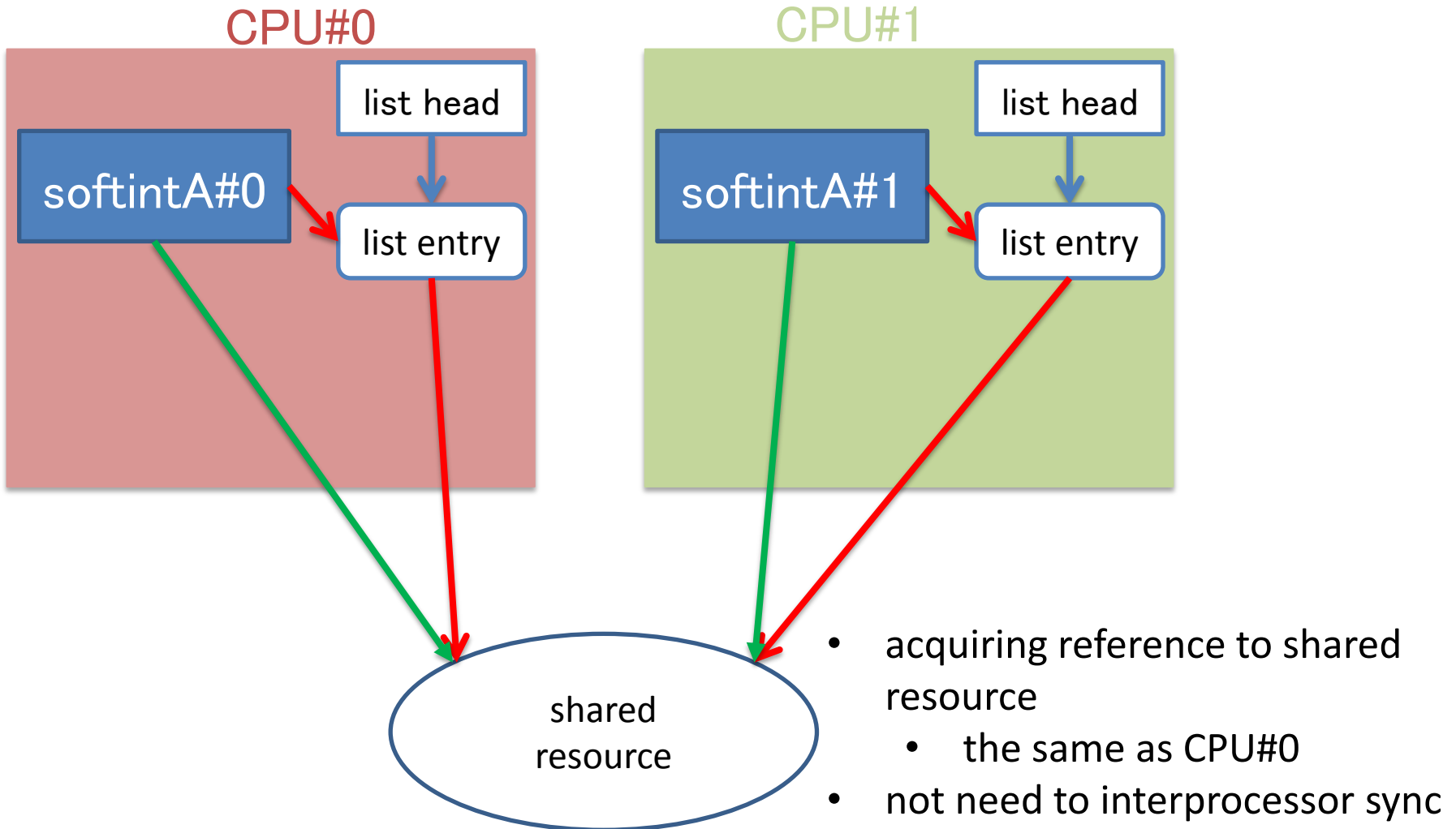
re-acquiring reference by CPU#0



- re-acquiring shared resource
 - the same as fist time
 - add list entry, and...
- can acquire infinitely
 - if we can allocate memory for list entry

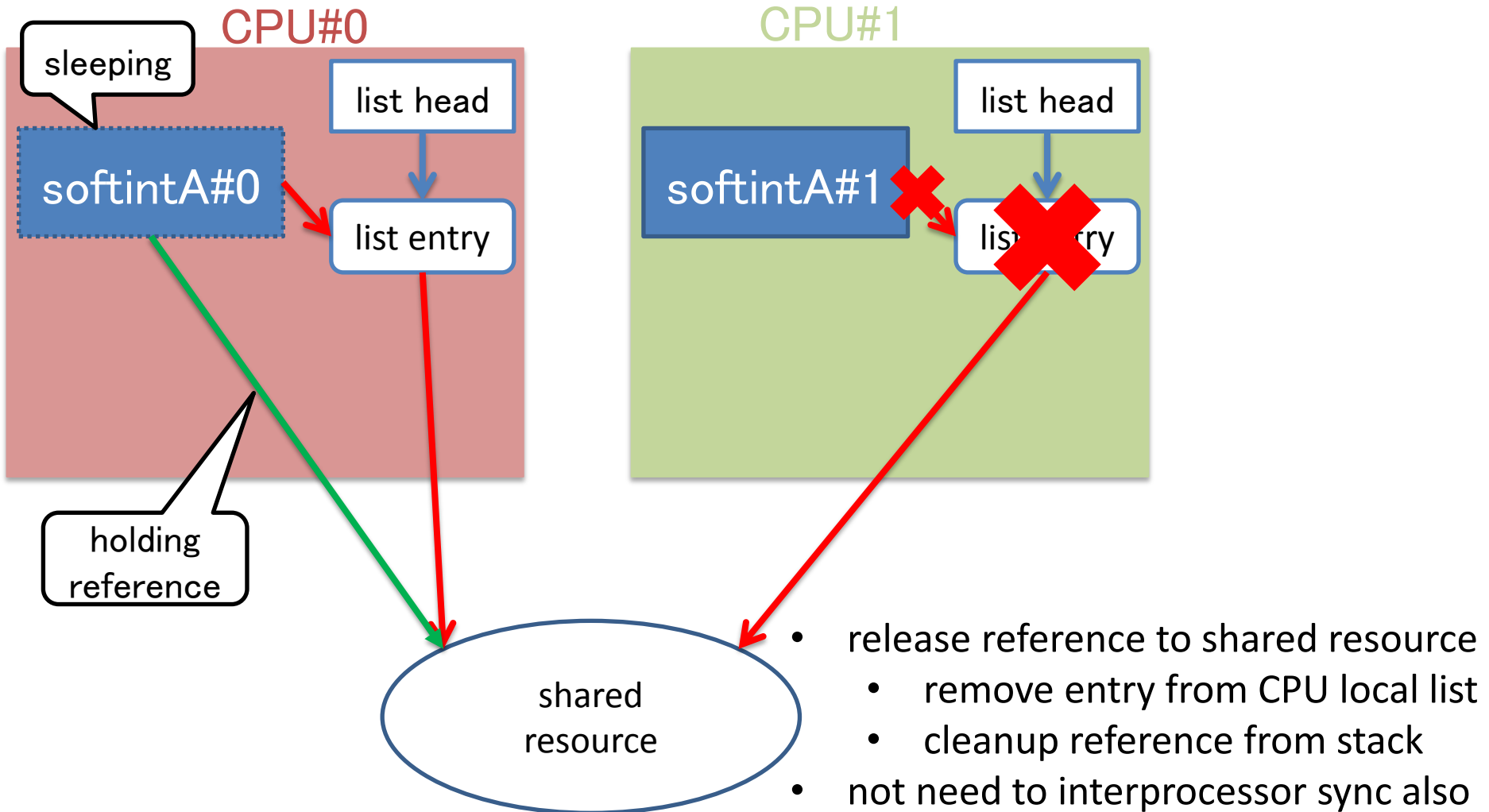
Operation overview (3/7)

acquiring reference by CPU#1



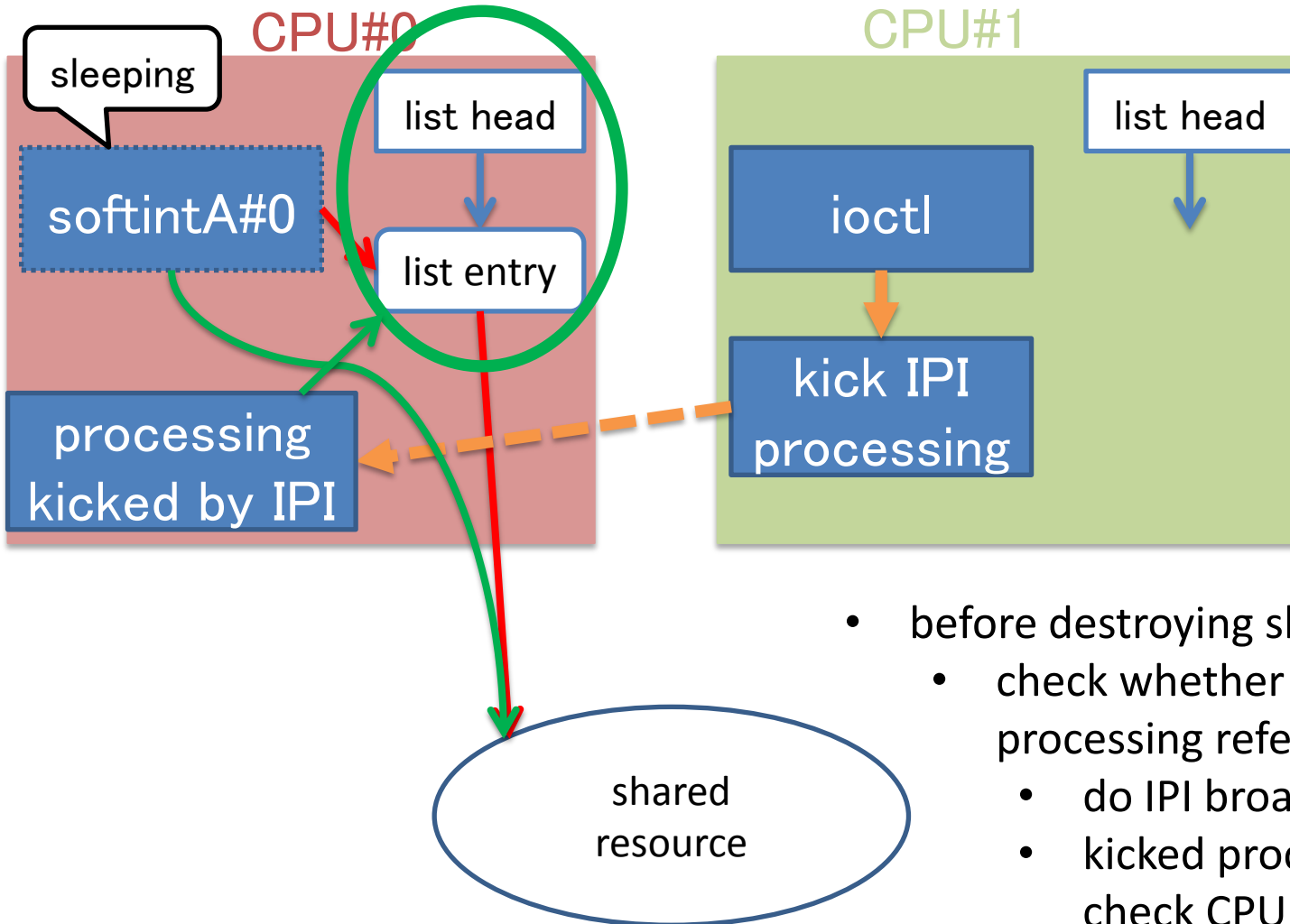
Operation overview (4/7)

release reference by CPU#1



Operation overview (5/7)

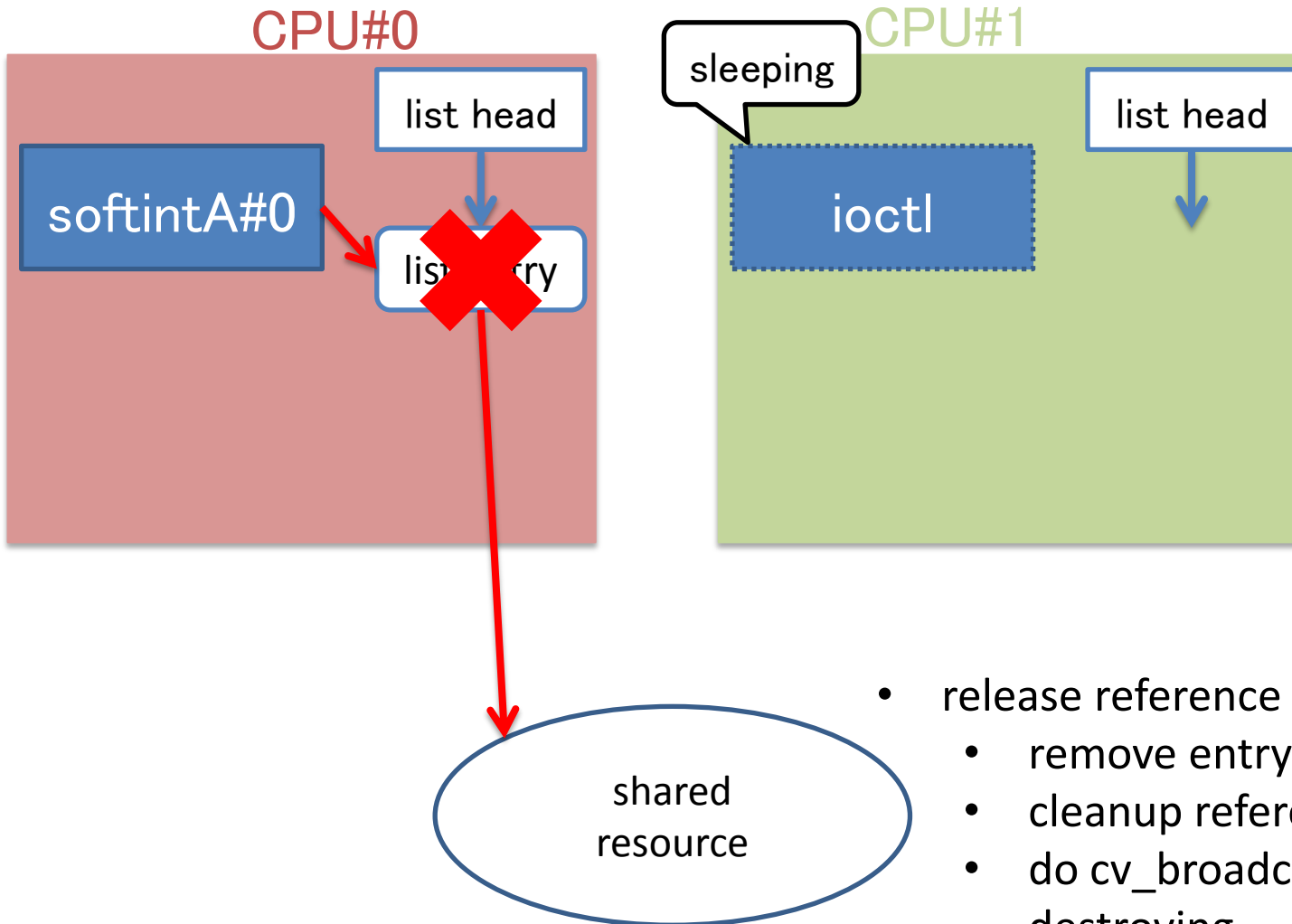
try to destroy resource by CPU#1



- before destroying shared resource
 - check whether other CPUs processing refer shared resource
 - do IPI broadcast and wait it
 - kicked processing just has to check CPU local list

Operation overview (6/7)

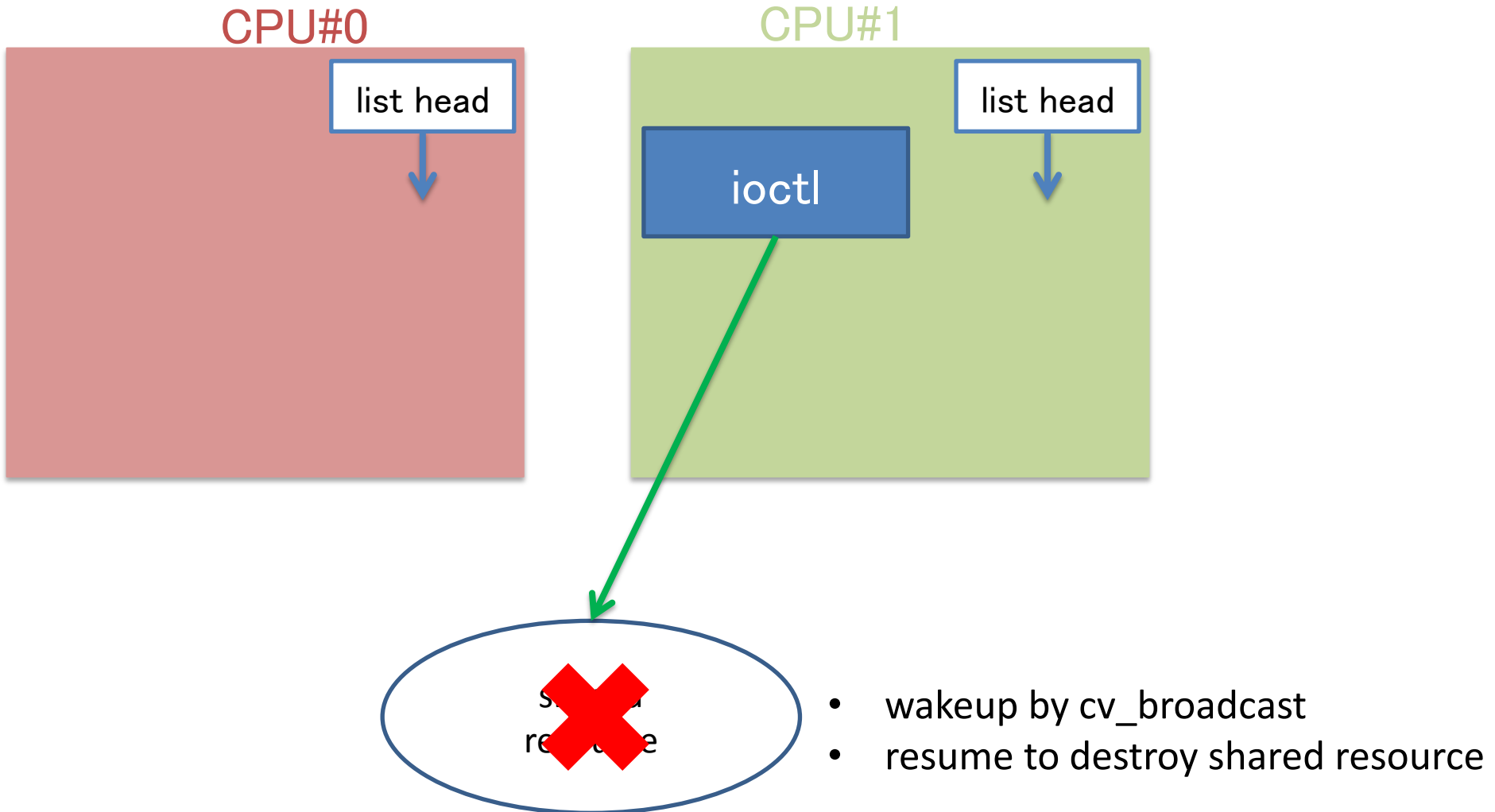
release reference by CPU#0



- release reference to shared resource
 - remove entry from CPU local list
 - cleanup reference from stack
 - do cv_broadcast if someone is destroying

Operation overview (7/7)

do destroy resource by CPU#1



APIs

```
struct psref_class *psref_class_create(const char *name, int ipl);
void    psref_class_destroy(struct psref_class *class);

void    psref_target_init(struct psref_target *target, struct psref_class *class);
void    psref_target_destroy(struct psref_target *target,
                             struct psref_class *class);

void    psref_acquire(struct psref *psref, struct psref_target *target,
                      struct psref_class *class);
void    psref_release(struct psref *psref, struct psref_target *target,
                      struct psref_class *class);
void    psref_copy(struct psref *pto, const struct psref *pfrom,
                   struct psref_class *class);

bool    psref_held(struct psref_target *target, struct psref_class *class);
```

Pseudo code (read side)

```
lookup_elem_and_sleepable_processing() {  
    struct record *elem;  
    struct psref psref;  
  
    s = pserialize_read_enter(); // protect the list itself.  
    LIST_FOREACH(elem, head, field) {  
        if (elem->key == key) {  
            psref_acquire(&psref, &elem->target); // protect the element.  
            break;  
        }  
    }  
    pserialize_read_exit(s); // unprotect the list, but  
                             // the element has been protected.  
  
    if (elem) {  
        some_processing_that_sleeps(elem);  
        psref_release(&psref, &elem->target); // psref holding reference section  
                                                // may sleep, so this cannot do  
                                                // before pserialize_read_exit().  
                                                // unprotect the element.  
    }  
}
```

// to keep the reference across
// function, pass psref as argument.

Pseudo code (write side)

```
remove_elem() {

    mutex_enter(lock);                                // protect against other write
                                                       // side processing.

    LIST_FOREACH(elem, head, field) {
        if (elem->key == key) {
            LIST_REMOVE(elem);
            pserialize_perform(psz);                // wait for reader lookups to
                                                       // finish.

            break;
        }
    }
    mutex_exit(lock);

    if (elem) {
        psref_target_destroy(&elem->psref_target); // wait for readers to drain.
        kmem_free(elem);                            // destroy itself.
    }
}
```

Summary

- Introduce psref
- psref enables us to work on parallelizing packet processing incrementally without making the significant changes
 - The significant changes are needed as pserialize(9) read side would require to avoid sleeping
- psref will be merged to NetBSD-current
 - soon?
- Welcome to feedback to use pserialize(9) and psref
- If you have questions, please ask riastradh@n.o 😊