Linux NAPI & MultiQueue + ルーティングテーブル

浅間 正和 @ 有限会社 銀座堂

2014/11/18

Internet Week 2014

S6 パケットフォワーディング＆ルーティングの実装技術

目次

- 高速に経路探索を行うための Linux の実装
  - いろいろな木
  - Linux kernel の IPv4 ルーティングテーブル
- 大量のパケットを処理するための Linux の実装
  - Receive Livelock と Linux NAPI
  - MultiQueue NIC と Receive Side Scaling(RSS)
- まとめ

目次

- 高速に経路探索を行うための Linux の実装
  - いろいろな木
  - Linux kernel の IPv4 ルーティングテーブル
- 大量のパケットを処理するための Linux の実装
  - Receive Livelock と Linux NAPI
  - MultiQueue NIC と Receive Side Scaling(RSS)
- まとめ

問題

- ある固定長のビット列の集合から与えられたキーに一致するものを探索したい

- 例として
  - 0000b
  - 0001b
  - 0010b
  - 0011b
  - 1111b

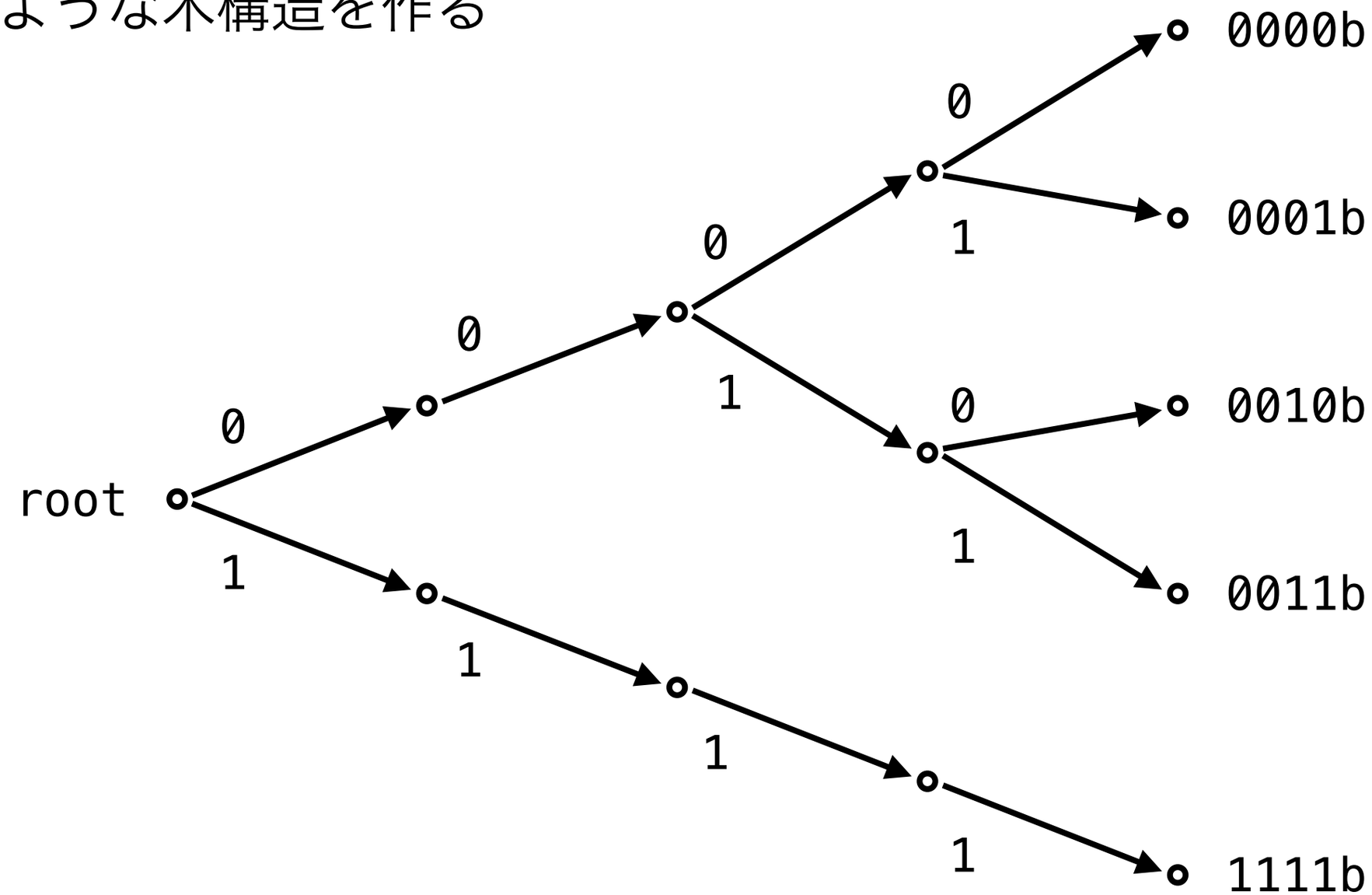
  の 5 つの要素が存在するビット列の集合を考える
- 与えられたキーが
  - 0011b

    → 4 個目と一致
  - 1100b

    → 一致しない

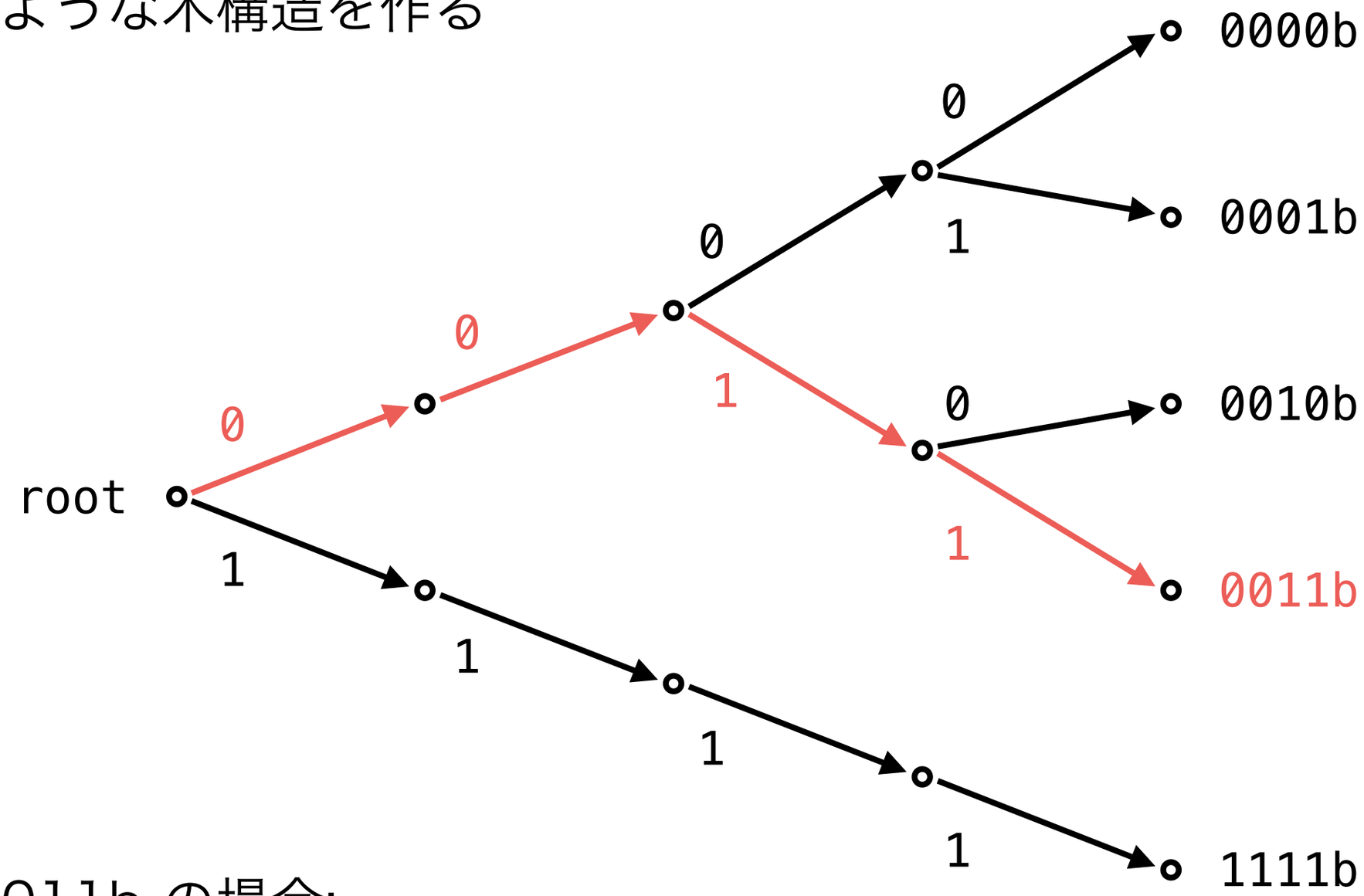
Binary Trie

左端から 1 ビットずつ 0 か 1 かで分岐
していくような木構造を作る



Binary Trie

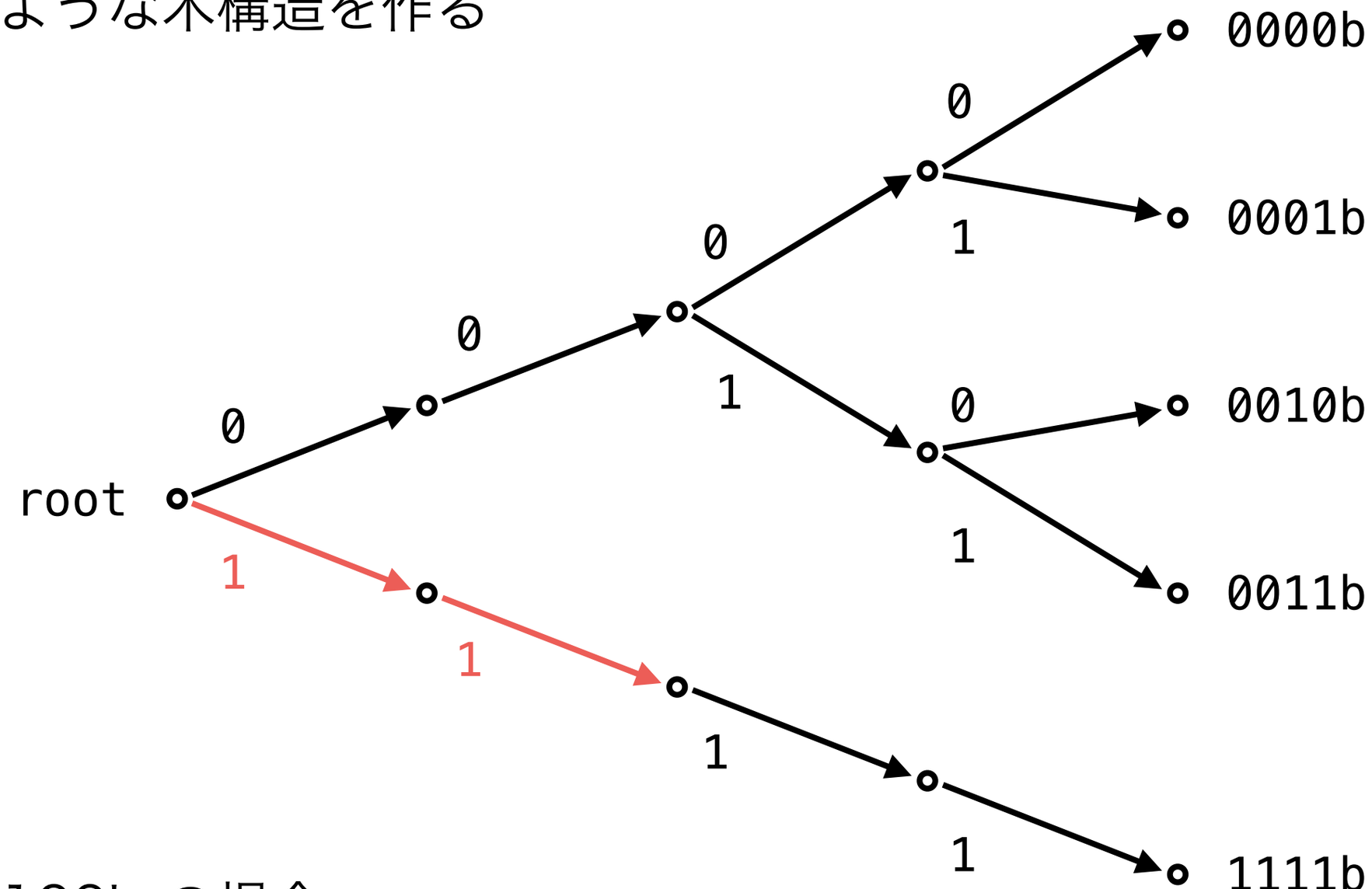
左端から 1 ビットずつ 0 か 1 かで分岐
していくような木構造を作る



キーが 0011b の場合:
赤色の矢印を辿り一致することが判る

Binary Trie

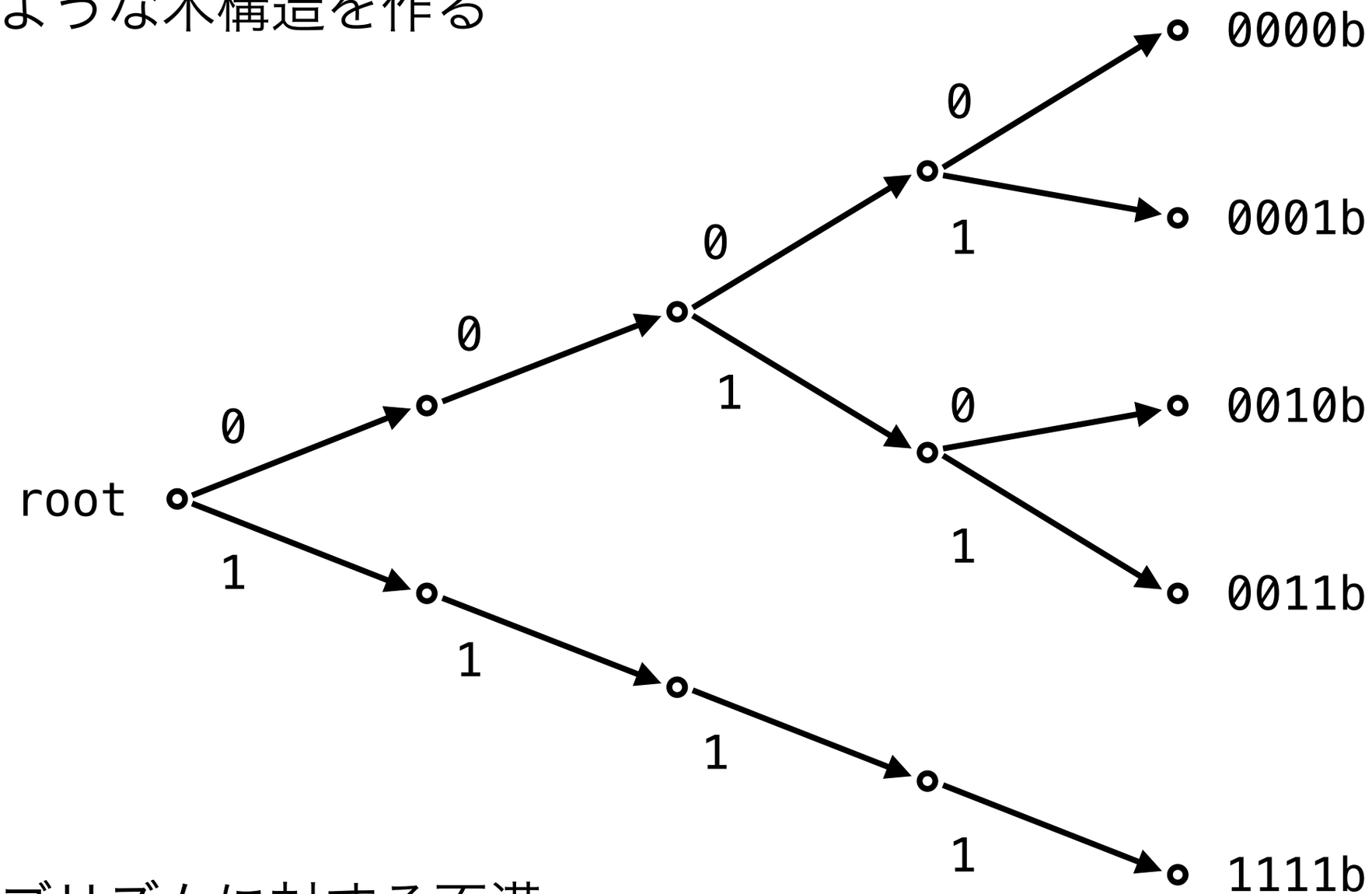
左端から 1 ビットずつ 0 か 1 かで分岐
していくような木構造を作る



キーが 1100b の場合:
赤色の矢印を辿り root から 2 つめのノードで
0 に対応する枝が存在しない為一致しない

Binary Trie

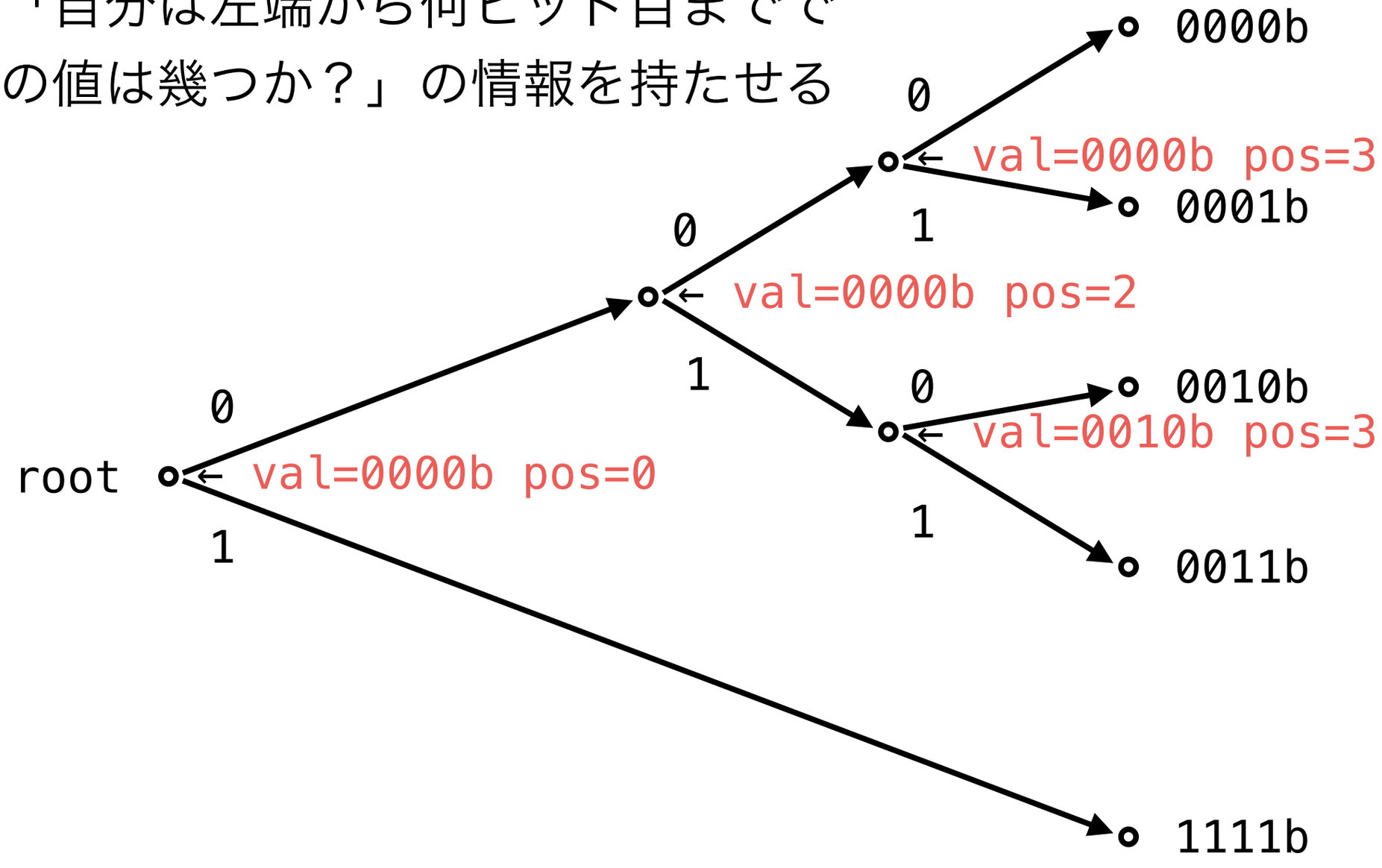
左端から 1 ビットずつ 0 か 1 かで分岐
していくような木構造を作る



このアルゴリズムに対する不満:
分岐していないノードを辿るのは無駄
(例: root → 1111b)

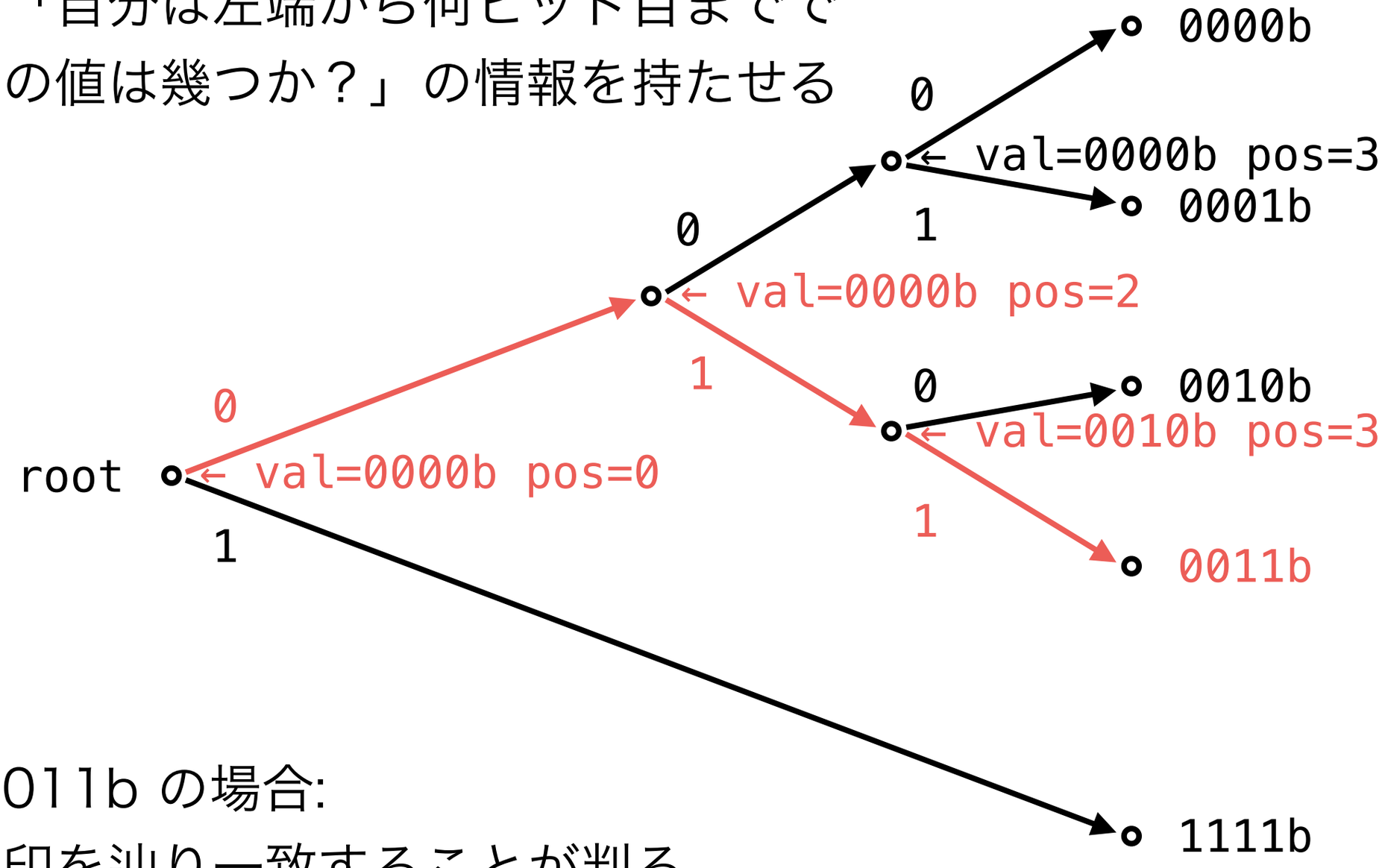
Path Compressed Binary Trie(Patricia Trie)

分岐がないときはスキップさせてしまう
ノードに「自分は左端から何ビット目までで
そこまでの値は幾つか？」の情報を持たせる



Path Compressed Binary Trie(Patricia Trie)

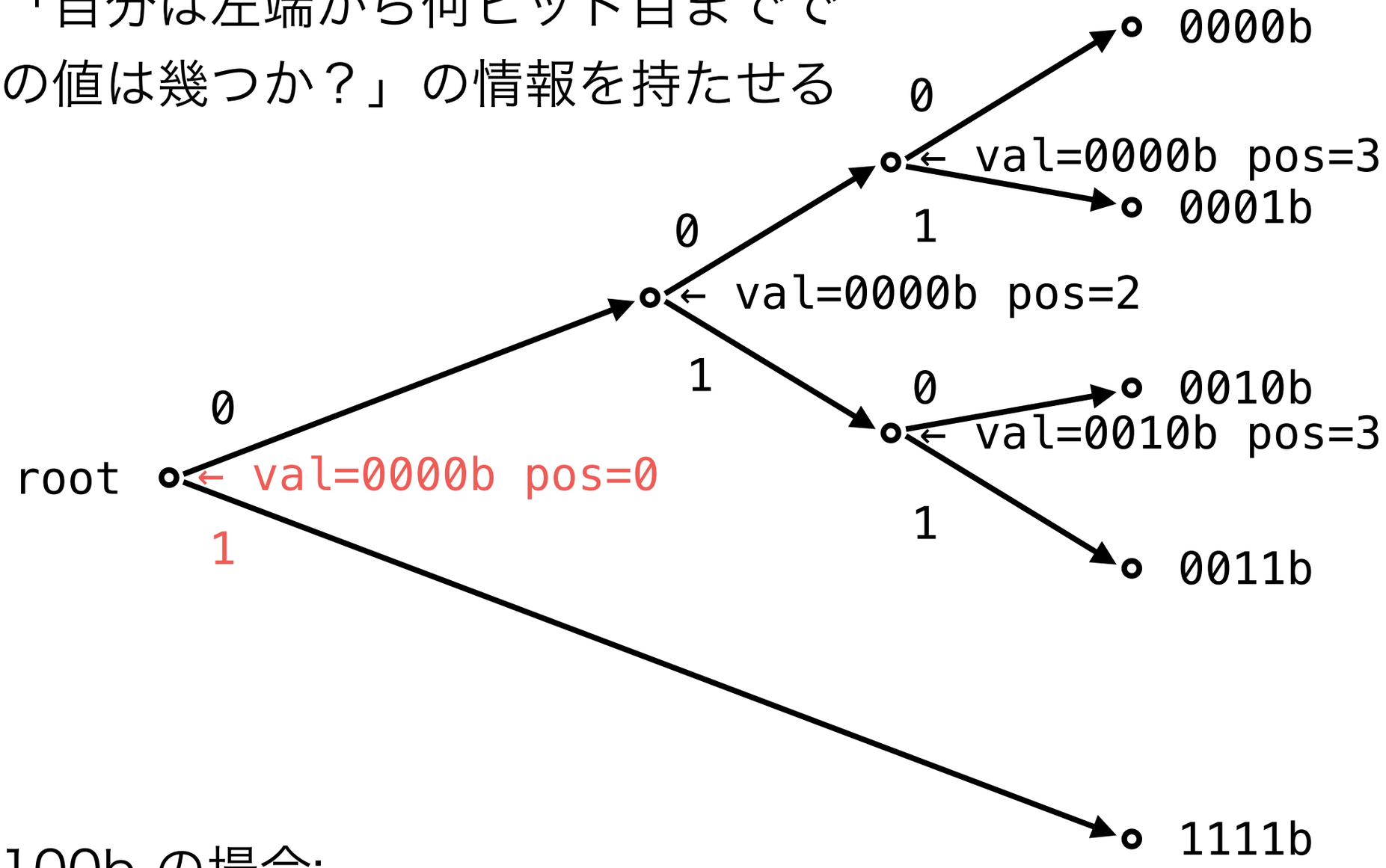
分岐がないときはスキップさせてしまう
ノードに「自分は左端から何ビット目までで
そこまでの値は幾つか？」の情報を持たせる



キーが 0011b の場合:
赤色の矢印を辿り一致することが判る
root から 2 つめのノードに飛んだ時 pos が 0 から 2 に飛んでいるので
キーがノードの val の [0, 2) と一致するか確認する

Path Compressed Binary Trie(Patricia Trie)

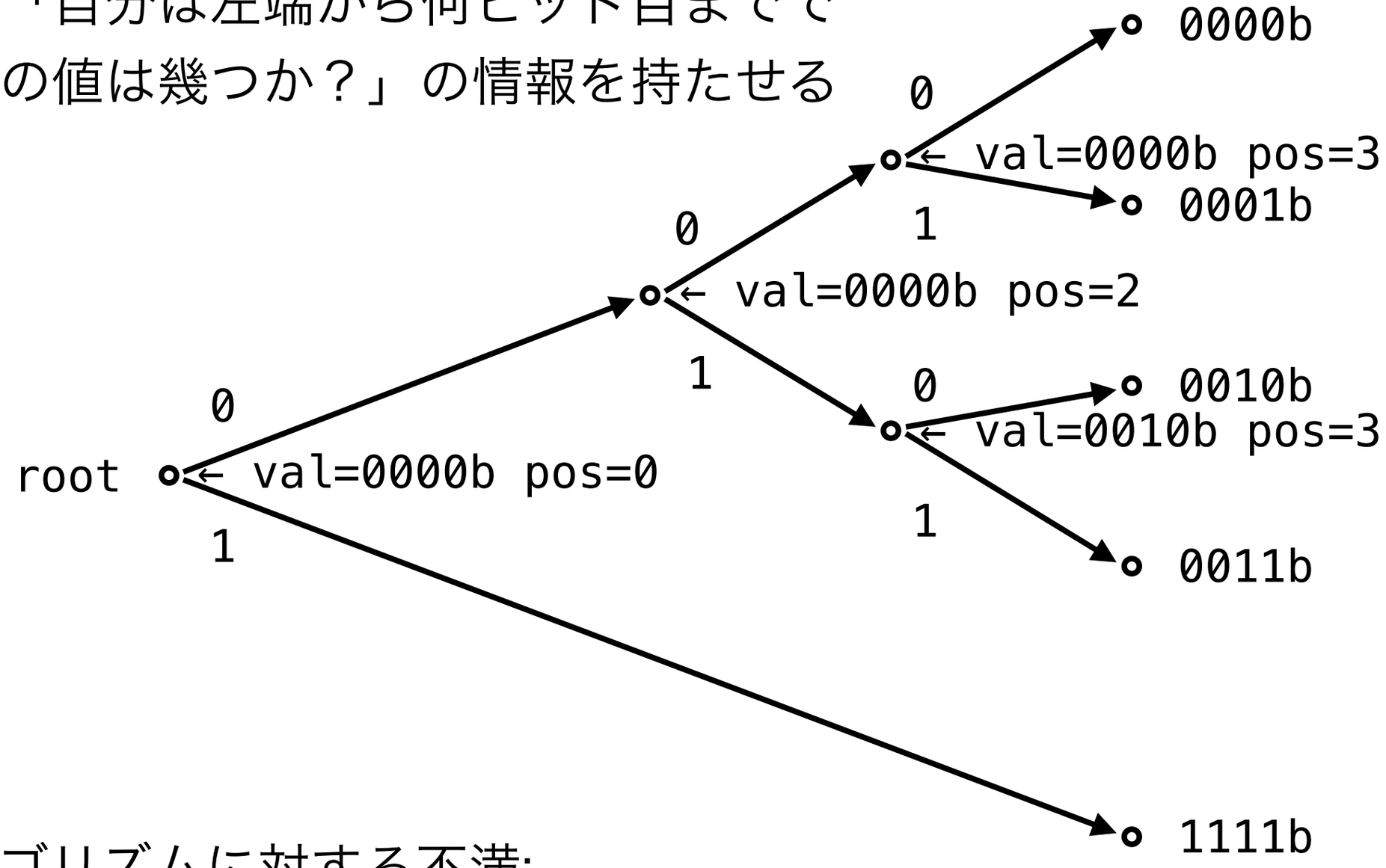
分岐がないときはスキップさせてしまう
ノードに「自分は左端から何ビット目までで
そこまでの値は幾つか？」の情報を持たせる



キーが 1100b の場合:
赤色の矢印を辿り root の次のノードがリーフ
で値が一致しないので一致しない

Path Compressed Binary Trie(Patricia Trie)

分岐がないときはスキップさせてしまう
ノードに「自分は左端から何ビット目までで
そこまでの値は幾つか？」の情報を持たせる

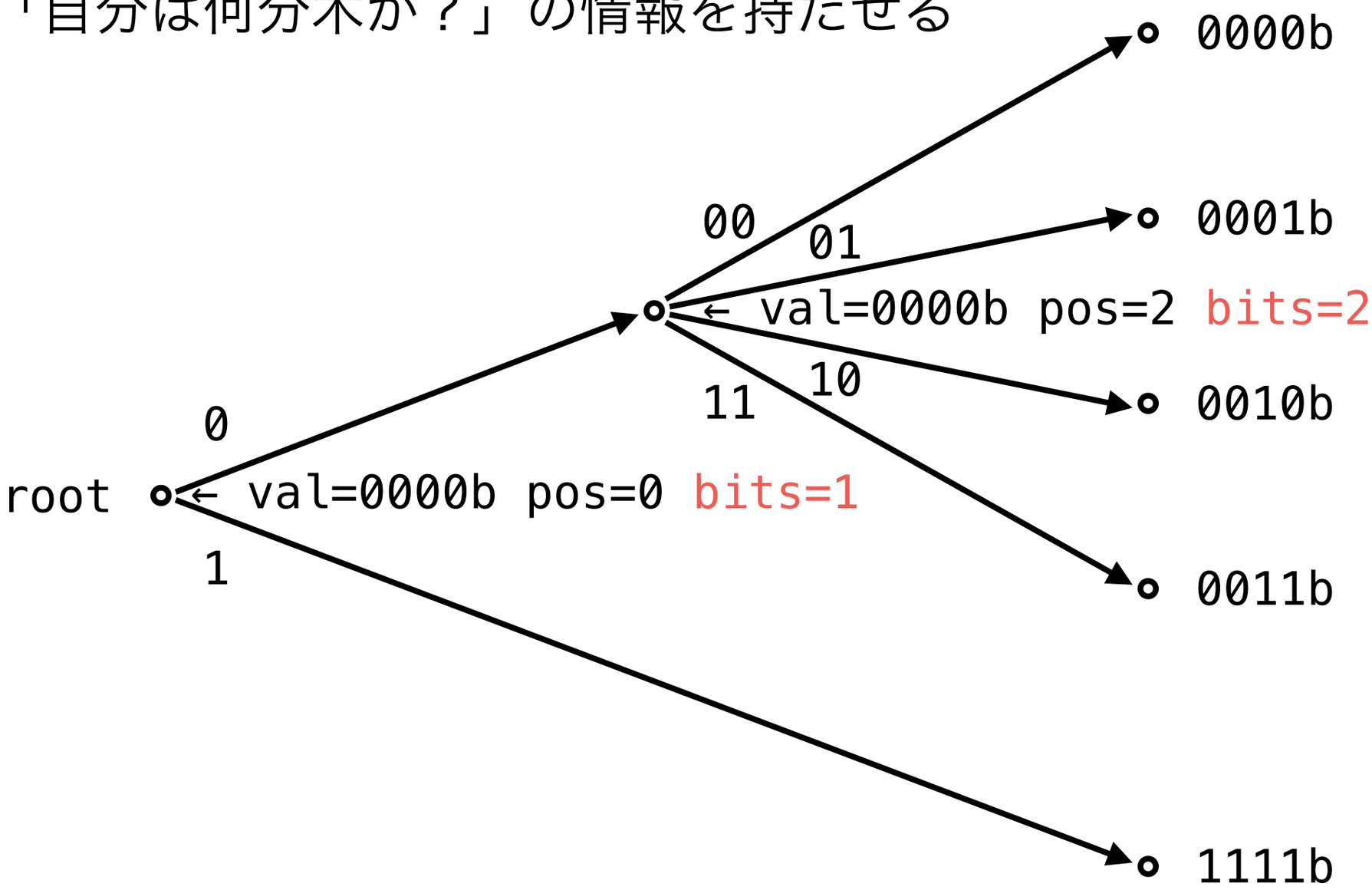


このアルゴリズムに対する不満:
ノードの密度が高い時は一発でリーフに近いノードに飛べるのでは？
(例: val=0000b pos=2 のノードから 2^2 通り枝が全て存在している)

Perfect Level and Path Compressed(LPC-)Trie

ノードが密集しているときは 2^n 分木にしてしまう(レベル圧縮)

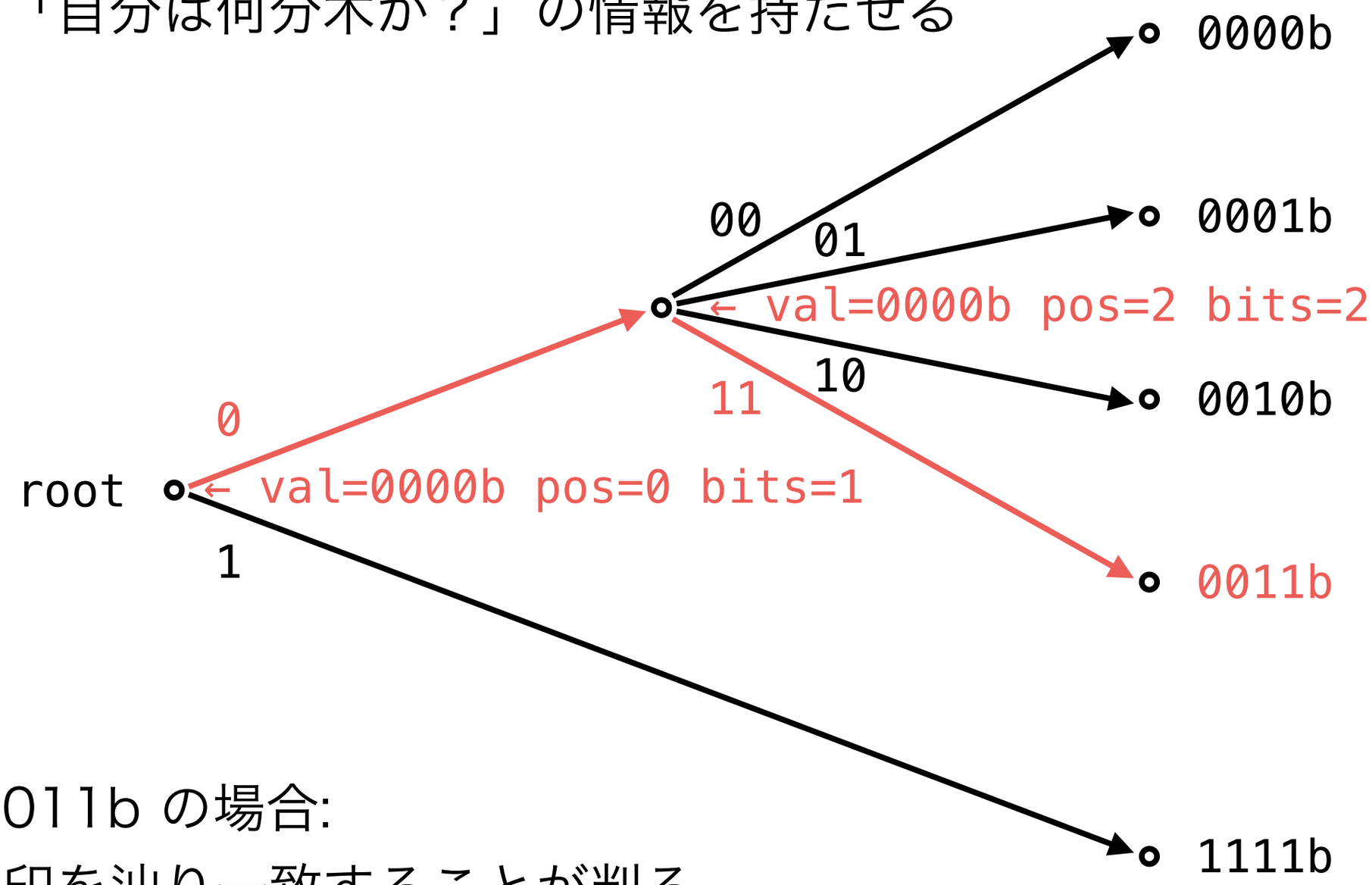
ノードに「自分は何分木か？」の情報を持たせる



Perfect Level and Path Compressed(LPC-)Trie

ノードが密集しているときは 2^n 分木にしてしまう(レベル圧縮)

ノードに「自分は何分木か？」の情報を持たせる



キーが 0011b の場合:

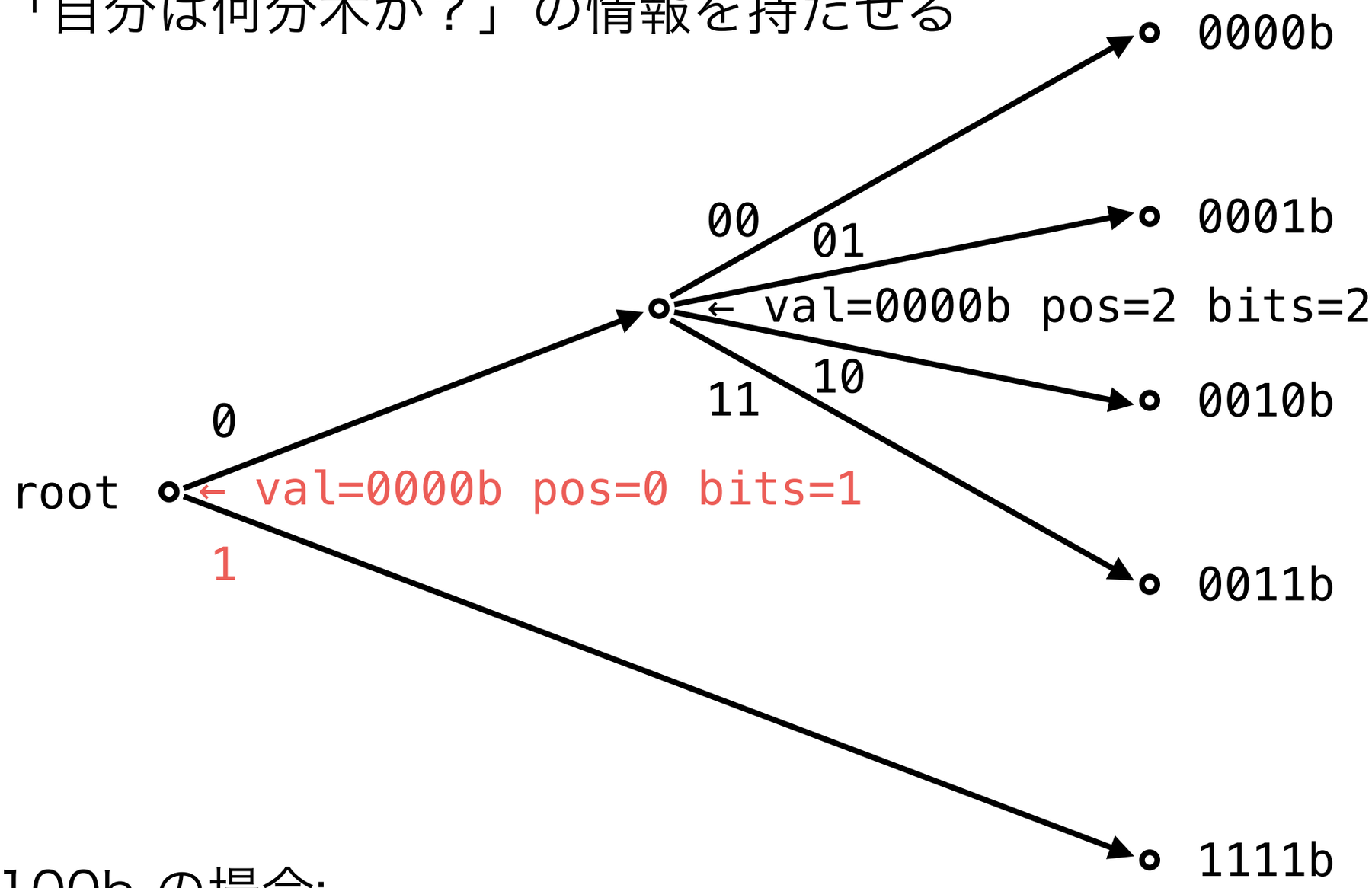
赤色の矢印を辿り一致することが判る

root から 2 つめのノードで bits が 2 になっているのでキーの [2, 4) ビットの値から 11 に対応する枝を辿る

Perfect Level and Path Compressed(LPC-)Trie

ノードが密集しているときは 2^n 分木にしてしまう(レベル圧縮)

ノードに「自分は何分木か？」の情報を持たせる



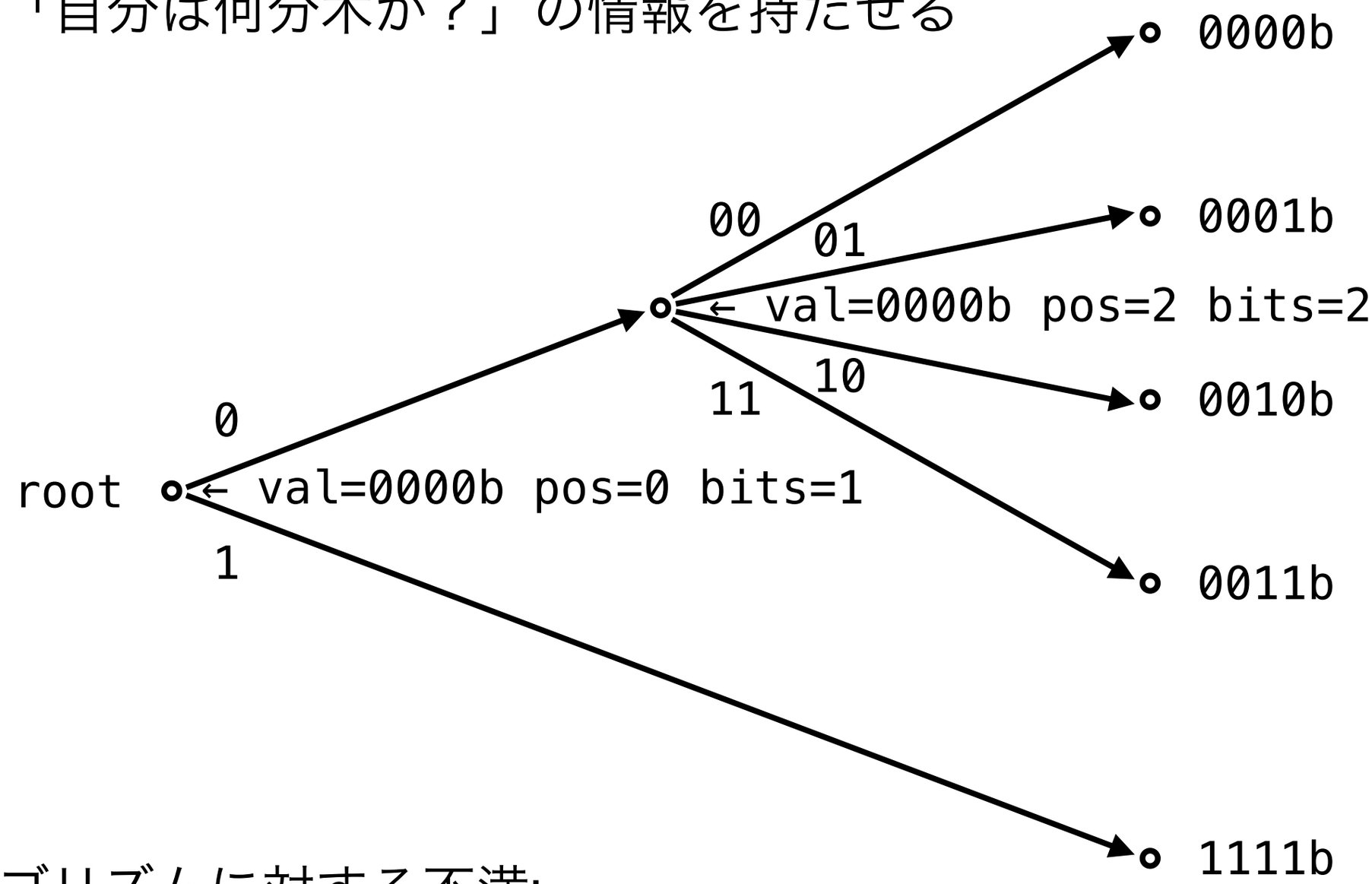
キーが 1100b の場合:

赤色の矢印を辿り root の次のノードがリーフで値が一致しないので一致しない

Perfect Level and Path Compressed(LPC-)Trie

ノードが密集しているときは 2^n 分木にしてしまう(レベル圧縮)

ノードに「自分は何分木か？」の情報を持たせる



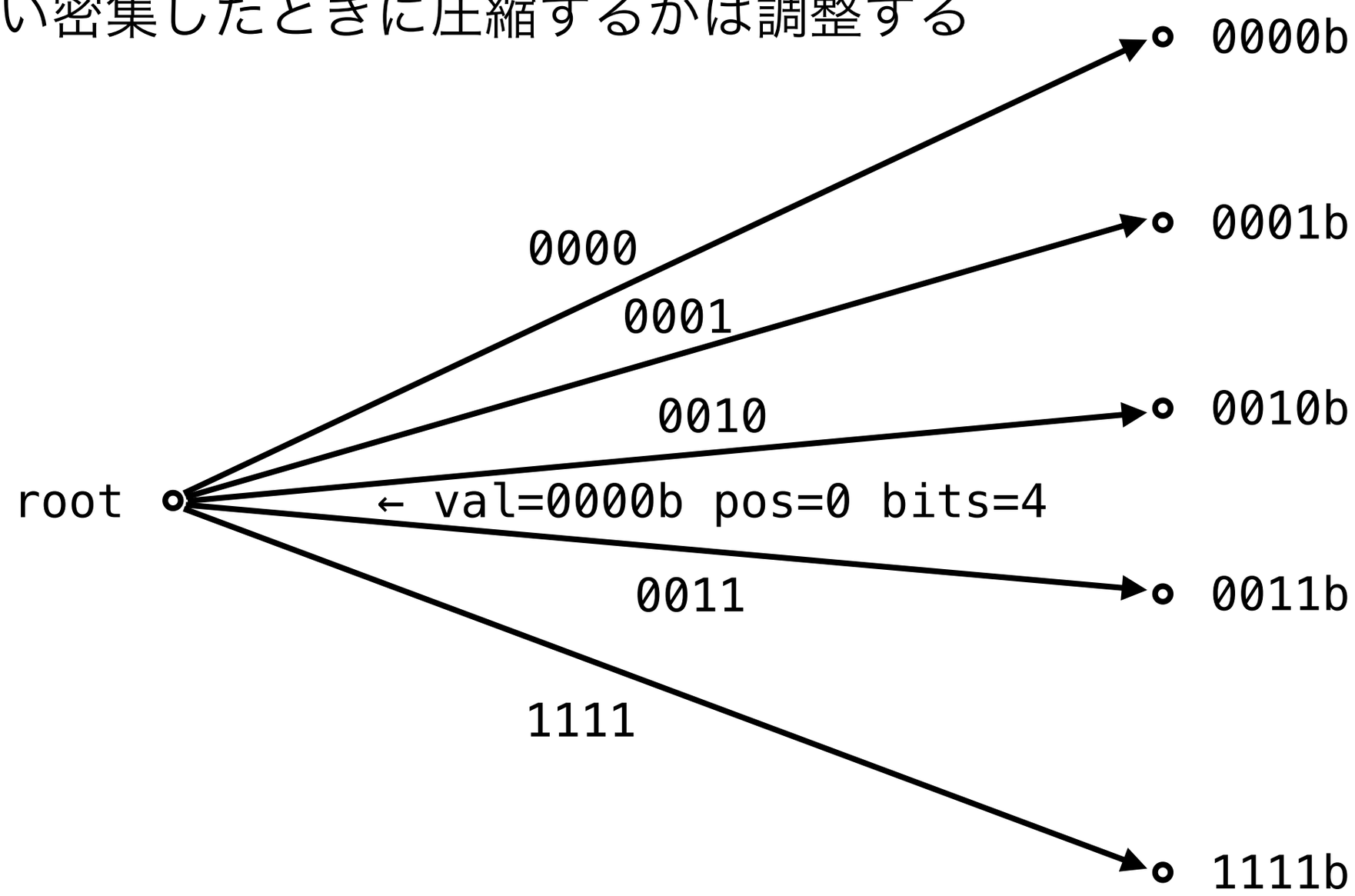
このアルゴリズムに対する不満:

もっと少ないノードの移動で探索できるようにならないの？

(だんだん欲深くなってきている)

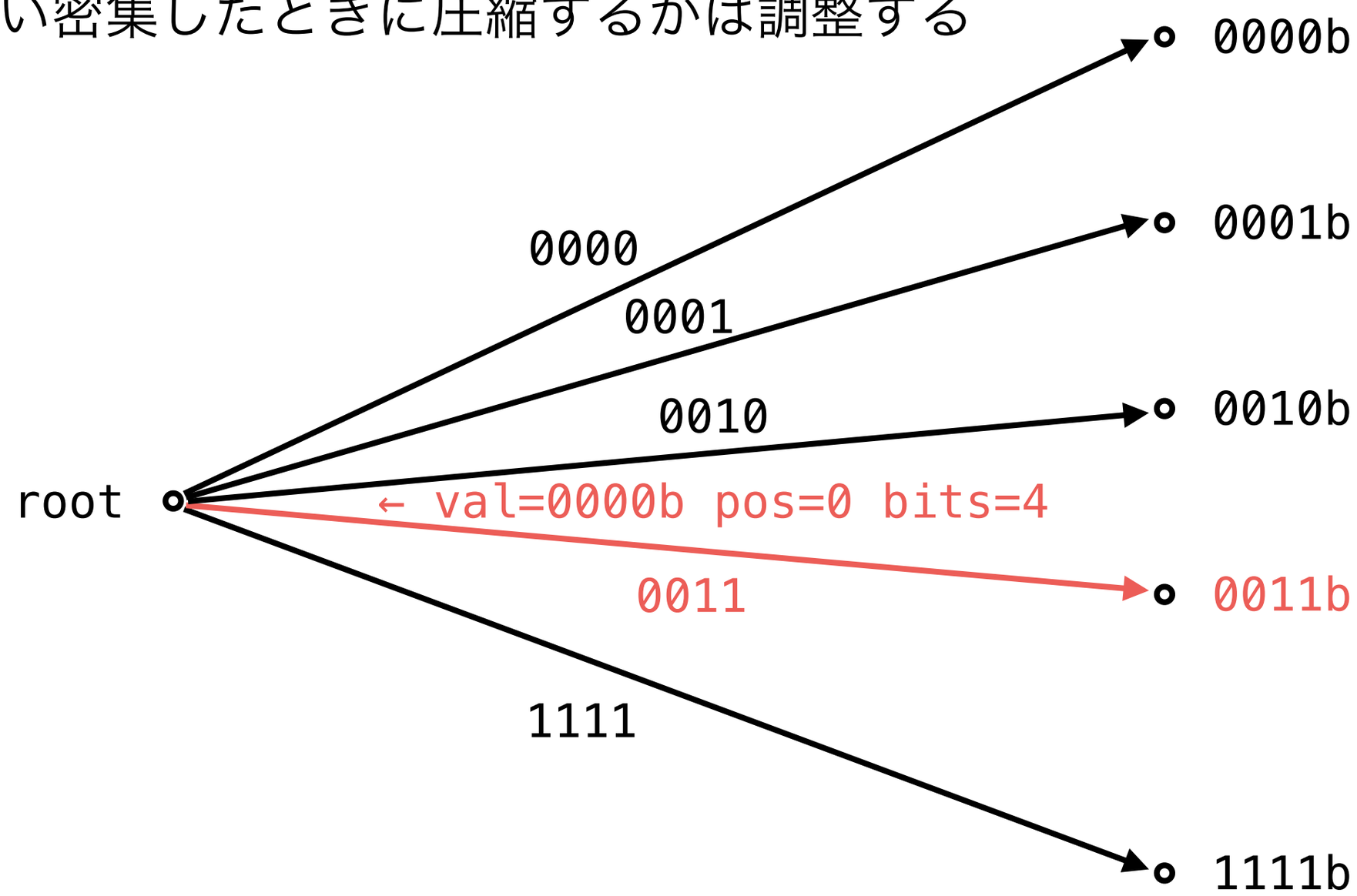
Relaxed Level and Path Compressed(LPC-)Trie

2^n 個の枝を満たせなくても(隙間が生じても)レベル圧縮
どれくらい密集したときに圧縮するかは調整する



Relaxed Level and Path Compressed(LPC-)Trie

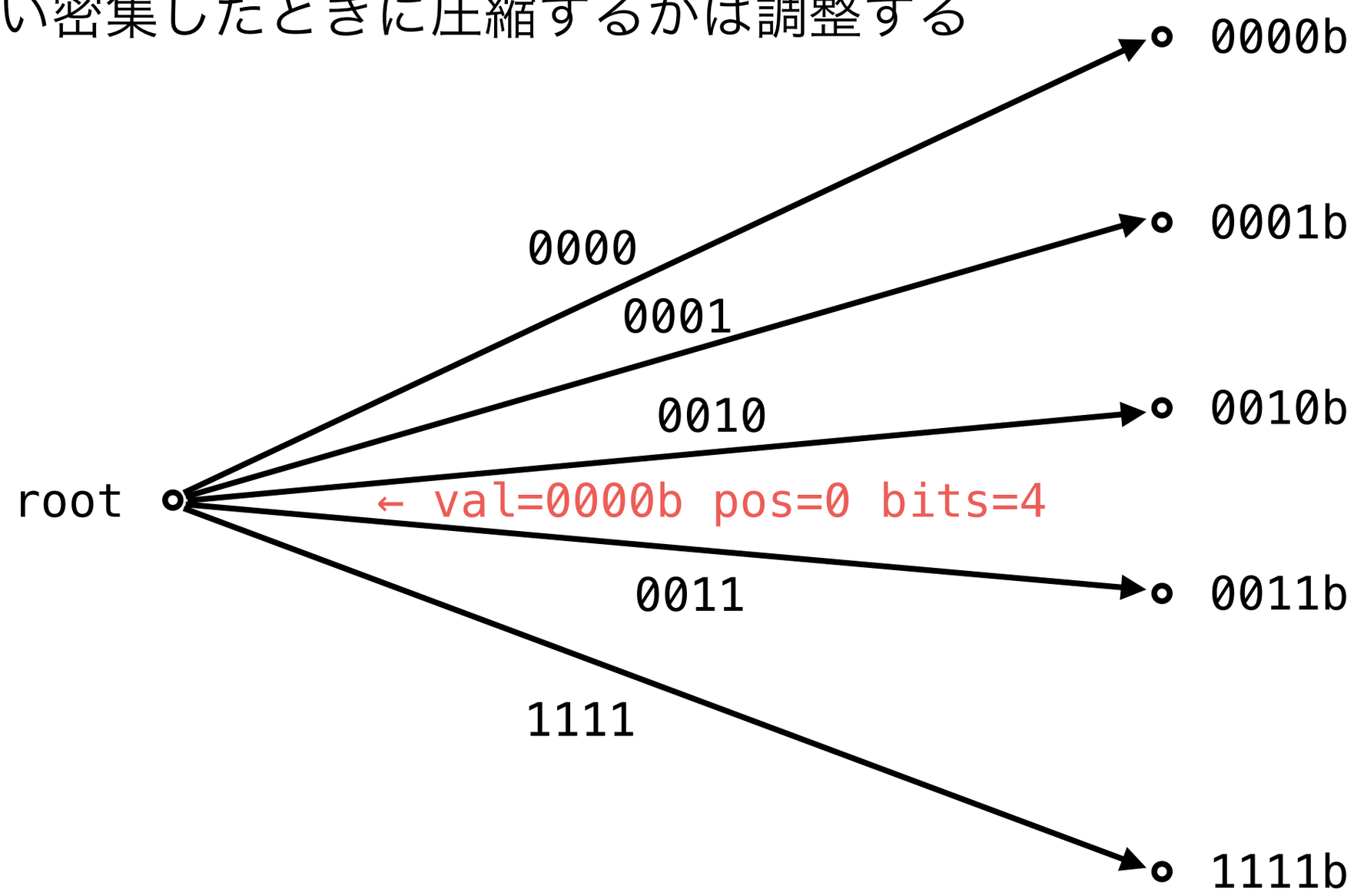
2^n 個の枝を満たせなくても(隙間が生じても)レベル圧縮
どれくらい密集したときに圧縮するかは調整する



キーが 0011b の場合:
赤色の矢印を辿り一致することが判る

Relaxed Level and Path Compressed(LPC-)Trie

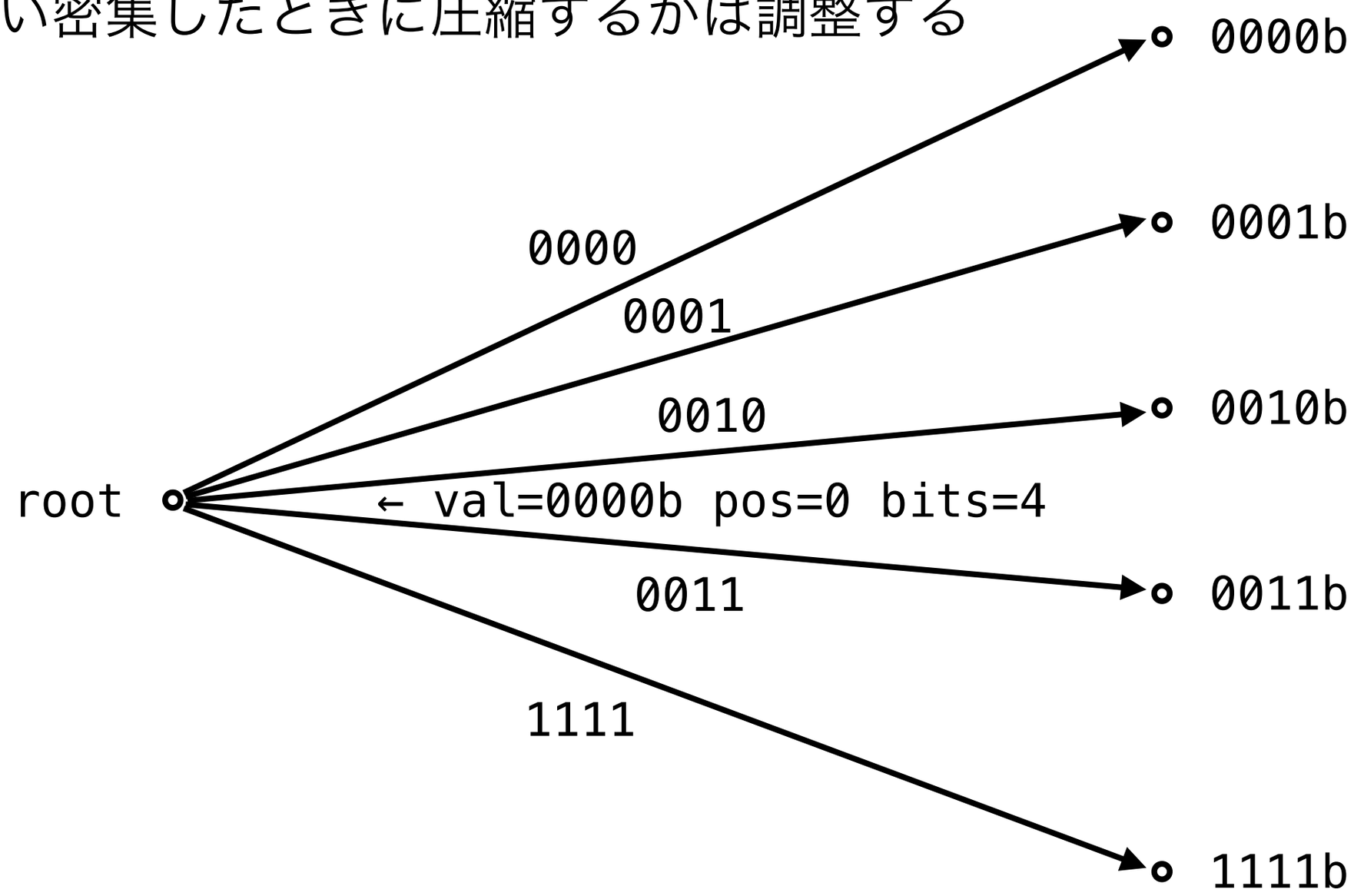
2^n 個の枝を満たせなくても(隙間が生じても)レベル圧縮
どれくらい密集したときに圧縮するかは調整する



キーが 1100b の場合:
root ノードに対応する枝が存在しないので一致しない

Relaxed Level and Path Compressed(LPC-)Trie

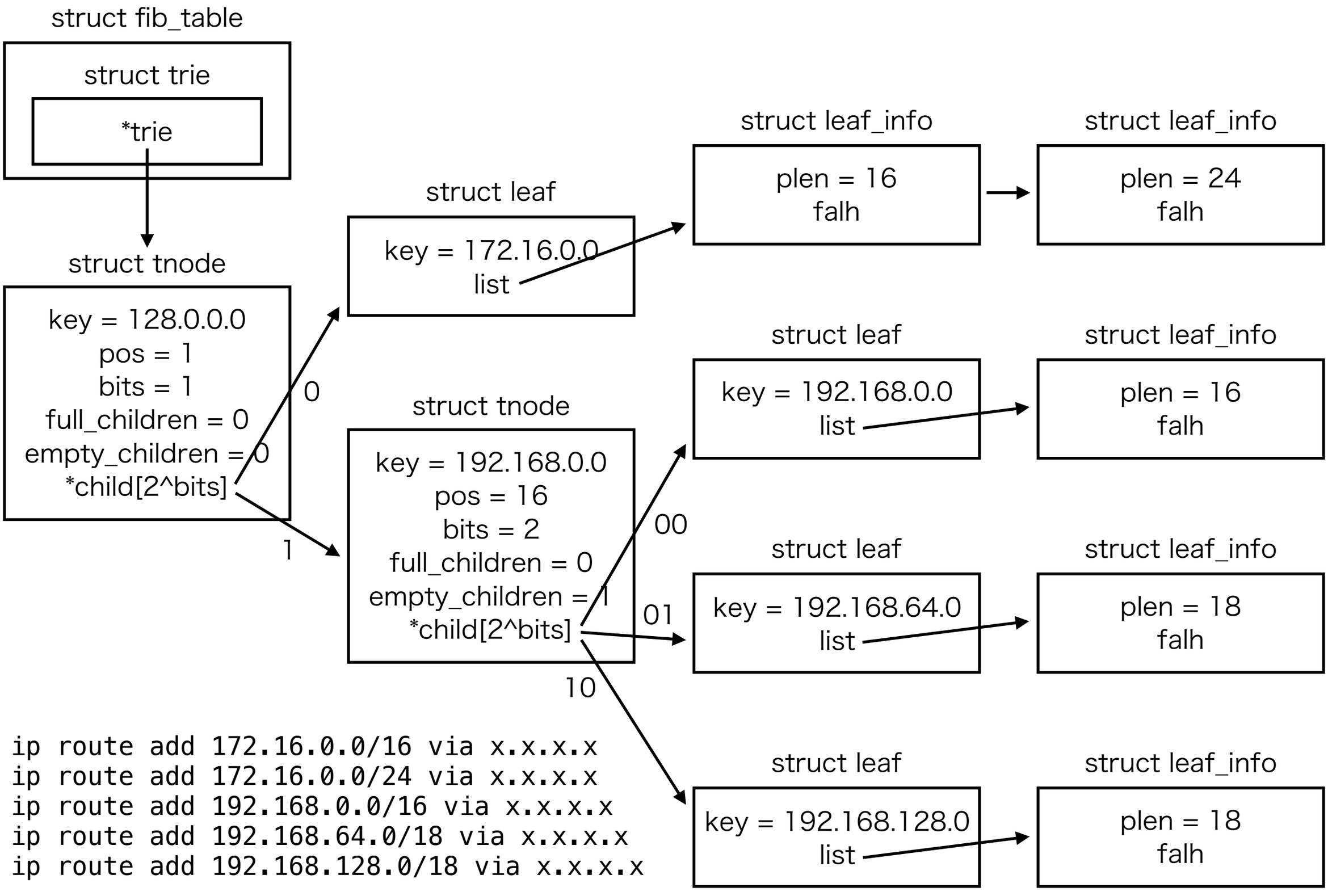
2^n 個の枝を満たせなくても(隙間が生じても)レベル圧縮
どれくらい密集したときに圧縮するかは調整する



Linux kernel 2.6.39 以降の IPv4 ルーティングテーブルはこの方式

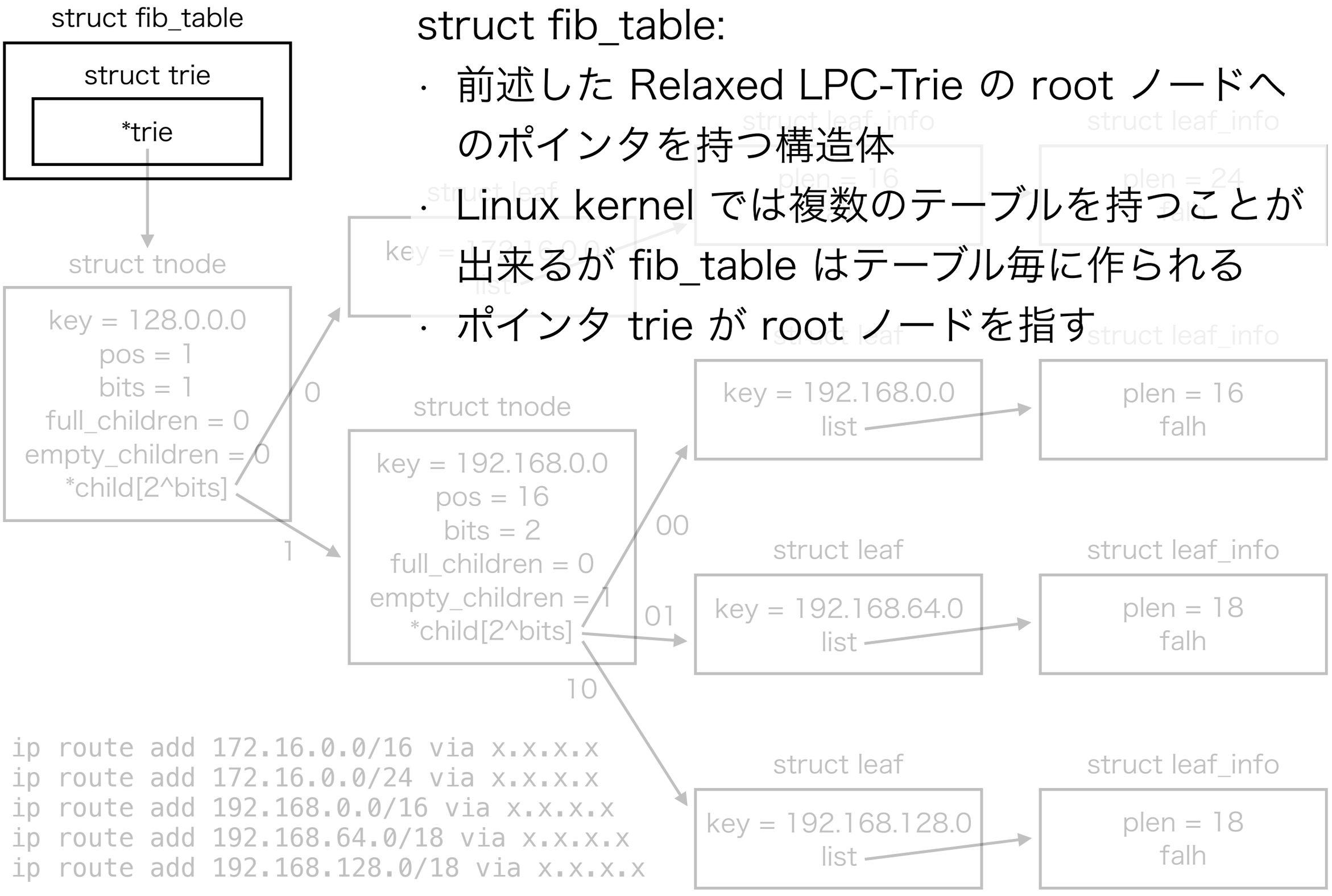
レベル圧縮後に 50% 以上の枝が埋まる場合にレベル圧縮を実施

Linux kernel 内部のデータ構造の例




```
ip route add 172.16.0.0/16 via x.x.x.x
ip route add 172.16.0.0/24 via x.x.x.x
ip route add 192.168.0.0/16 via x.x.x.x
ip route add 192.168.64.0/18 via x.x.x.x
ip route add 192.168.128.0/18 via x.x.x.x
```

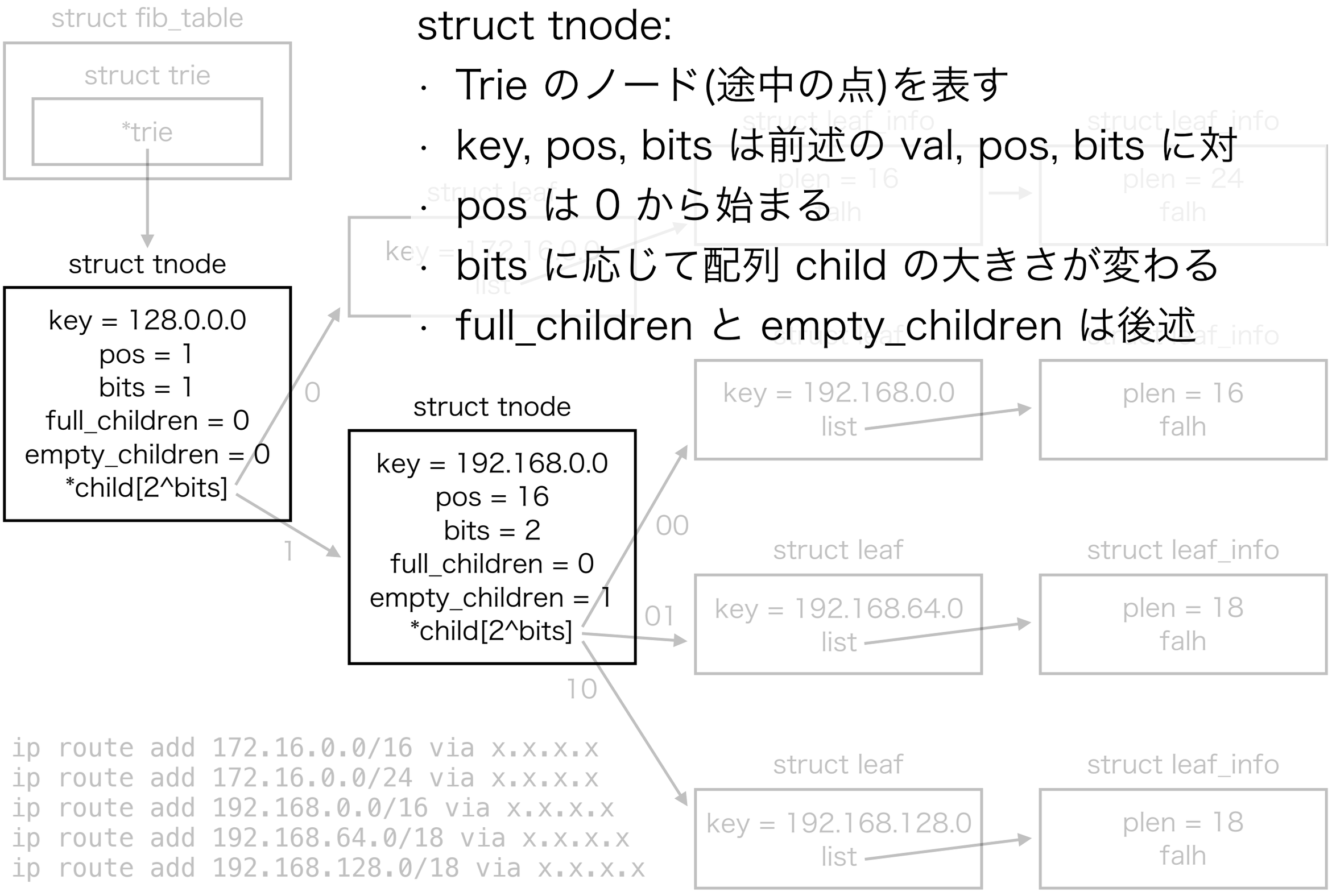
Linux kernel 内部のデータ構造の例



- struct fib_table:
- 前述した Relaxed LPC-Trie の root ノードへのポインタを持つ構造体
- Linux kernel では複数のテーブルを持つことが出来るが fib_table はテーブル毎に作られる
- ポインタ trie が root ノードを指す

```
ip route add 172.16.0.0/16 via x.x.x.x
ip route add 172.16.0.0/24 via x.x.x.x
ip route add 192.168.0.0/16 via x.x.x.x
ip route add 192.168.64.0/18 via x.x.x.x
ip route add 192.168.128.0/18 via x.x.x.x
```

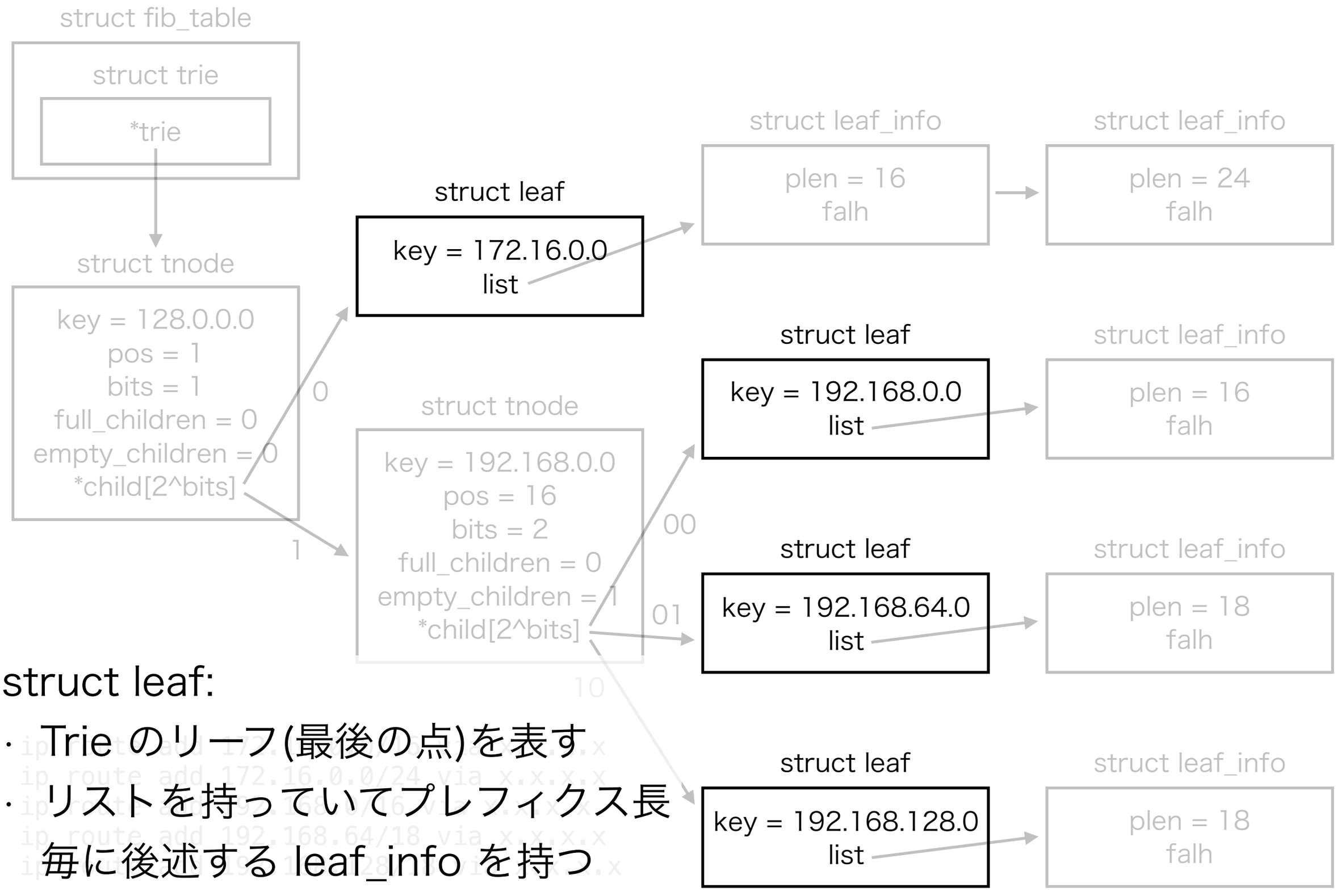
Linux kernel 内部のデータ構造の例



- struct tnode:
- Trie のノード(途中の点)を表す
- key, pos, bits は前述の val, pos, bits に対
- pos は 0 から始まる
- bits に応じて配列 child の大きさが変わる
- full_children と empty_children は後述

```
ip route add 172.16.0.0/16 via x.x.x.x
ip route add 172.16.0.0/24 via x.x.x.x
ip route add 192.168.0.0/16 via x.x.x.x
ip route add 192.168.64.0/18 via x.x.x.x
ip route add 192.168.128.0/18 via x.x.x.x
```

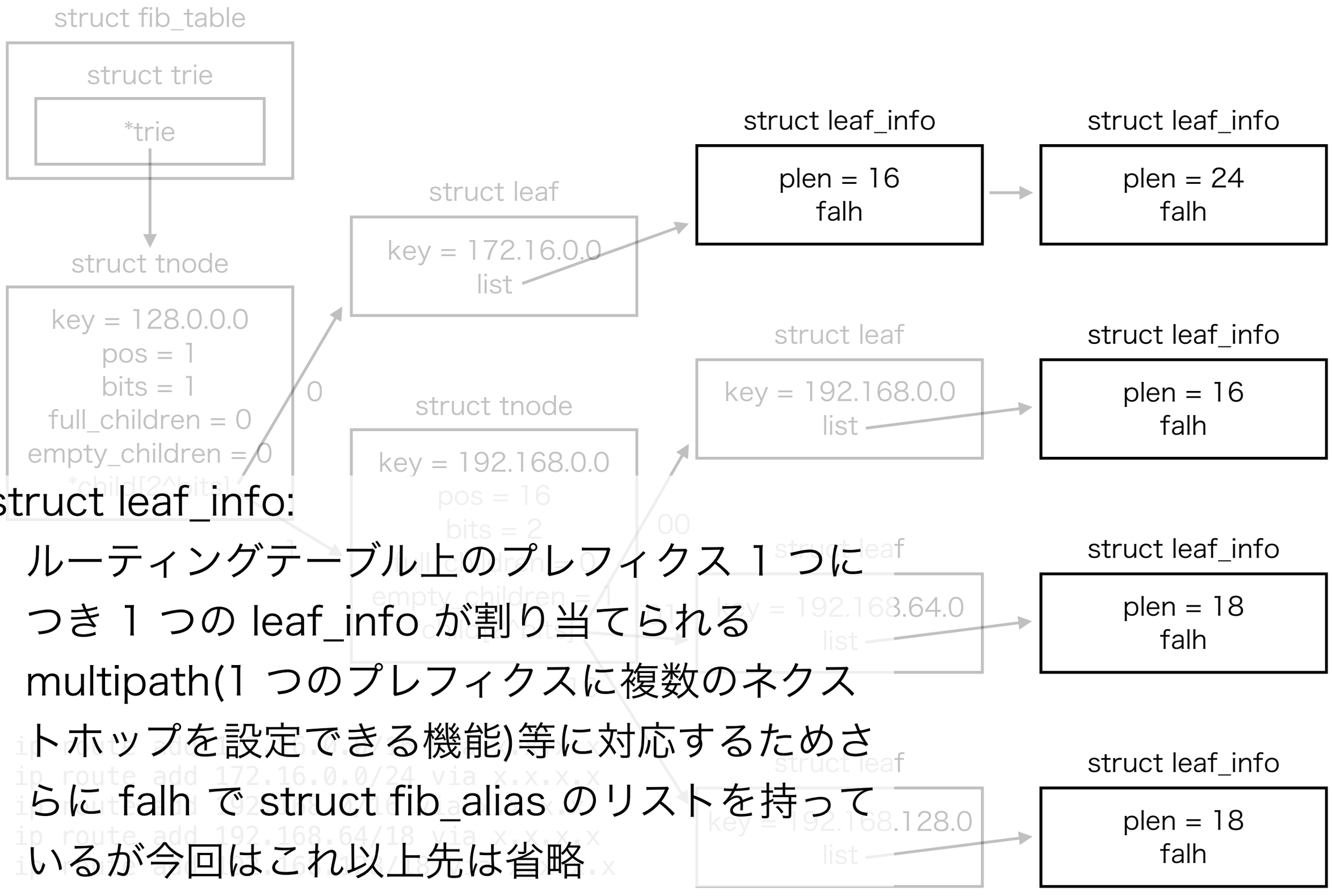
Linux kernel 内部のデータ構造の例



struct leaf:

- Trie のリーフ(最後の点)を表す
- リストを持っていてプレフィクス長毎に後述する leaf_info を持つ

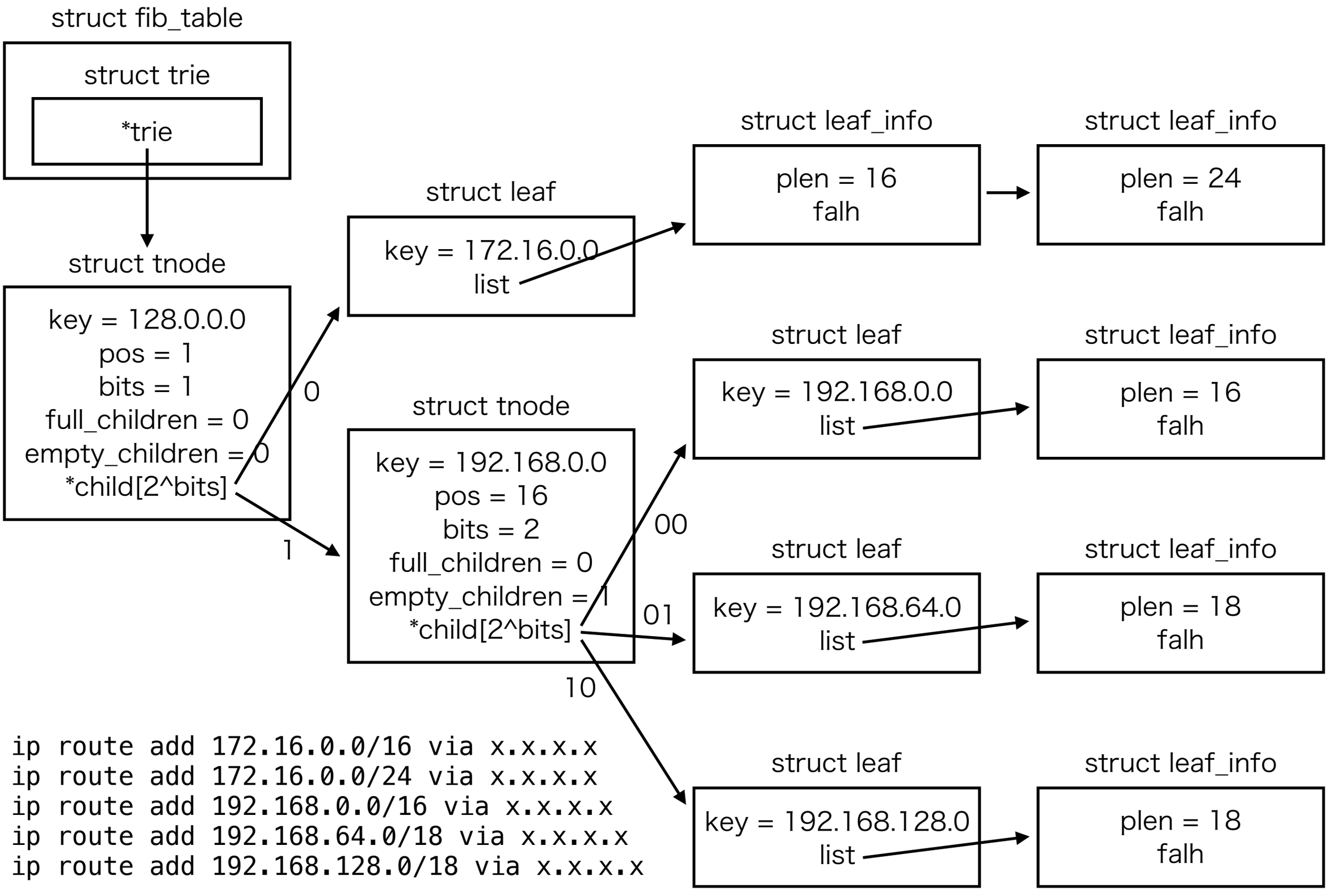
Linux kernel 内部のデータ構造の例



struct leaf_info:

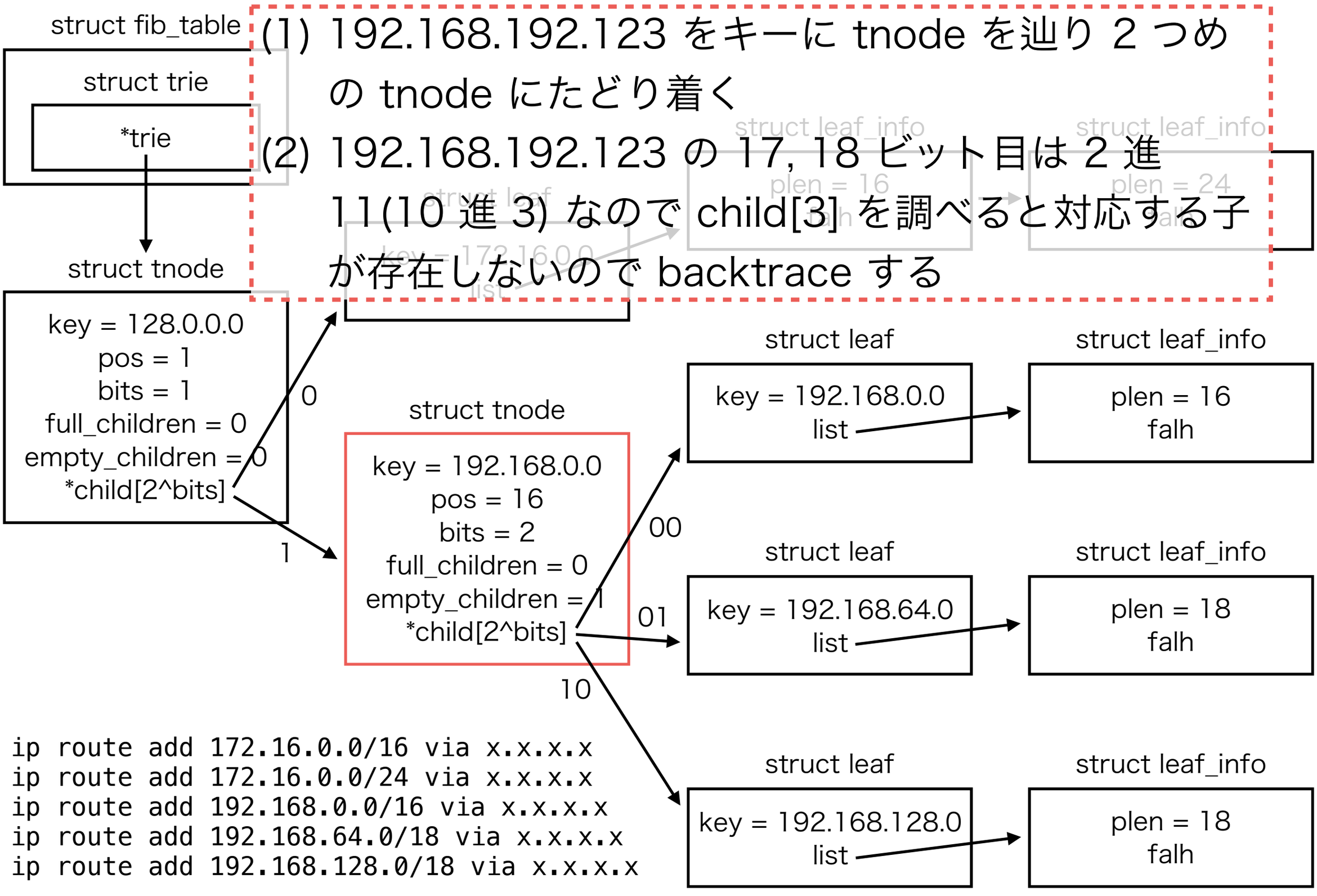
- ルーティングテーブル上のプレフィクス 1 つにつき 1 つの leaf_info が割り当てられる
- multipath(1 つのプレフィクスに複数のネクストホップを設定できる機能)等に対応するためさらに falh で struct fib_alias のリストを持っているが今回はこれ以上先は省略

Linux kernel 内部のデータ構造の例




```
ip route add 172.16.0.0/16 via x.x.x.x
ip route add 172.16.0.0/24 via x.x.x.x
ip route add 192.168.0.0/16 via x.x.x.x
ip route add 192.168.64.0/18 via x.x.x.x
ip route add 192.168.128.0/18 via x.x.x.x
```

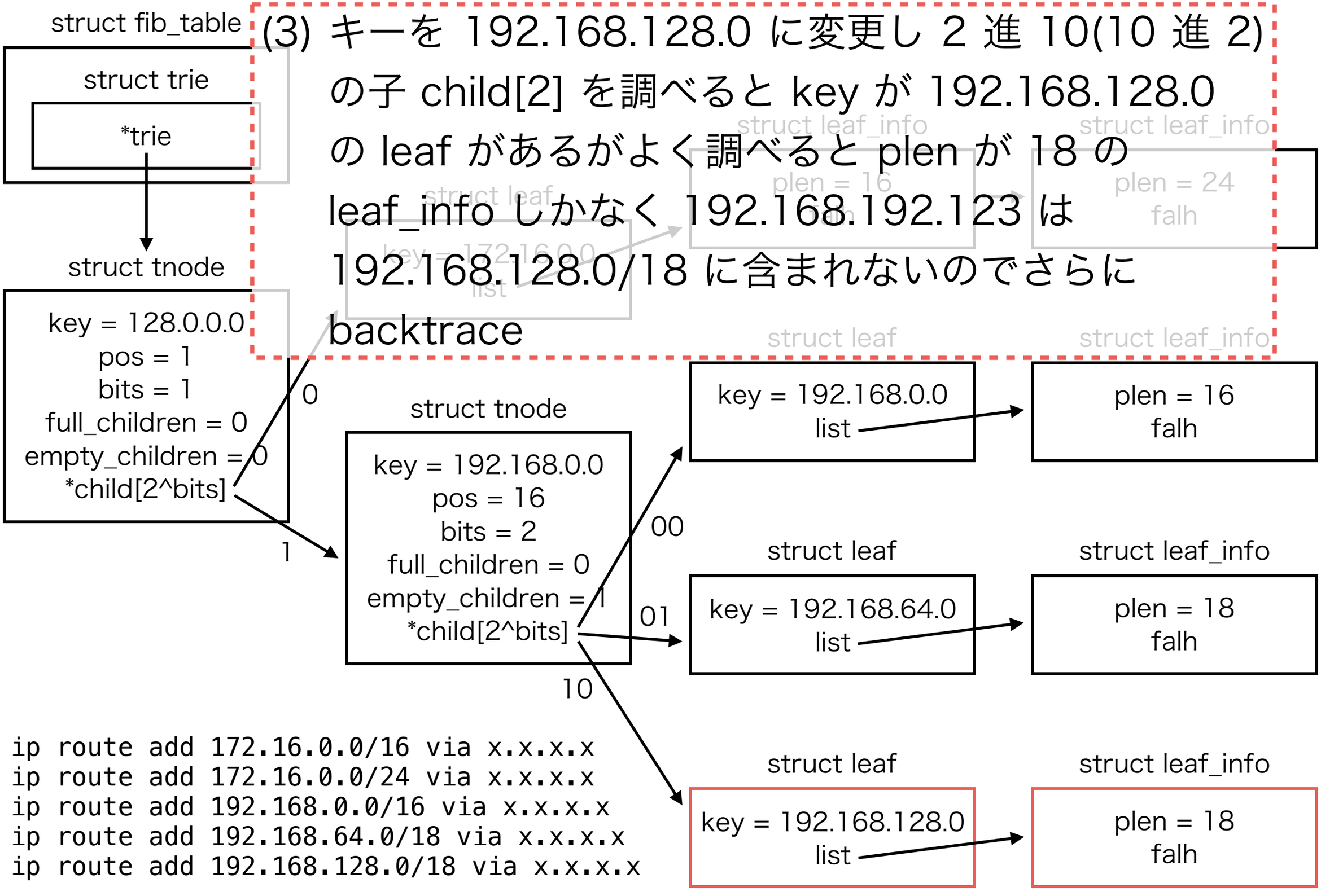
例) 宛先が 192.168.192.123 の場合の探索




```
ip route add 172.16.0.0/16 via x.x.x.x
ip route add 172.16.0.0/24 via x.x.x.x
ip route add 192.168.0.0/16 via x.x.x.x
ip route add 192.168.64.0/18 via x.x.x.x
ip route add 192.168.128.0/18 via x.x.x.x
```

例) 宛先が 192.168.192.123 の場合の探索

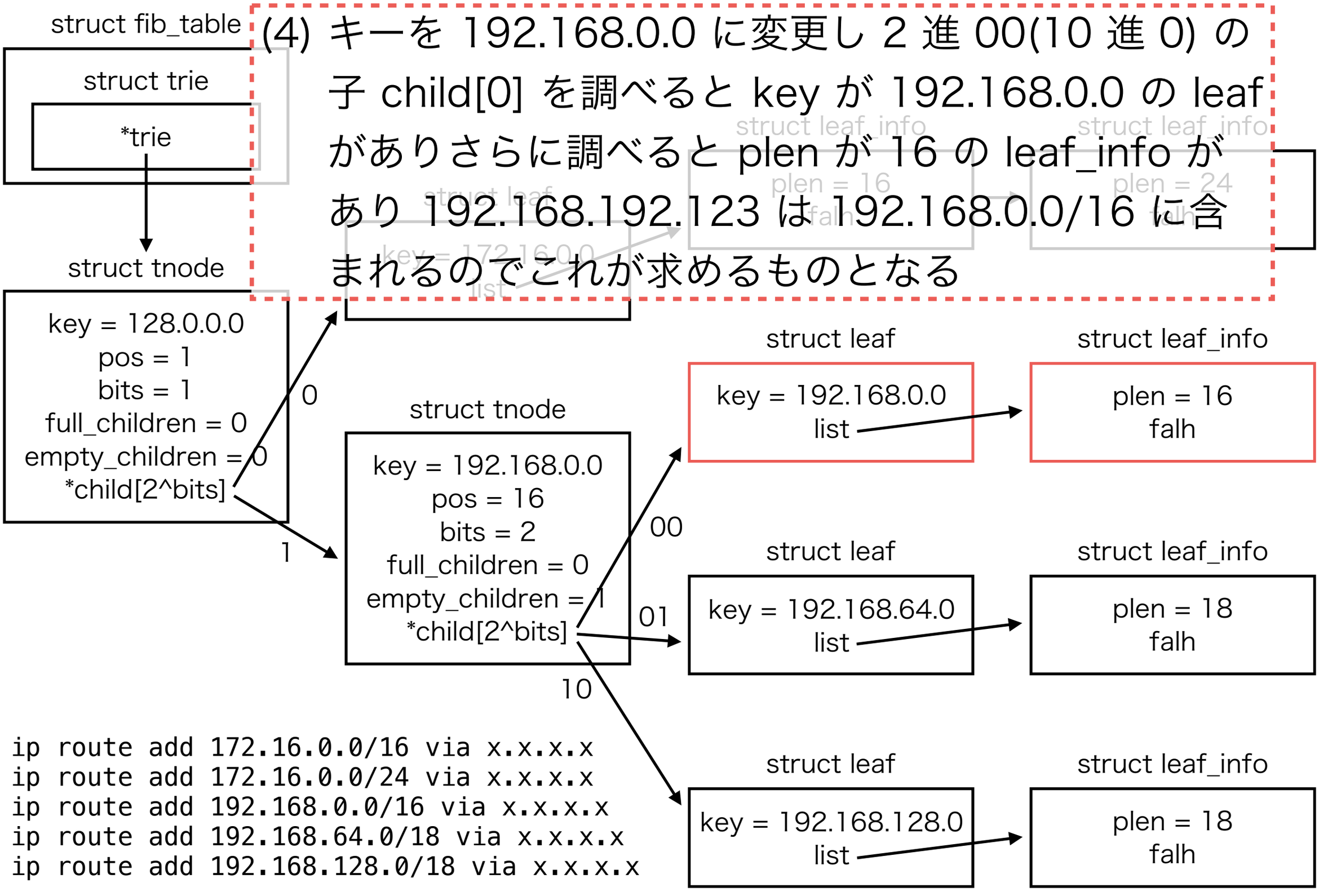
(3) キーを 192.168.128.0 に変更し 2 進 10(10 進 2) の子 child[2] を調べると key が 192.168.128.0 の leaf があるがよく調べると plen が 18 の leaf_info しかなく 192.168.192.123 は 192.168.128.0/18 に含まれないのでさらに backtrace




```
ip route add 172.16.0.0/16 via x.x.x.x
ip route add 172.16.0.0/24 via x.x.x.x
ip route add 192.168.0.0/16 via x.x.x.x
ip route add 192.168.64.0/18 via x.x.x.x
ip route add 192.168.128.0/18 via x.x.x.x
```

例) 宛先が 192.168.192.123 の場合の探索

(4) キーを 192.168.0.0 に変更し 2 進 00(10 進 0) の子 child[0] を調べると key が 192.168.0.0 の leaf がありさらに調べると plen が 16 の leaf_info があり 192.168.192.123 は 192.168.0.0/16 に含まれるのでこれが求めるものとなる

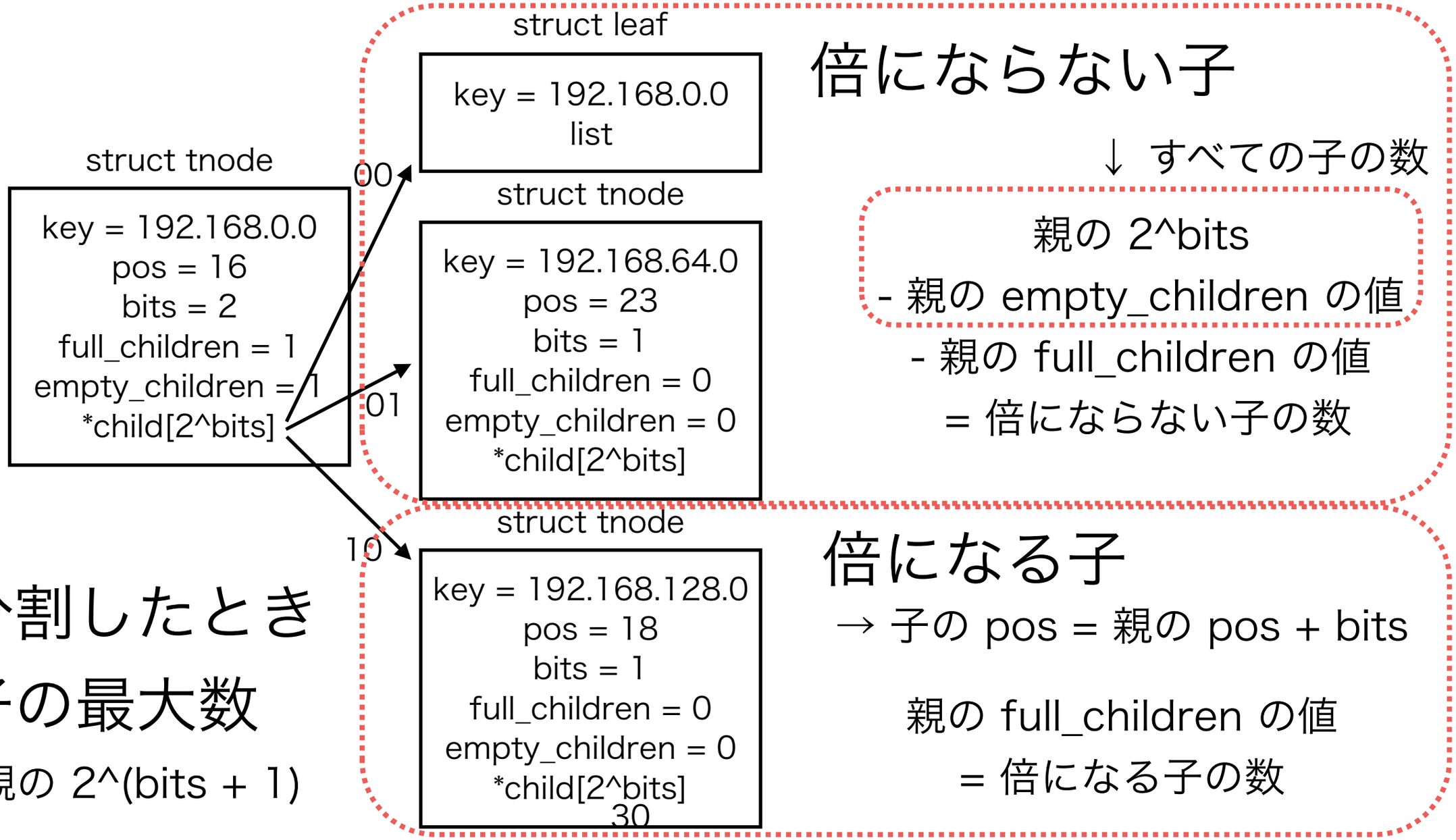



```
ip route add 172.16.0.0/16 via x.x.x.x
ip route add 172.16.0.0/24 via x.x.x.x
ip route add 192.168.0.0/16 via x.x.x.x
ip route add 192.168.64.0/18 via x.x.x.x
ip route add 192.168.128.0/18 via x.x.x.x
```

レベル圧縮をする条件

- 以下の条件を満たすときノードのレベル圧縮を実施

$$\frac{\text{倍にならない子の数} + \text{倍になる子の数}\times 2}{\text{仮に分割したときの子の最大数}} \geqq 0.5$$



目次

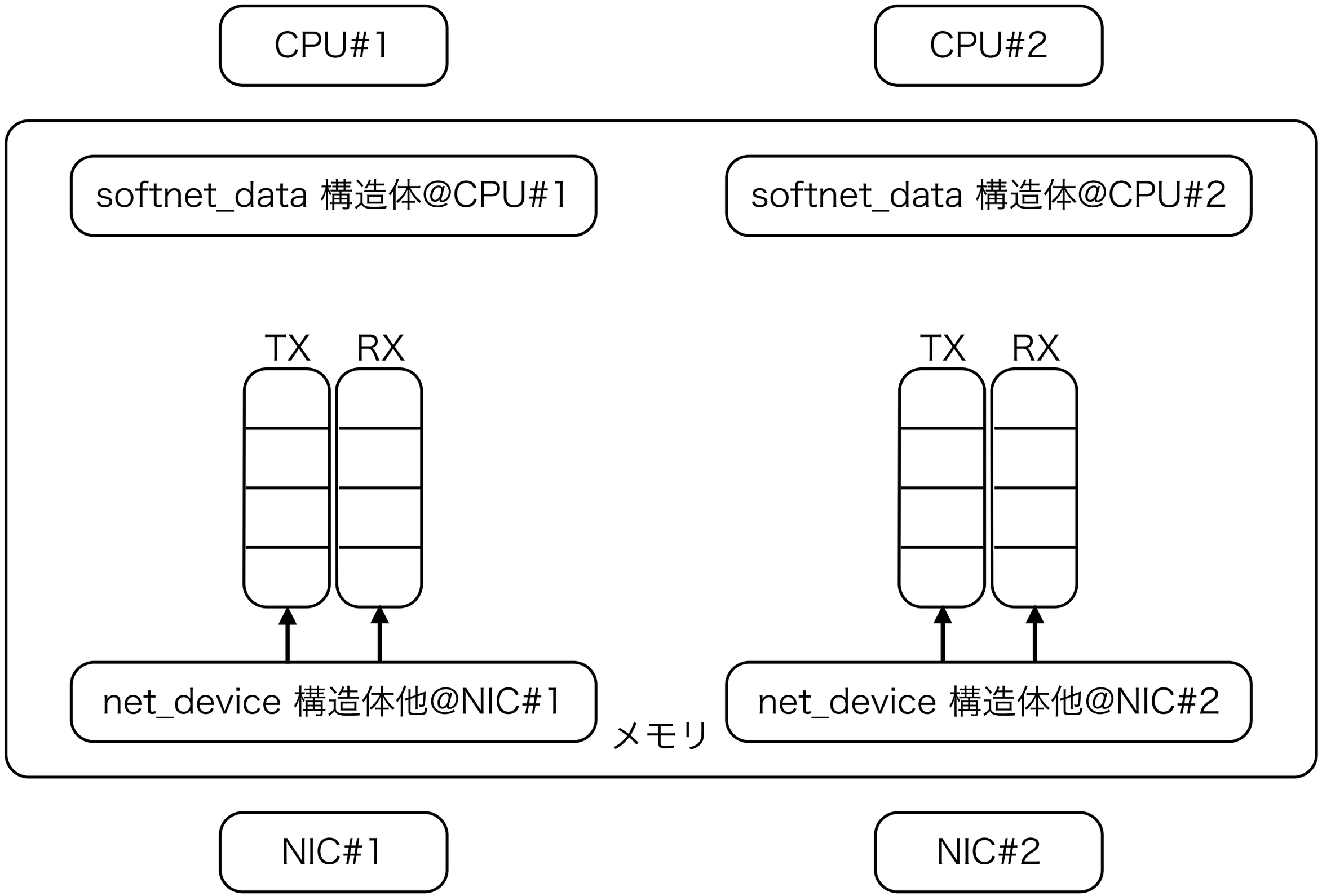
- 高速に経路探索を行うための Linux の実装
  - いろいろな木
  - Linux kernel の IPv4 ルーティングテーブル
- 大量のパケットを処理するための Linux の実装
  - Receive Livelock と Linux NAPI
  - MultiQueue NIC と Receive Side Scaling(RSS)
- まとめ

Linux NAPI

- 背景
  - デッド・ロックしているわけでもないのに割り込みが溢れてしまうことで割り込みハンドラしか実行していないような状態(Receive Livelock)になることが指摘されていた
  - 割り込み処理はかなり高い優先度を割り当てられているためパケット受信の際に何も考えず割り込みを発生させると他の処理を一切出来ない状態になる
- 仕組み
  - 頻繁にパケットが流れていないときは従来通り NIC からの割り込み通知で CPU が割り込みハンドラに移行し受信パケットを処理する
  - パケットを受信した際に割り込みを無効化し処理が完了するまではポーリング・モードで動作
  - 処理すべきパケットの処理が完了したら割り込みを有効化
- 歴史
  - 2002 年 11 月にリリースされた Linux カーネル 2.4.20 の tg3 ドライバにおいて実装され順次その他の NIC ドライバでも採用

※1) Eliminating Receive Livelock in an Interrupt-Driven Kernel
http://www.stanford.edu/class/cs240/readings/p217-mogul.pdf

Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
TX
RX
TX
RX
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

Linux NAPI

CPU#1

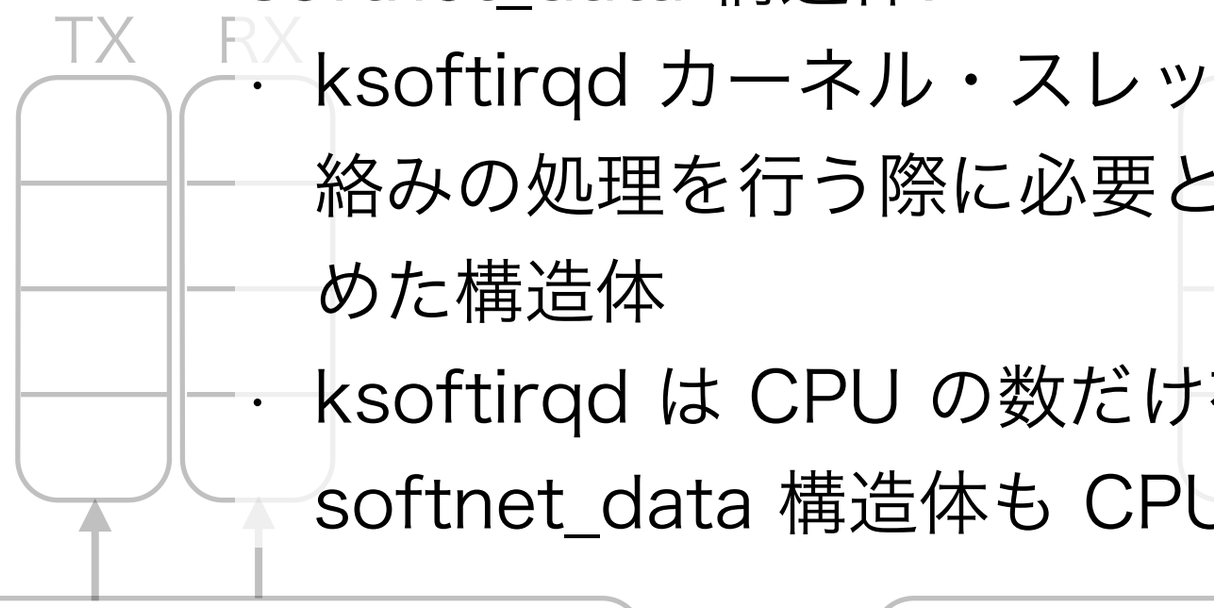
CPU#2

softnet_data 構造体@CPU#1

softnet_data 構造体@CPU#2

softnet_data 構造体:

- ksoftirqd カーネル・スレッドがネットワーク絡みの処理を行う際に必要とするデータをまとめた構造体
- ksoftirqd は CPU の数だけ存在するので softnet_data 構造体も CPU の数だけ存在する



net_device 構造体他@NIC#1

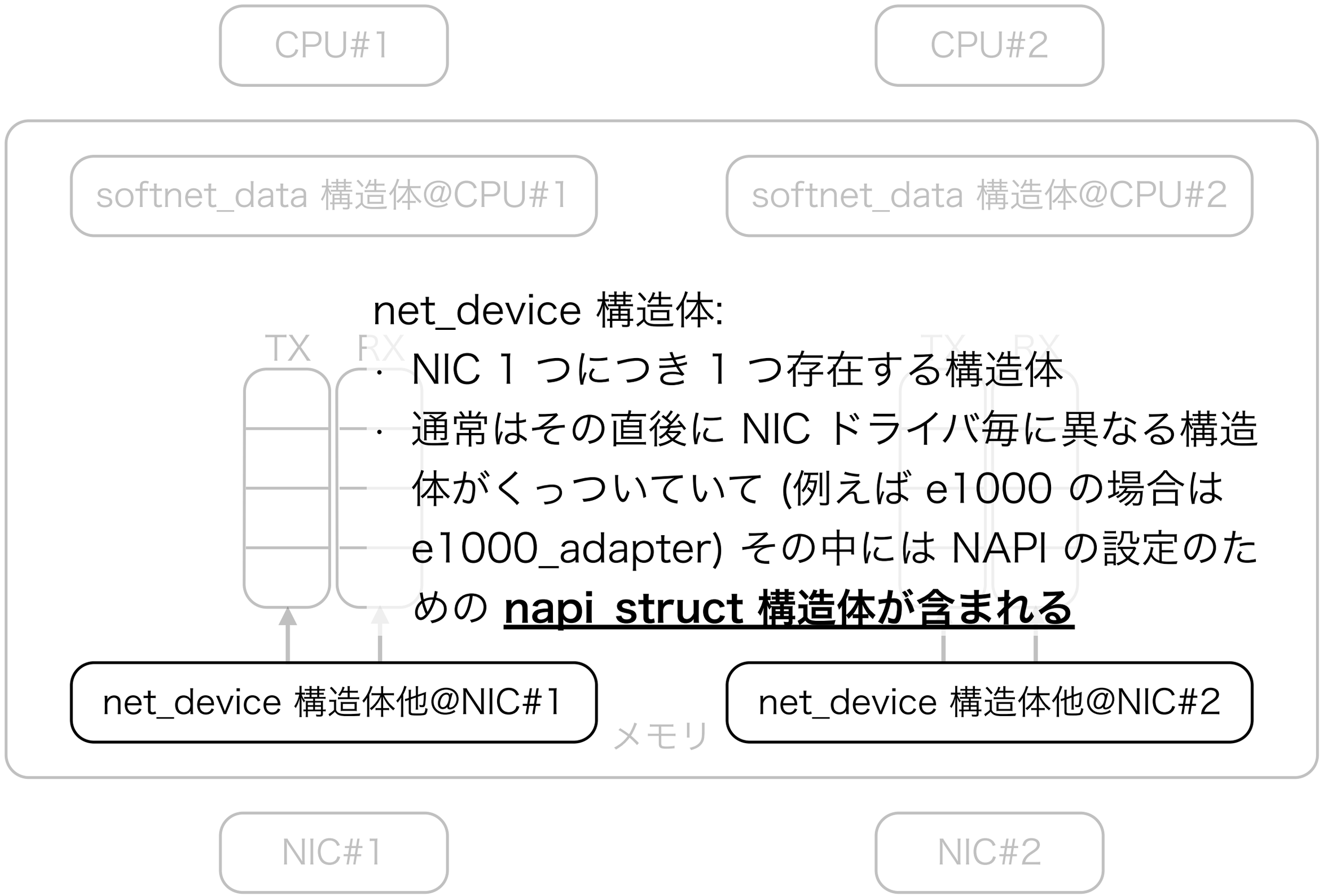
net_device 構造体他@NIC#2

メモリ

NIC#1

NIC#2

Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
net_device 構造体:
・ NIC 1 つにつき 1 つ存在する構造体
・ 通常はその直後に NIC ドライバ毎に異なる構造体がくっついていて (例えば e1000 の場合は e1000_adapter) その中には NAPI の設定のための napi_struct 構造体が含まれる
TX
RX
TX
RX
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

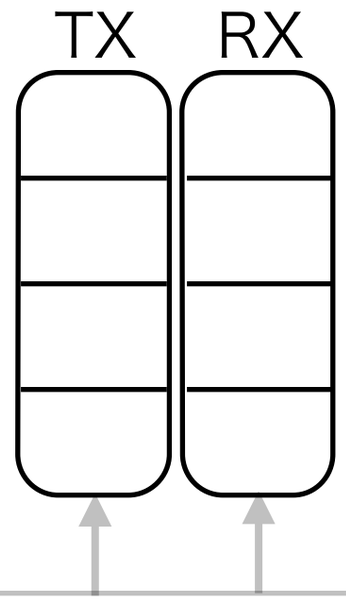
Linux NAPI

CPU#1

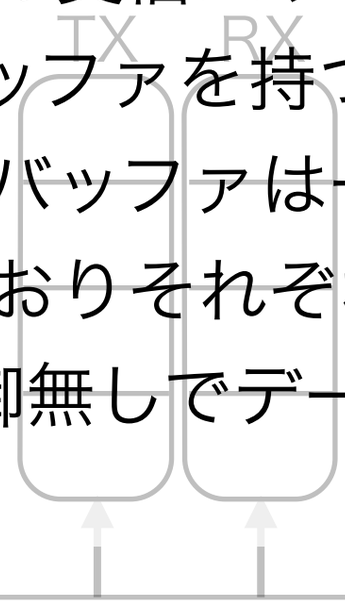
CPU#2

softnet_data 構造体@CPU#1

softnet_data 構造体@CPU#2



- NIC は受信 (RX) 用の受信バッファと送信 (TX) 用の送信バッファを持つ
- 受信バッファと送信バッファは一般的にリング構造をとっておりそれぞれ NIC と CPU の間で排他制御無しでデータのやり取りを行う



net_device 構造体他@NIC#1

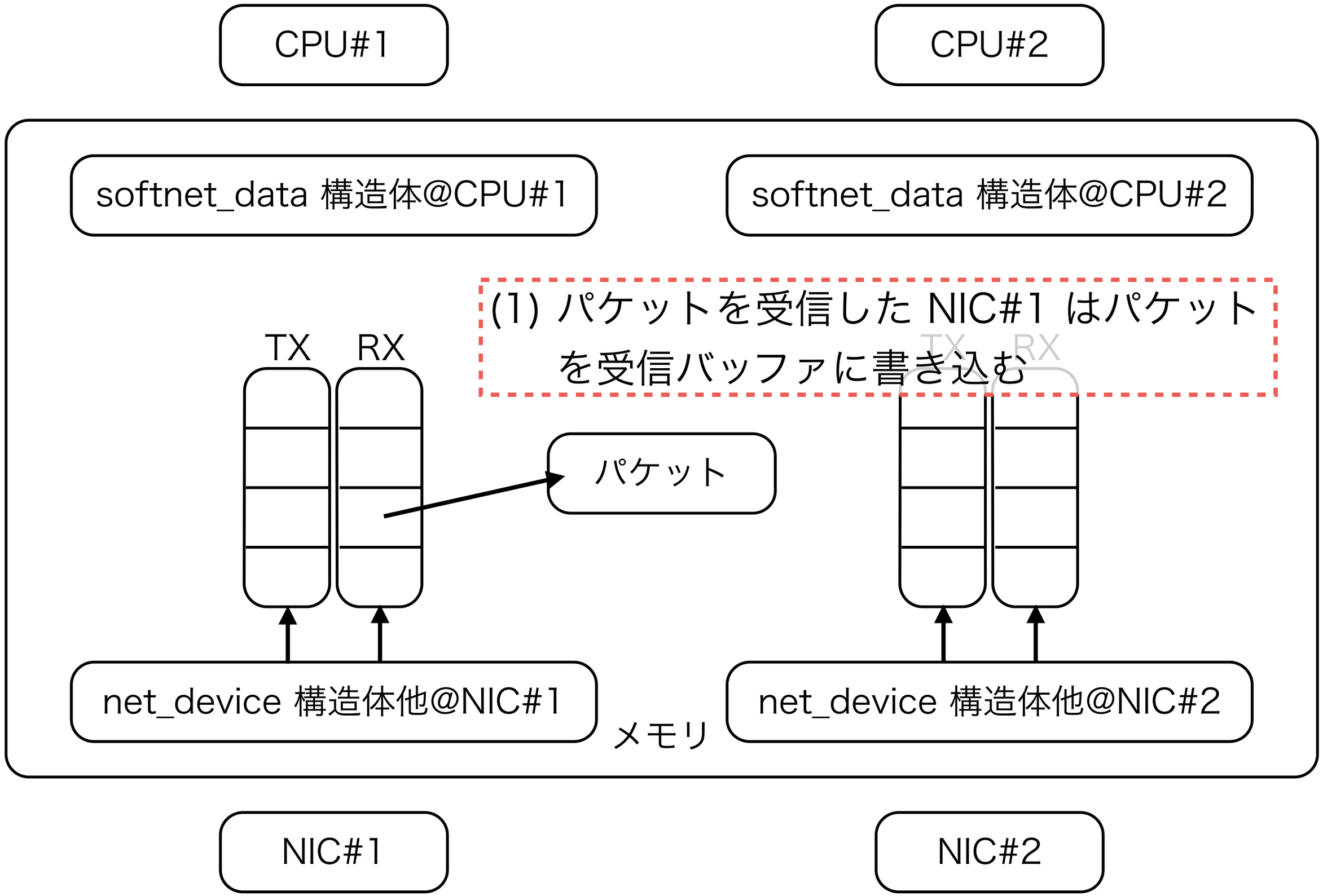
メモリ

net_device 構造体他@NIC#2

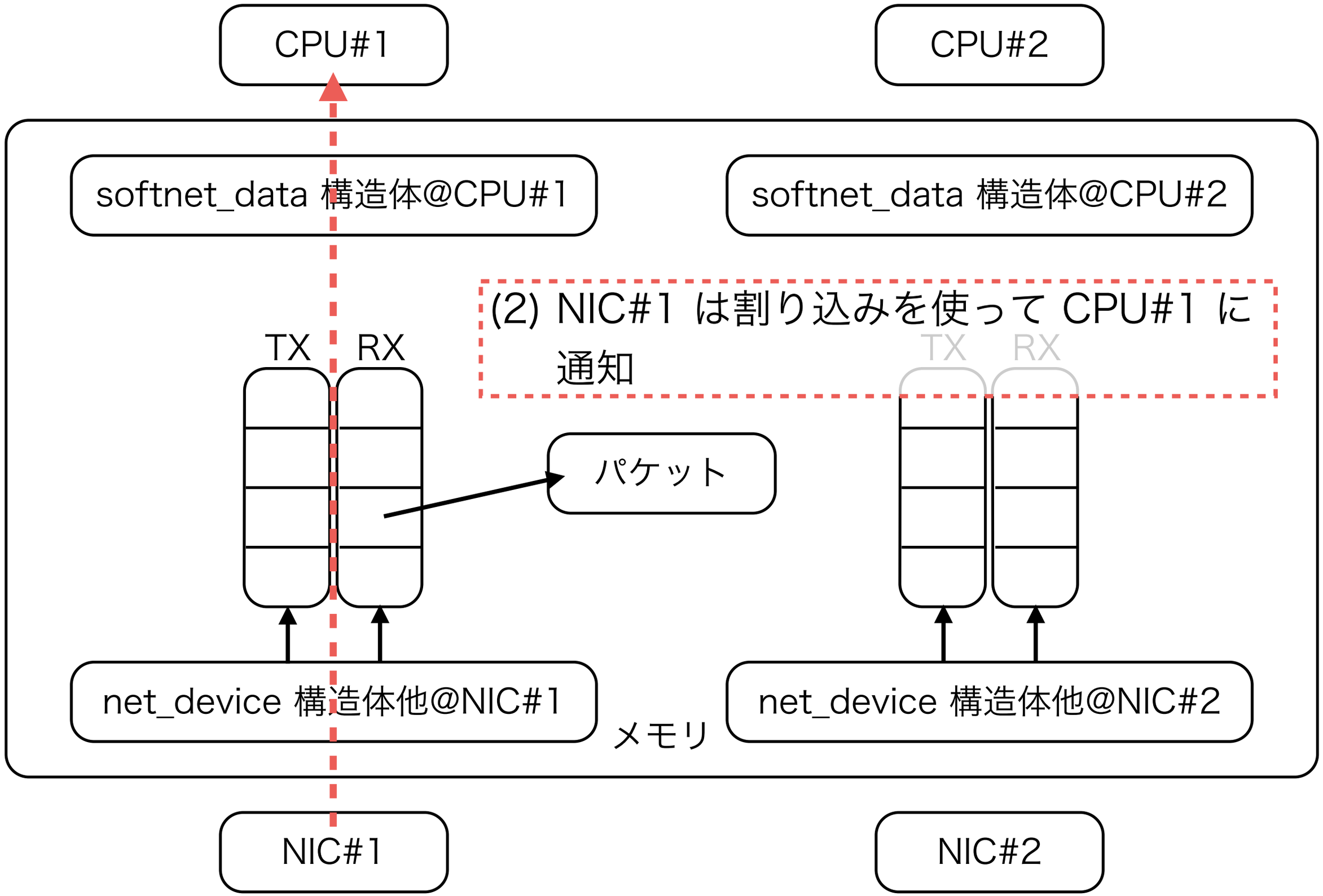
NIC#1

NIC#2

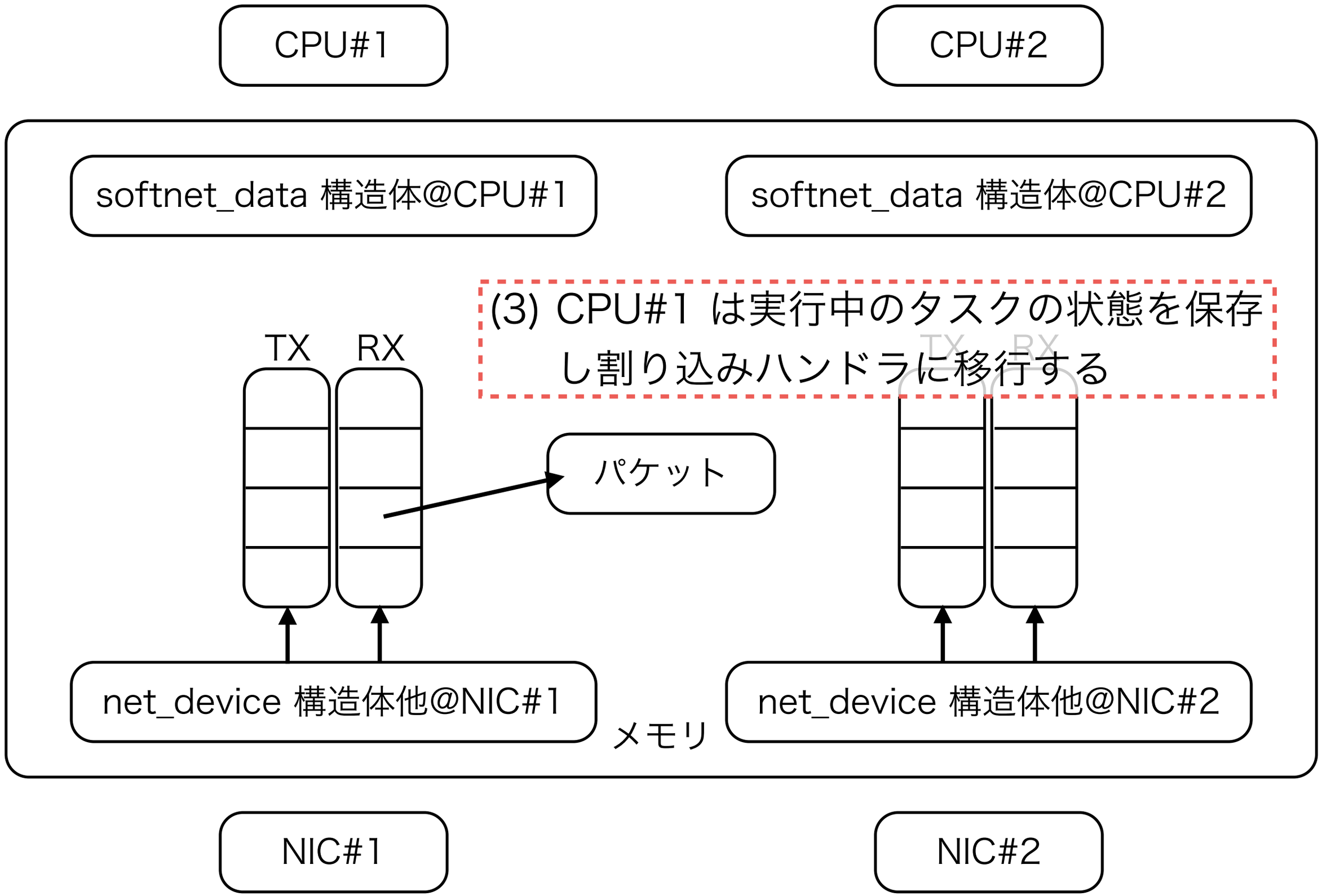
Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
(1) パケットを受信した NIC#1 はパケットを受信バッファに書き込む
TX
RX
TX
RX
パケット
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

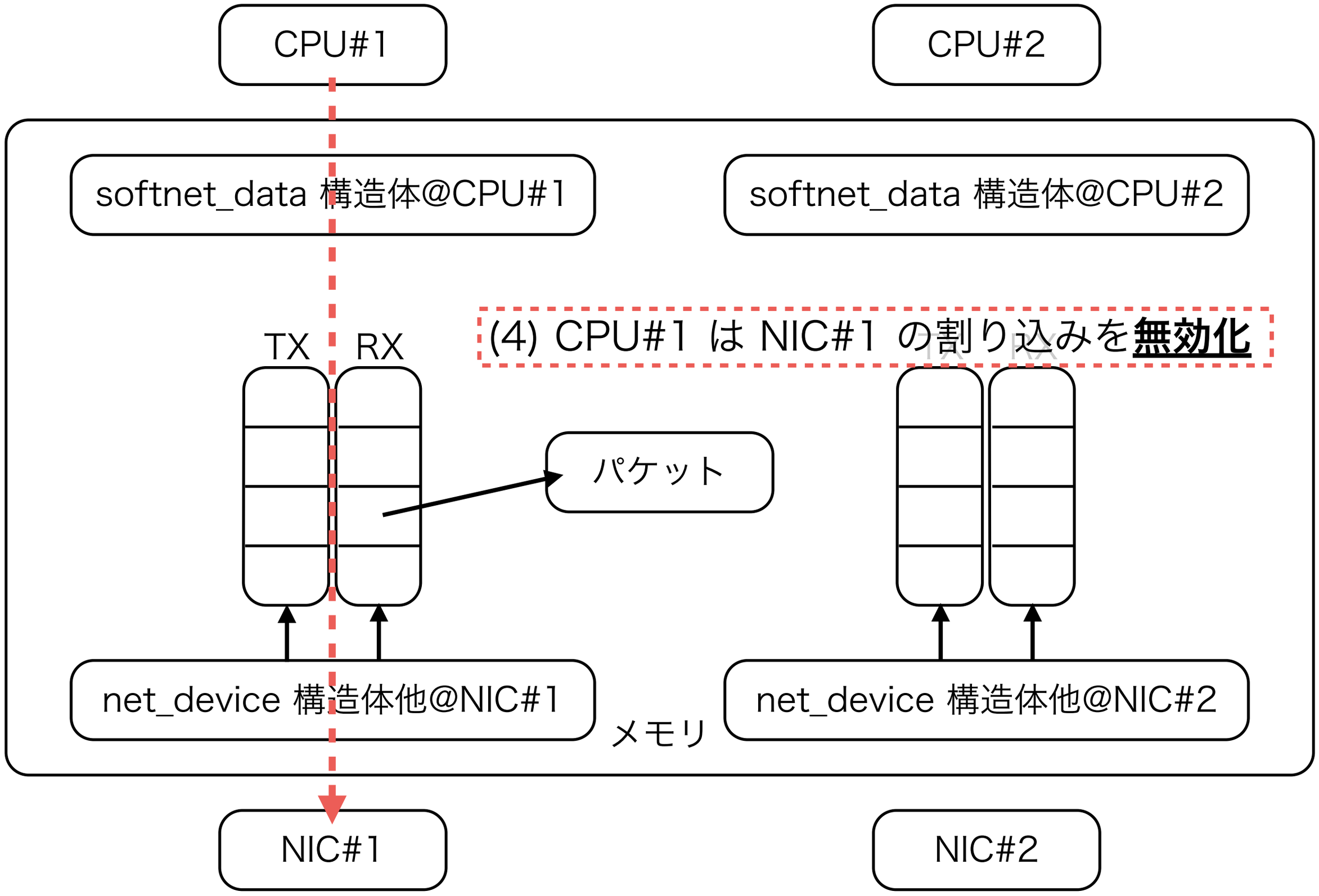
Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
(2) NIC#1 は割り込みを使って CPU#1 に通知
TX
RX
TX
RX
パケット
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

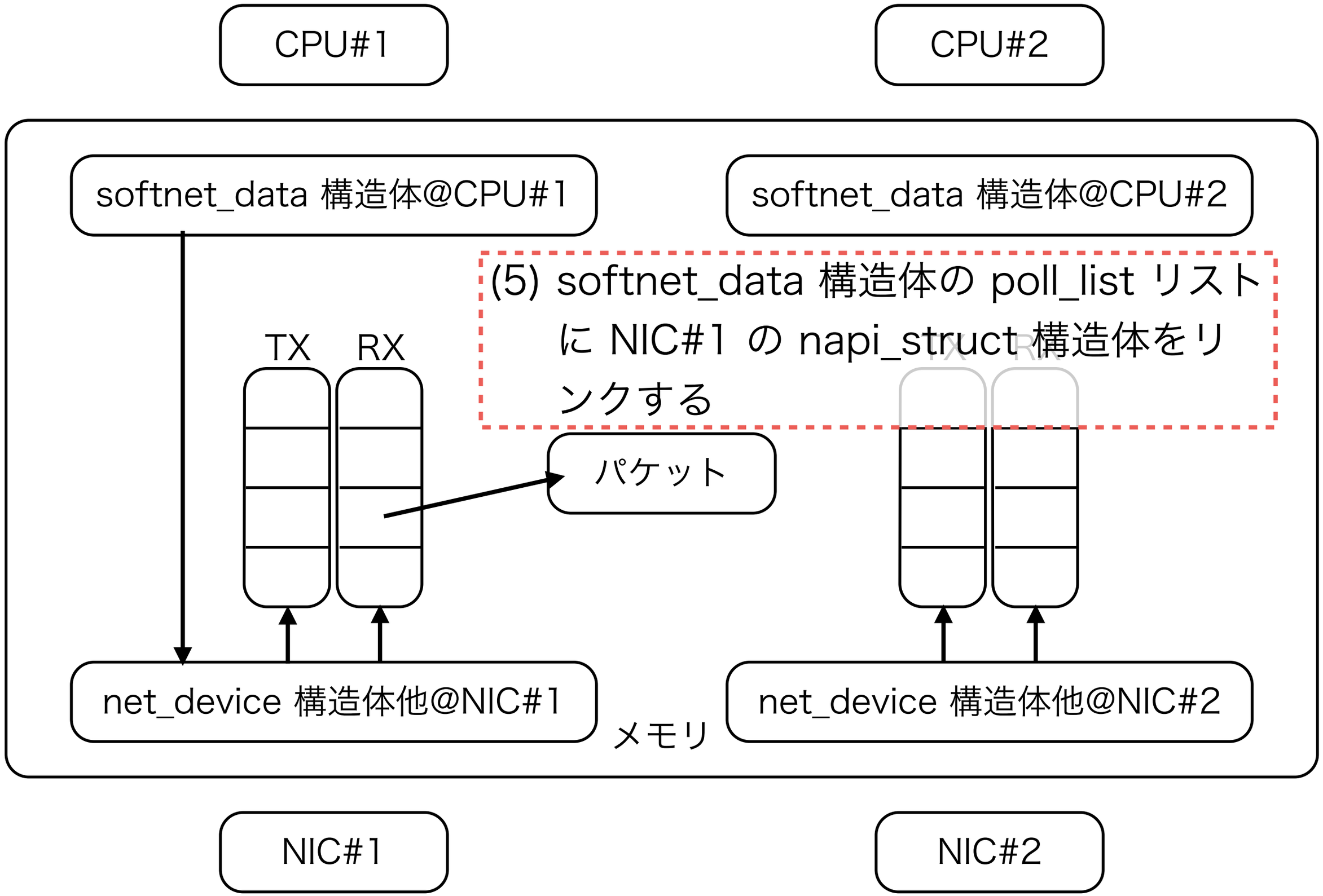
Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
(3) CPU#1 は実行中のタスクの状態を保存し割り込みハンドラに移行する
TX
RX
パケット
TX
RX
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

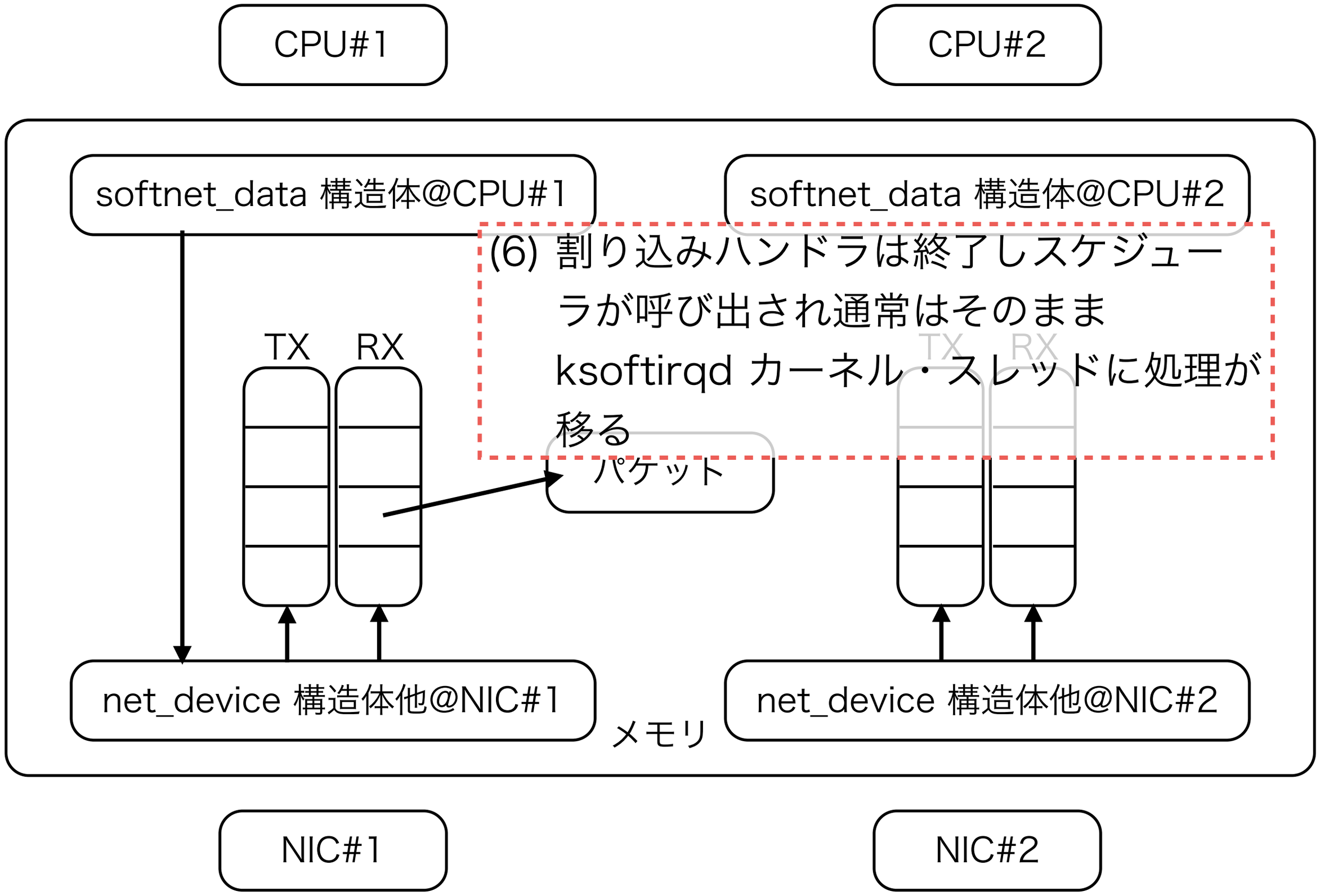
Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
TX
RX
(4) CPU#1 は NIC#1 の割り込みを無効化
TX
RX
パケット
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

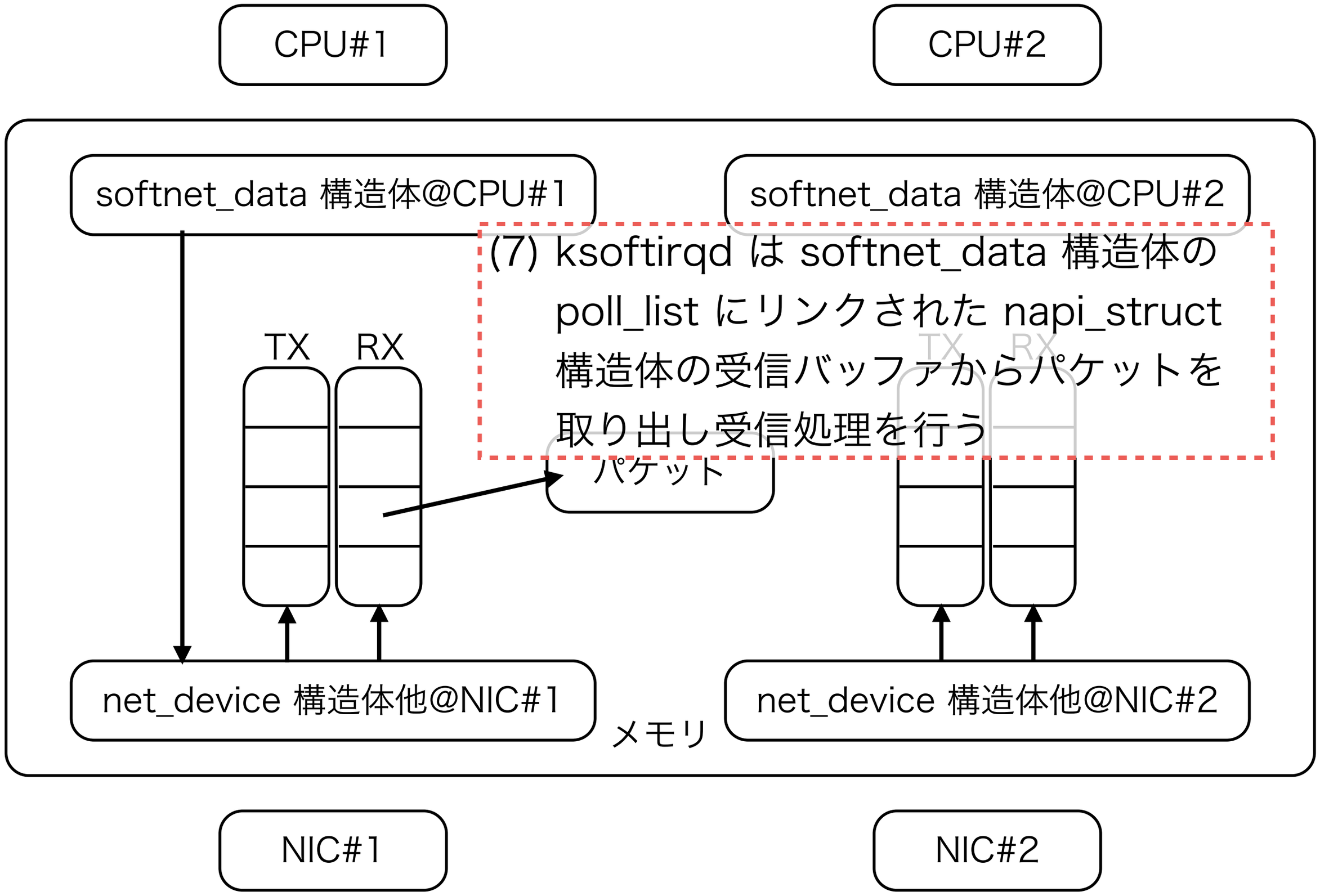
Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
(5) softnet_data 構造体の poll_list リストに NIC#1 の napi_struct 構造体をリンクする
TX
RX
パケット
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

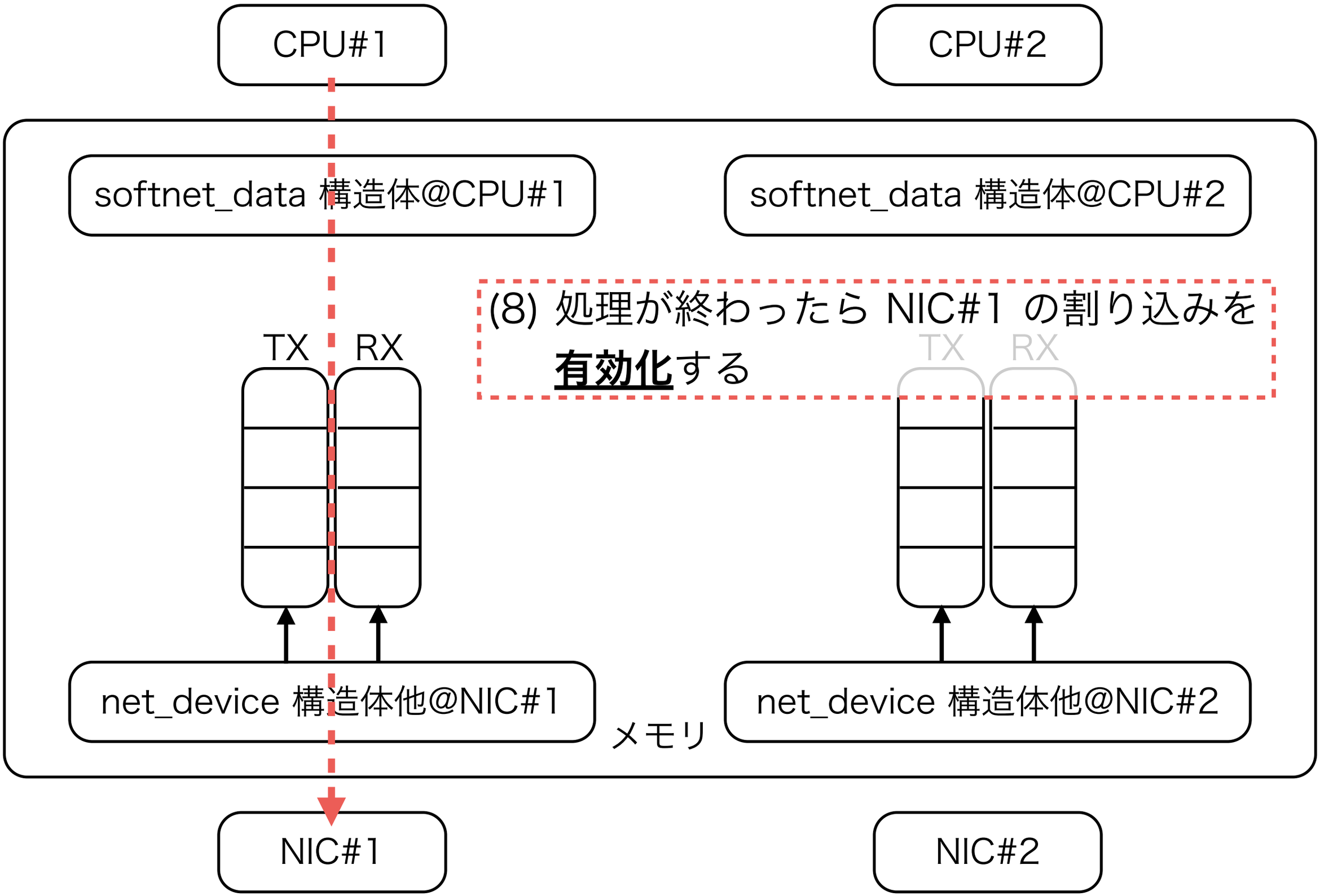
Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
(6) 割り込みハンドラは終了しスケジューラが呼び出され通常はそのままksoftirqd カーネル・スレッドに処理が移る
TX
RX
TX
RX
パケット
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
(7) ksoftirqd は softnet_data 構造体の poll_list にリンクされた napi_struct 構造体の受信バッファからパケットを取り出し受信処理を行う
TX
RX
パケット
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

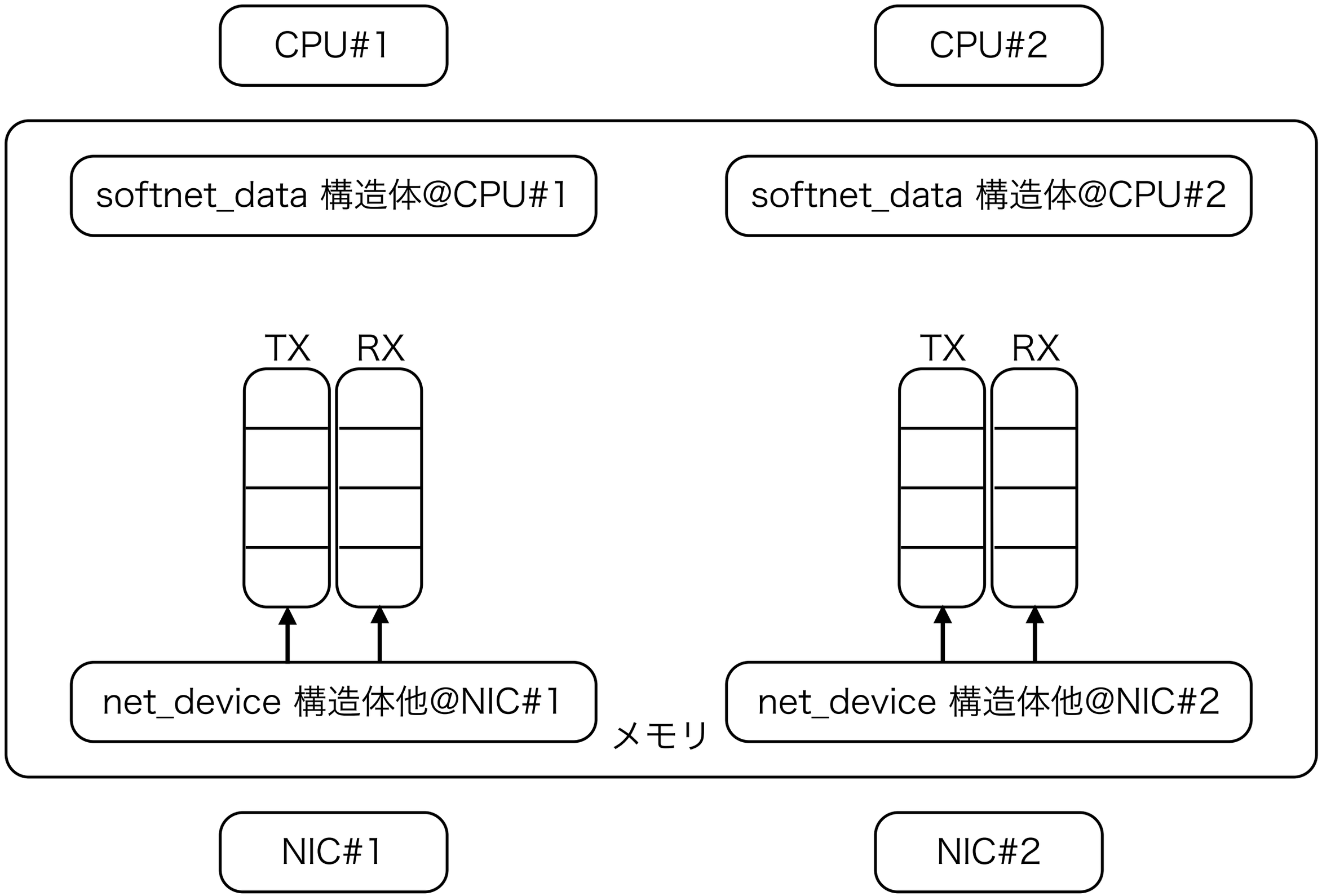
Linux NAPI


CPU#1
CPU#2
softnet_data 構造体@CPU#1
softnet_data 構造体@CPU#2
(8) 処理が終わったら NIC#1 の割り込みを
有効化する
TX
RX
TX
RX
net_device 構造体他@NIC#1
net_device 構造体他@NIC#2
メモリ
NIC#1
NIC#2

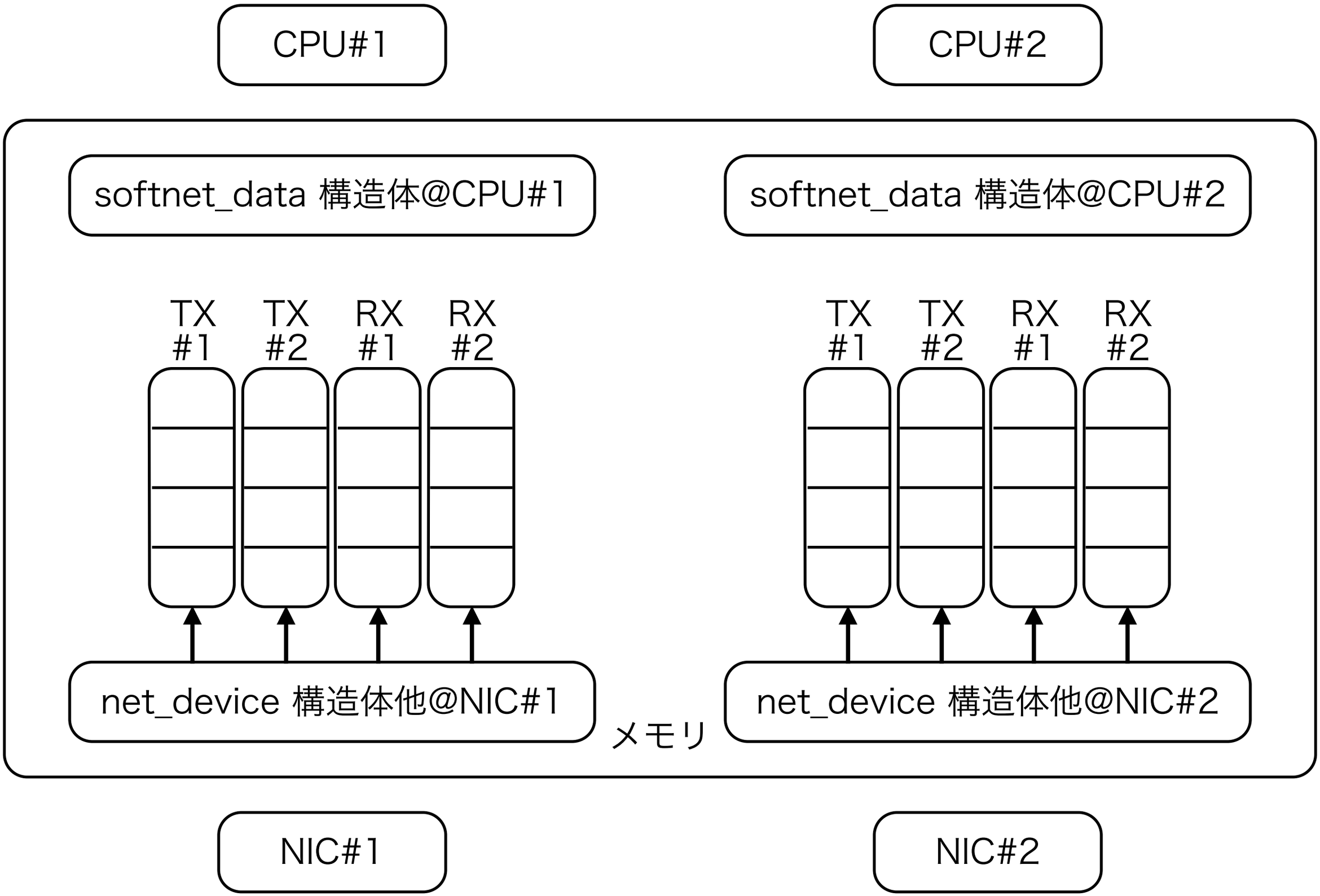
Receive Side Scaling (RSS)

- 背景
  - 最近のシステムでは CPU コアを複数持っているものが当たり前となっており ネットワークの処理もそれらで並列実行したい
- 仕組み
  - NIC 側で複数のキューを持たせる
  - パケットの受信時にヘッダ情報からキューを選択し対応する CPU に割り込みする
  - 同一フローのパケット (IP アドレスと TCP/UDP ポート番号が同じパケット) は同じキューが選択されるためリオーダは生じない
  - どの情報からキューを選択するかは変更可能
- 前提
  - PCI-Express バスに接続された MultiQueue に対応した NIC
    - PCI-Express で導入された MSI/MSI-X を用いて NIC から割り込みのルーティングする必要があるため
- 歴史
  - 2007 年 4 月にリリースされた Linux カーネル 2.6.21 の cxgb3 ドライバにおいてサポートされ順次その他の(主に 10G の)NIC ドライバでも採用

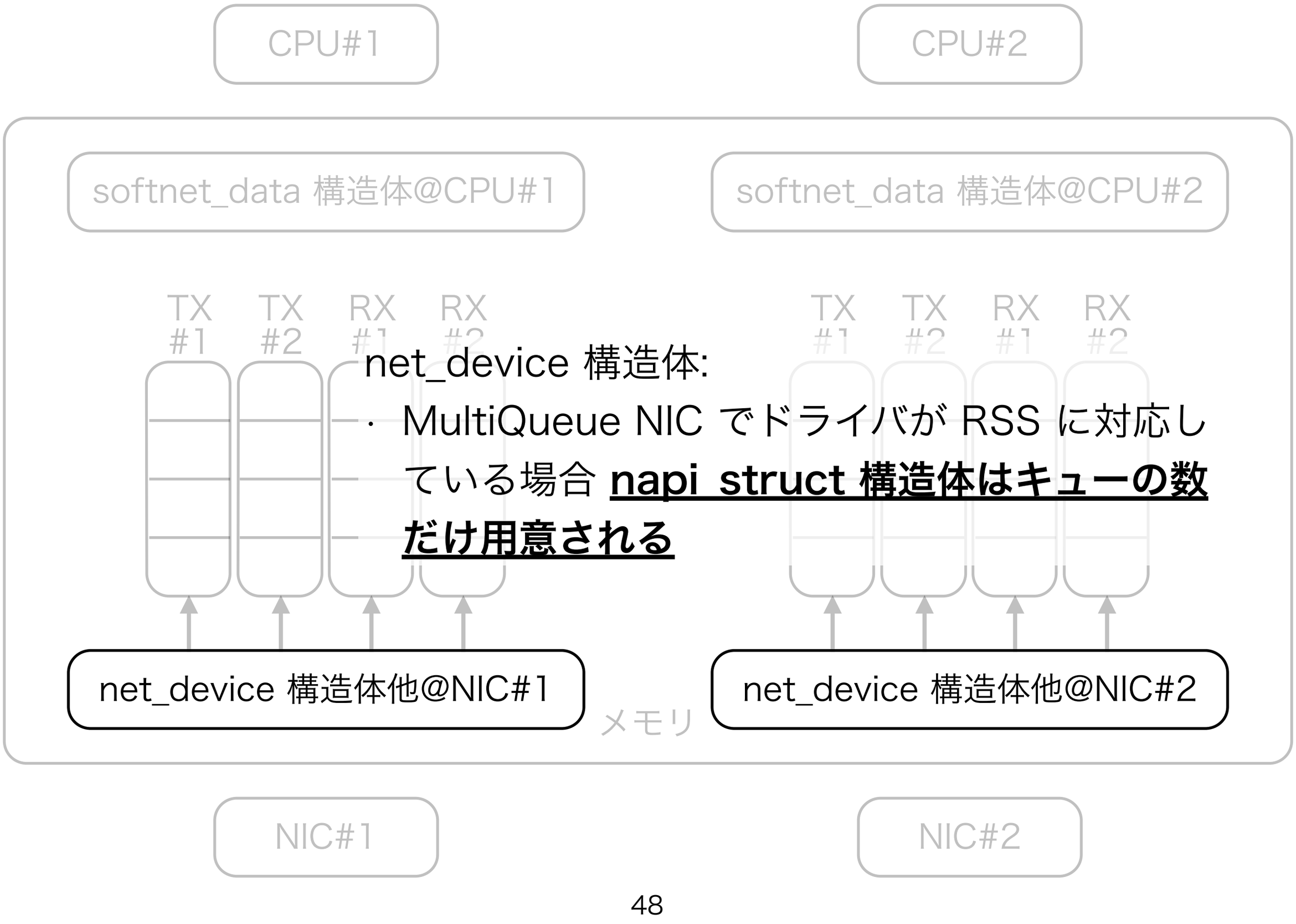
Receive Side Scaling (RSS)



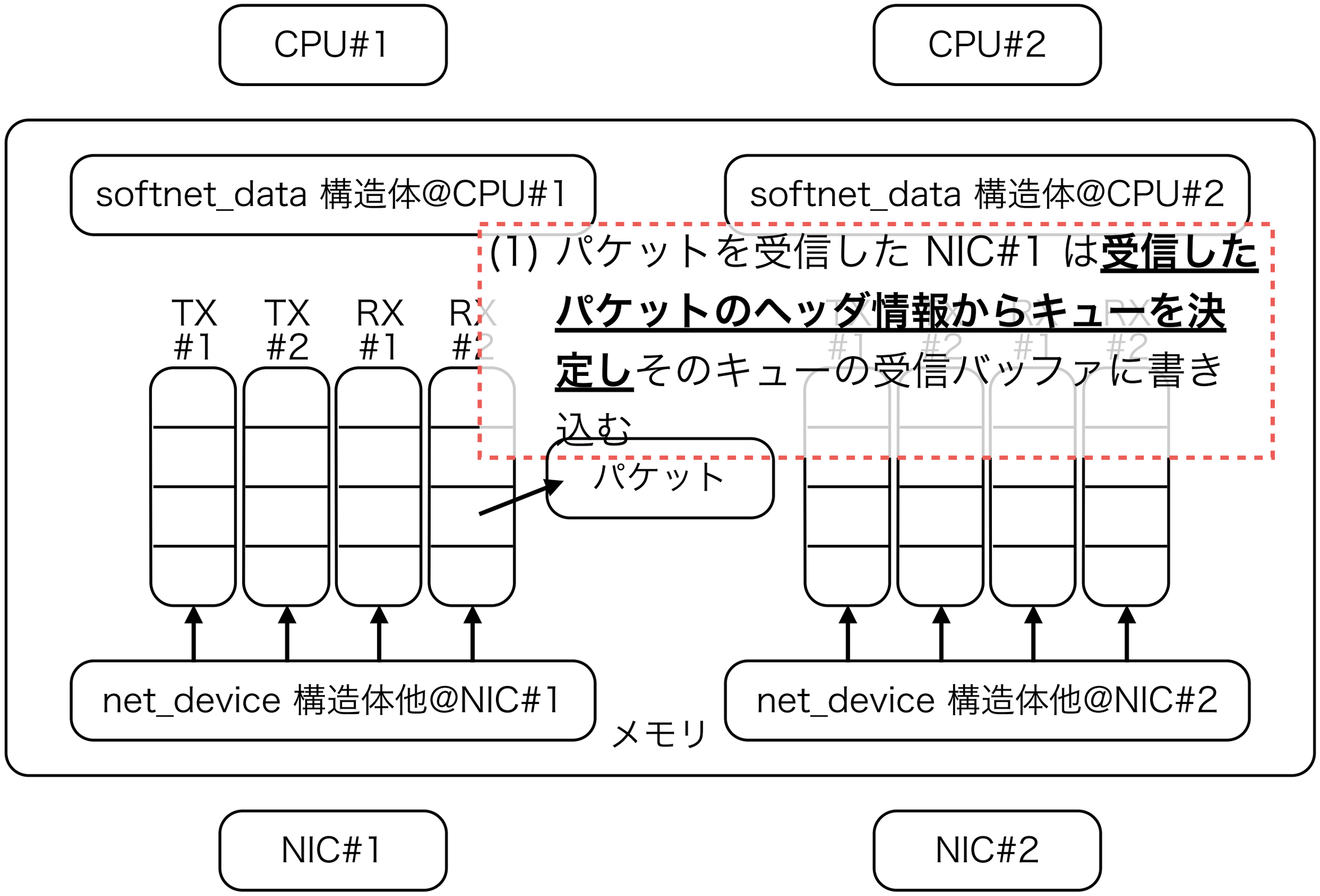
Receive Side Scaling (RSS)



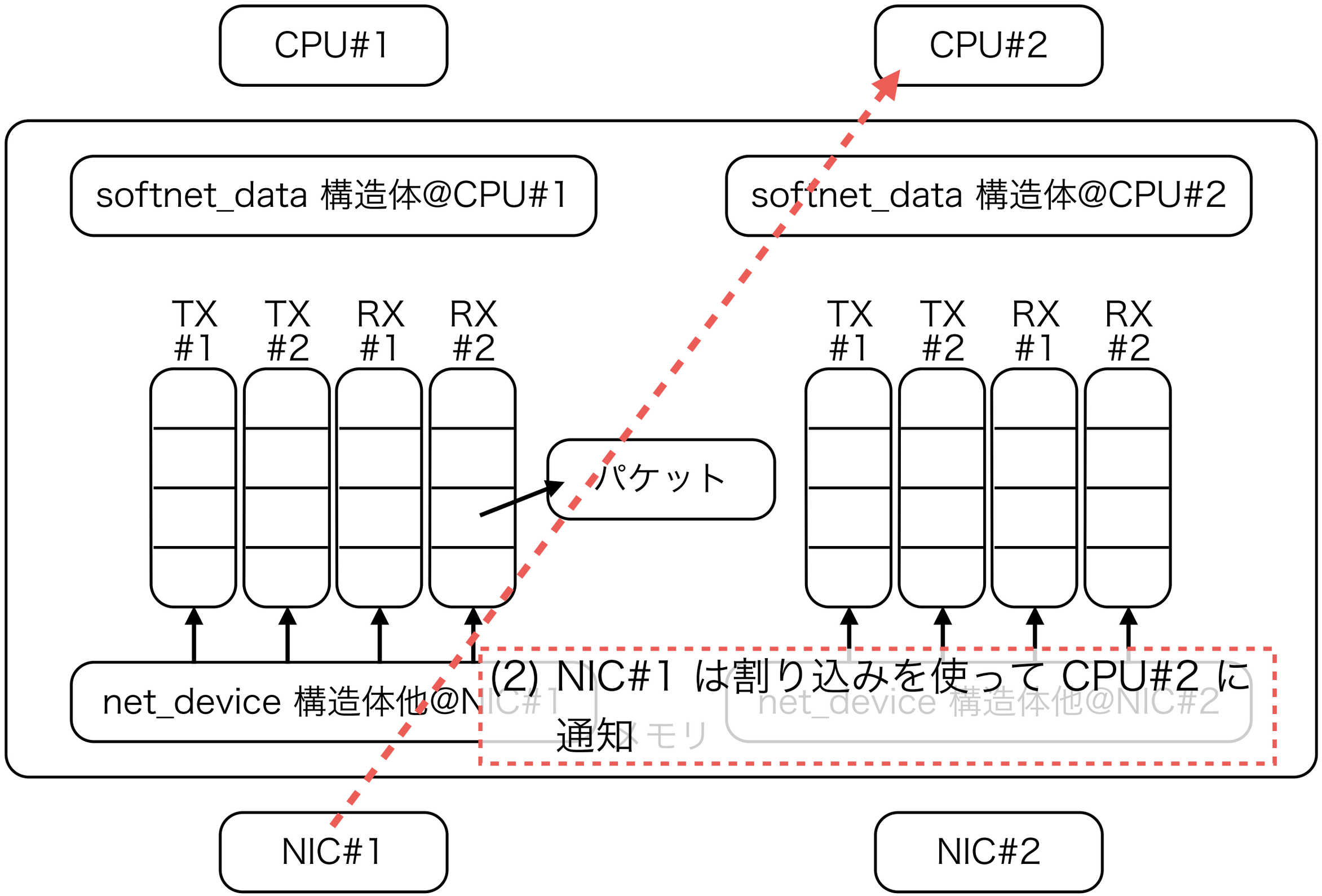
Receive Side Scaling (RSS)



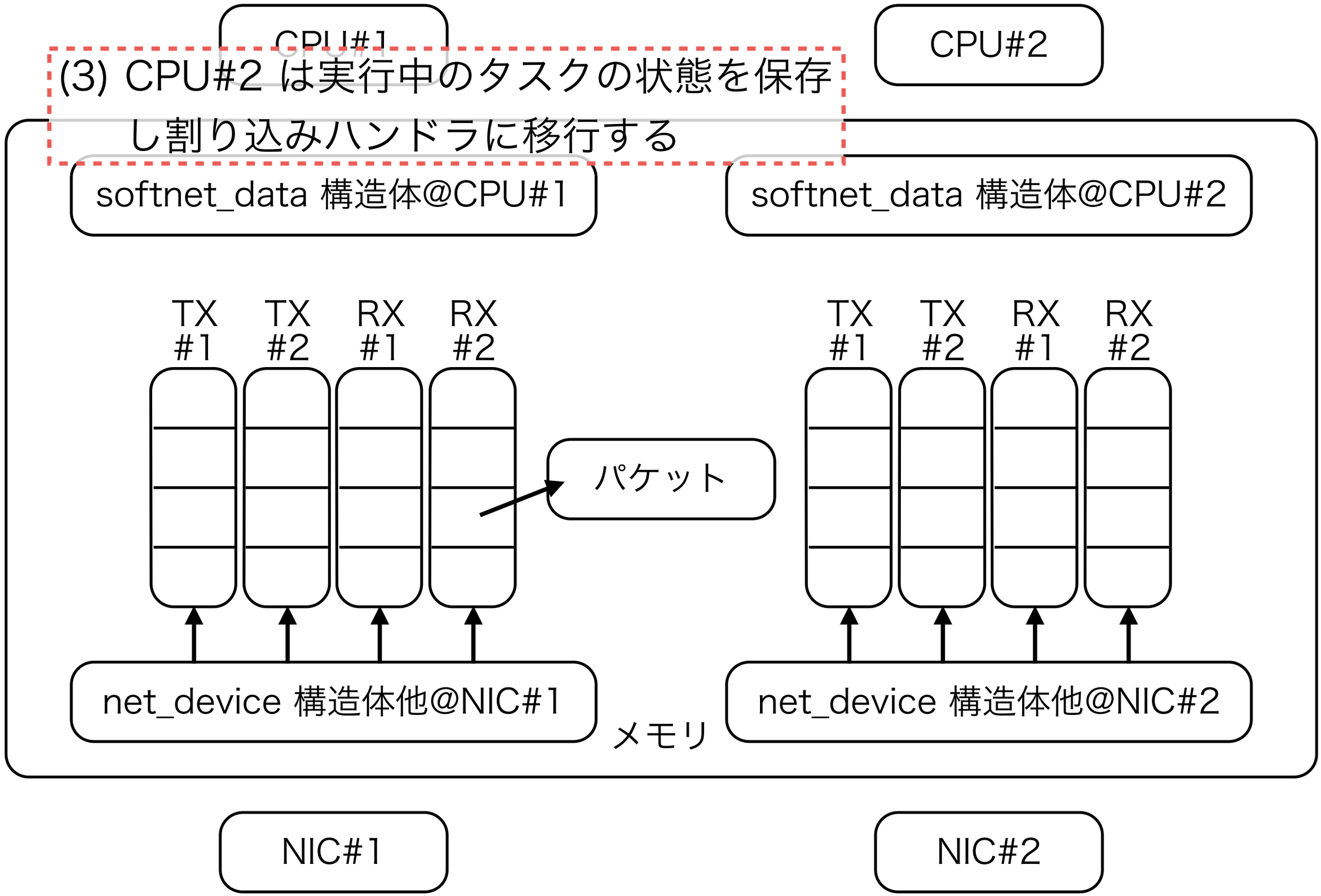
Receive Side Scaling (RSS)



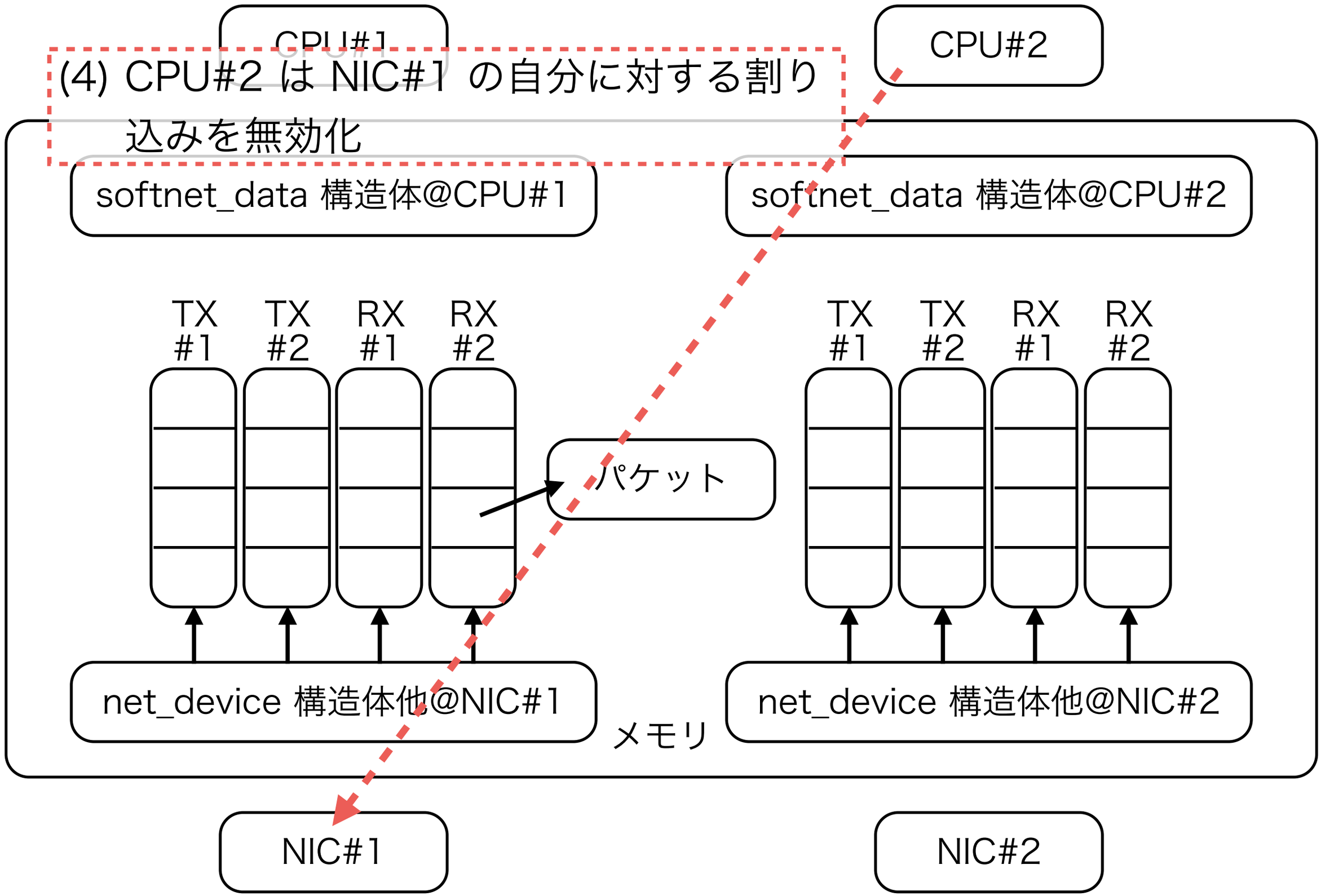
Receive Side Scaling (RSS)



Receive Side Scaling (RSS)

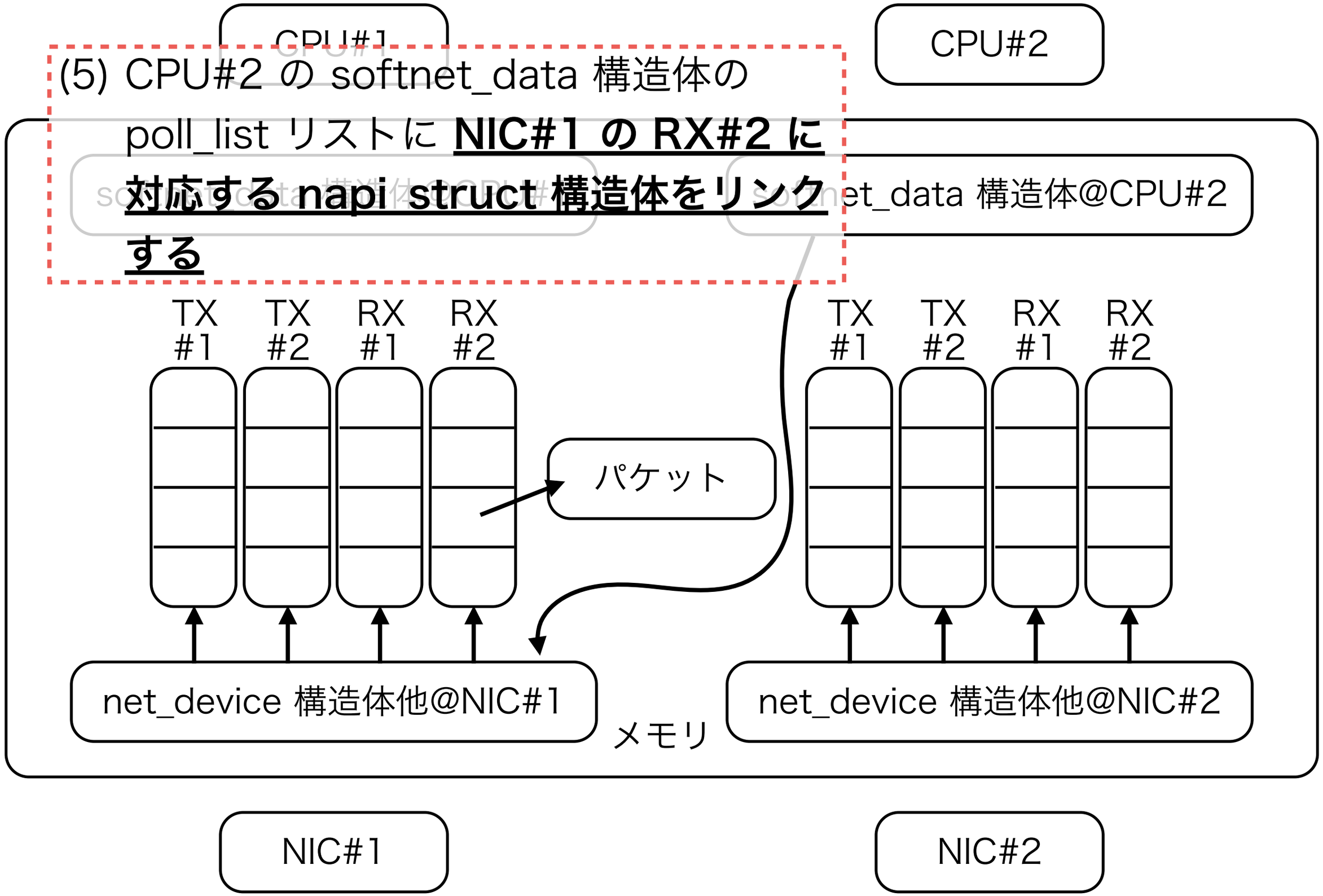


Receive Side Scaling (RSS)



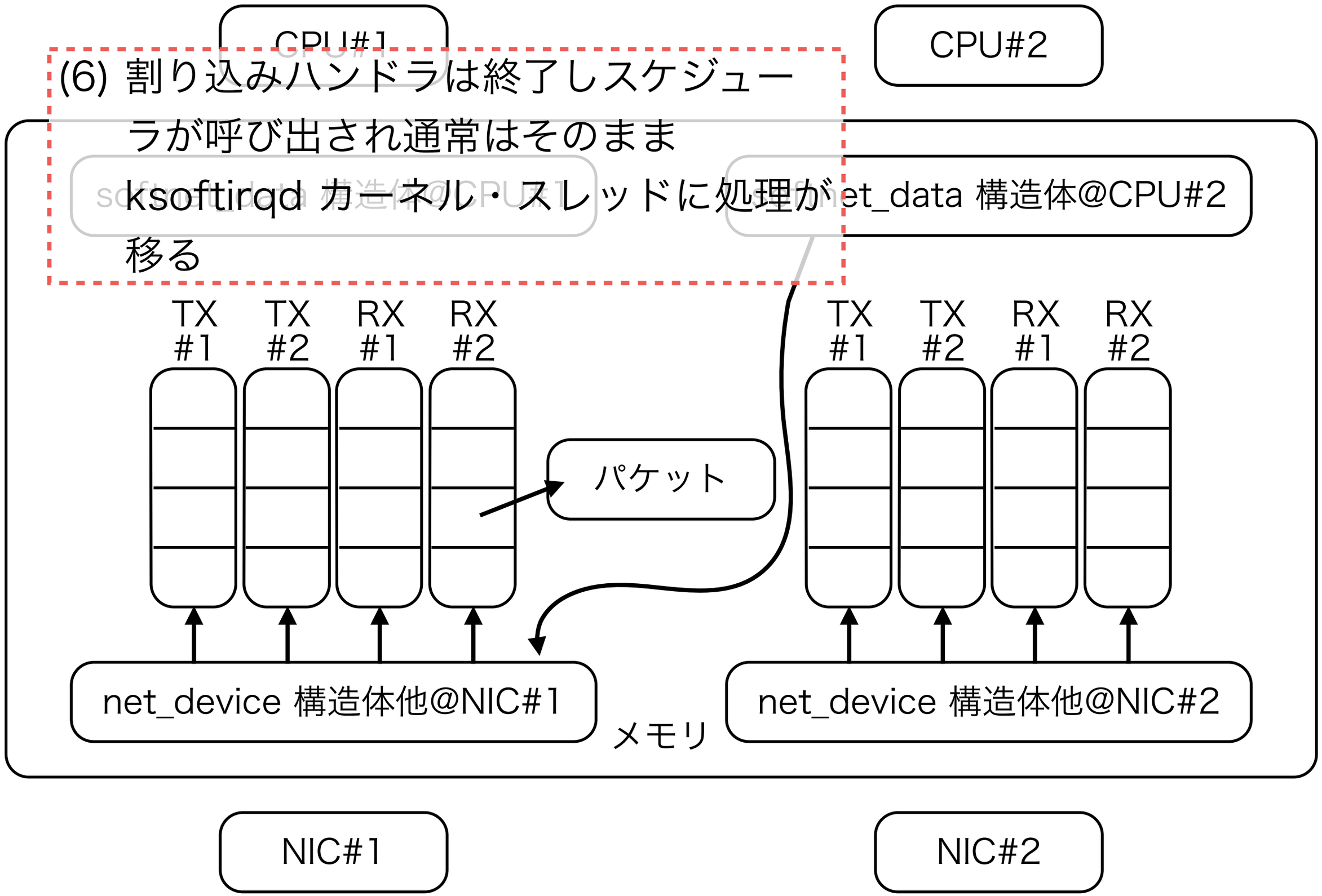
Receive Side Scaling (RSS)

(5) CPU#2 の softnet_data 構造体の poll_list リストに **NIC#1 の RX#2 に対応する napi struct 構造体をリンクする**



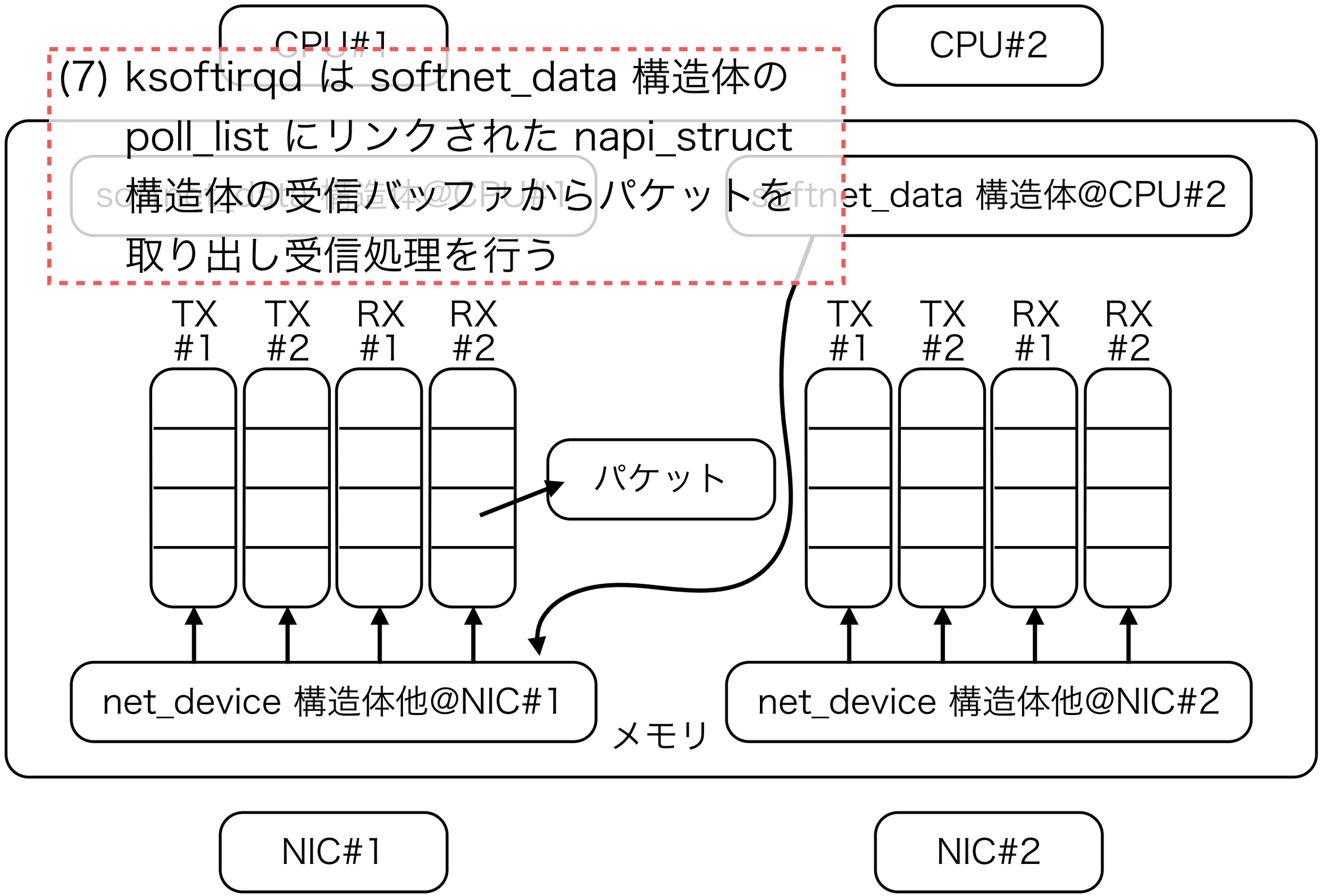
Receive Side Scaling (RS)

(6) 割り込みハンドラは終了しスケジューラが呼び出され通常はそのまま ksoftirqd カーネル・スレッドに処理が移る

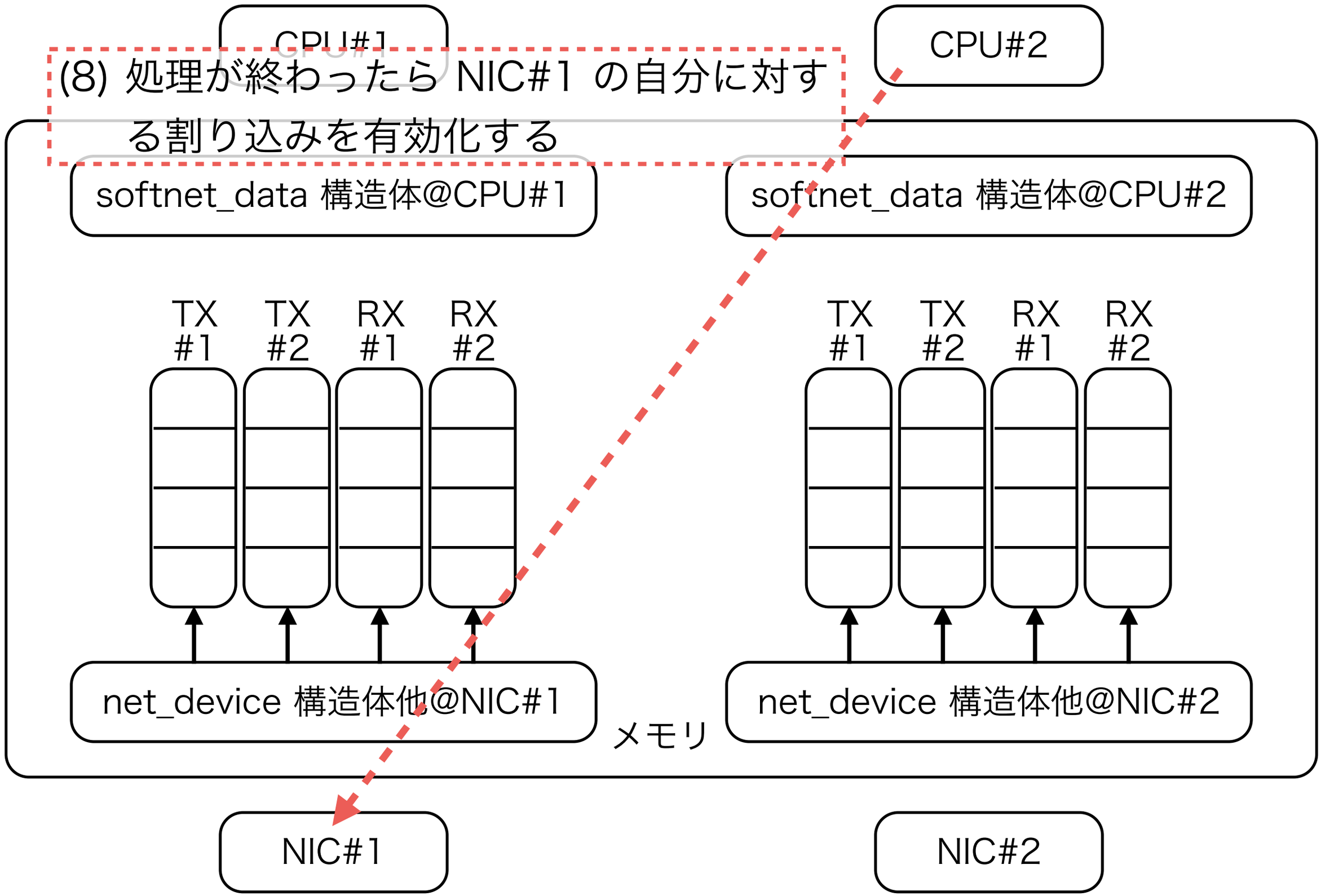


Receive Side Scaling (RS)

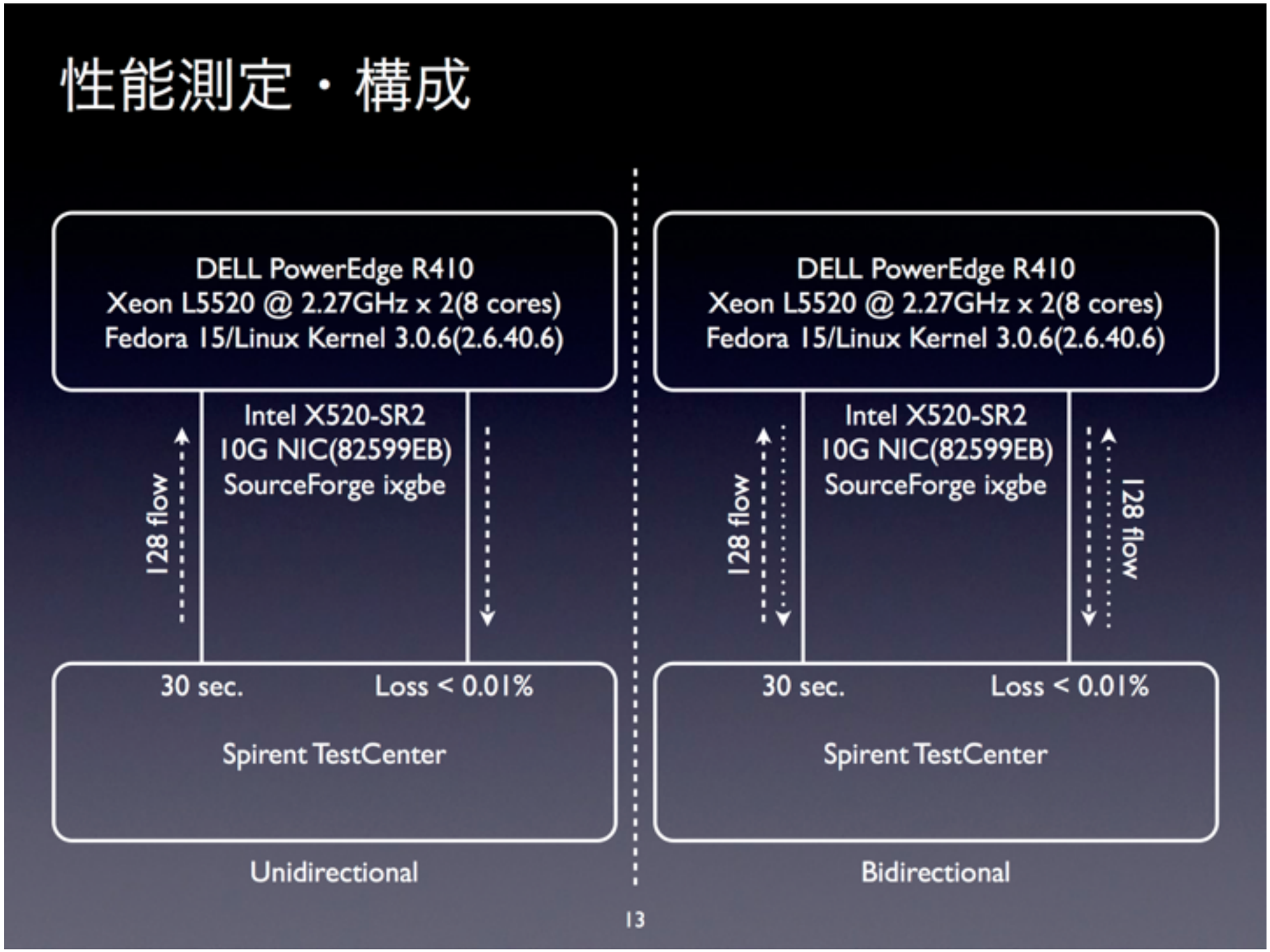
(7) ksoftirqd は softnet_data 構造体の poll_list にリンクされた napi_struct 構造体の受信バッファからパケットを取り出し受信処理を行う



Receive Side Scaling (RSS)

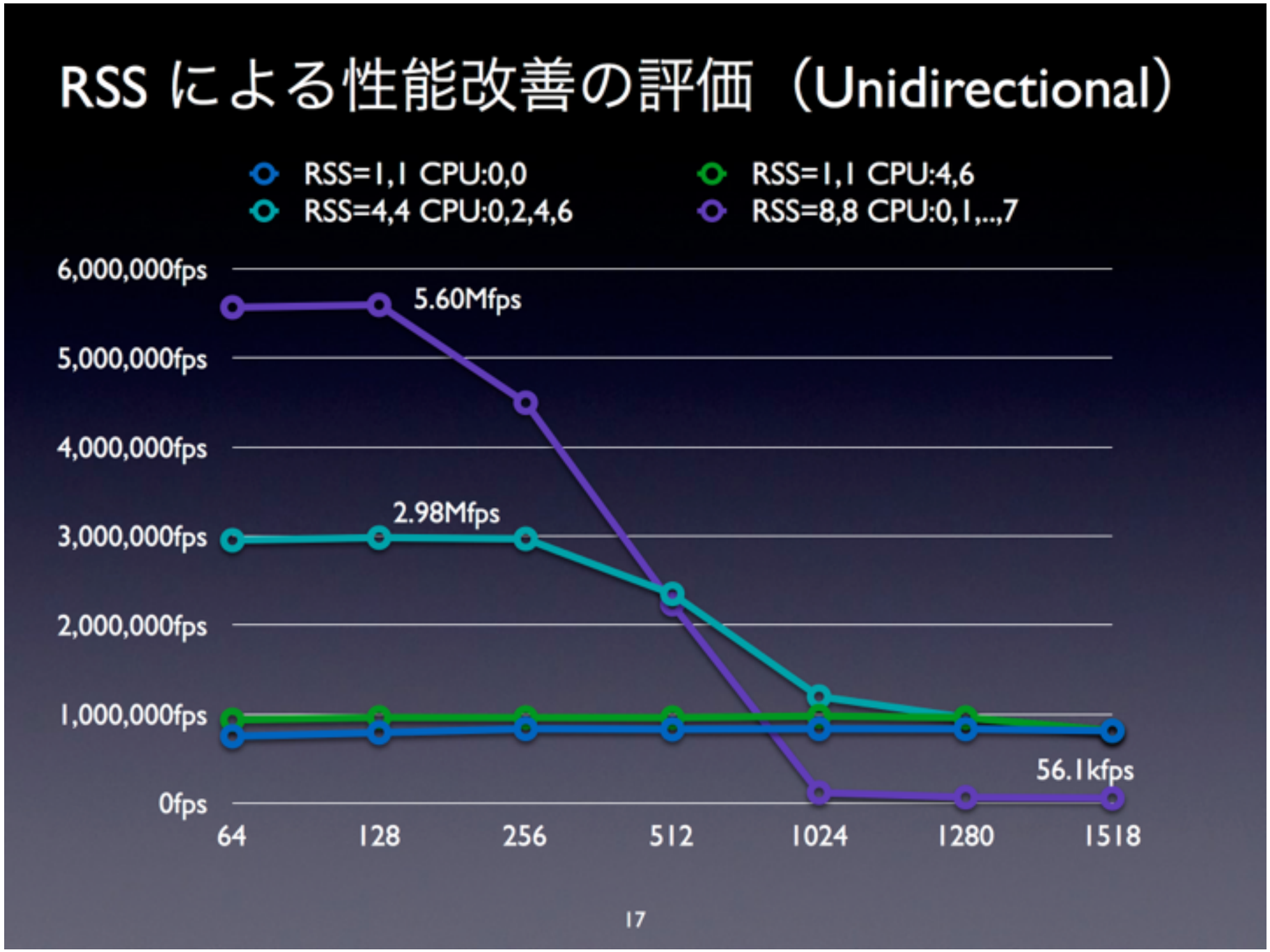


Receive Side Scaling (RSS)



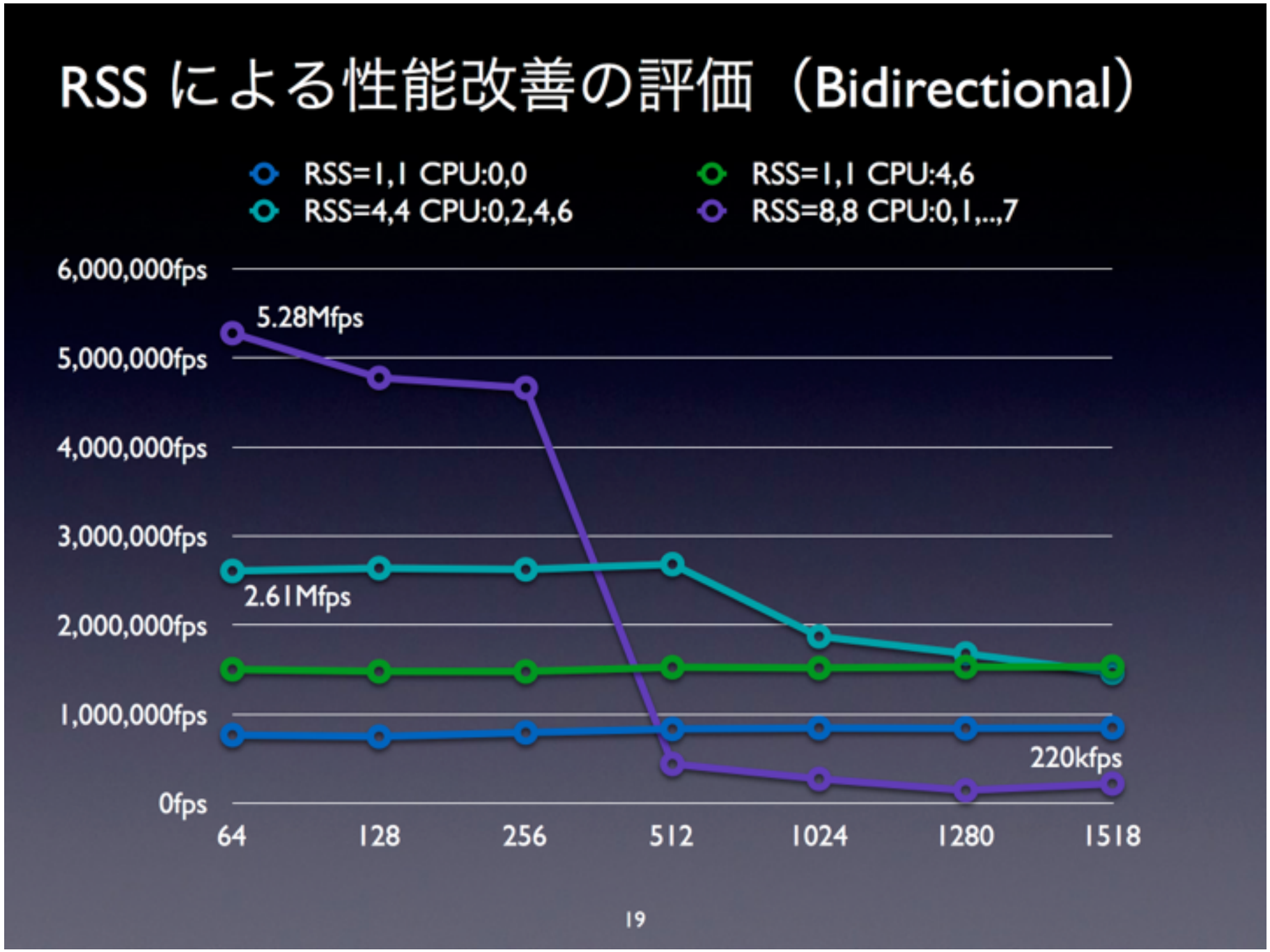
Internet Week 2011 S9 仮想化時代のパケットフォワーディング
Linux システムにおけるパケットフォワーディング概要(浅間 正和/有限会社 銀座堂)の発表資料から引用

Receive Side Scaling (RSS)



Internet Week 2011 S9 仮想化時代のパケットフォワーディング
Linux システムにおけるパケットフォワーディング概要(浅間 正和/有限会社 銀座堂)の発表資料から引用

Receive Side Scaling (RSS)



Internet Week 2011 S9 仮想化時代のパケットフォワーディング
Linux システムにおけるパケットフォワーディング概要(浅間 正和/有限会社 銀座堂)の発表資料から引用

目次

- 高速に経路探索を行うための Linux の実装
  - いろいろな木
  - Linux kernel での IPv4 ルーティングテーブルの実装
- 大量のパケットを処理するための Linux の実装
  - Recieve Livelock と Linux NAPI
  - MultiQueue NIC と Receive Side Scaling(RSS)
- まとめ

まとめ

- 高速に経路探索を行うための Linux の実装
  - Linux kernel のルーティングテーブルの実装を追うことで Linux kernel がどのような高速化の工夫を行っているかを説明した
- 大量のパケットを処理するための Linux の実装
  - 大量のパケット受信により処理性能が激減する問題(Recieve Livelock)を紹介し Linux kernel がとった対策(Linux NAPI)を説明した
  - MultiQueue 機能を持った NIC でパケットを並列処理するための仕組み (Receive Side Scaling(RSS))を説明した

- (ちょっと古いけど…)ちゃんとしたルーターテスターを用いて行ったベンチマークでは 1 core あたり 1 Mpps しか出せておらず 10Gbps Ethernet のワイヤーレートには程遠い感じ
- 汎用 OS の I/O スケジューラにこれ以上の性能を期待するのは難しい？
- なにか良い案はないものか…

参考文献

- 高速に経路探索を行うための Linux の実装
  - https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/fib_trie.txt
    - IPv4 ルーティングテーブルを解説している
  - http://www.nada.kth.se/~snilsson/publications/
    - Linux kernel の IPv4 ルーティングテーブルの実装の元になった論文と Java による実装コードが置かれた場所
- 大量のパケットを処理するための Linux の実装
  - https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/scaling.txt
    - Receive Side Scaling(RSS) 等のネットワーク性能を CPU コア数に応じてスケールさせるための技術を解説している
  - http://www.linuxfoundation.org/collaborate/workgroups/networking/napi
    - Linux NAPI の実装について詳しく解説している

参考) レベル圧縮する閾値による違い

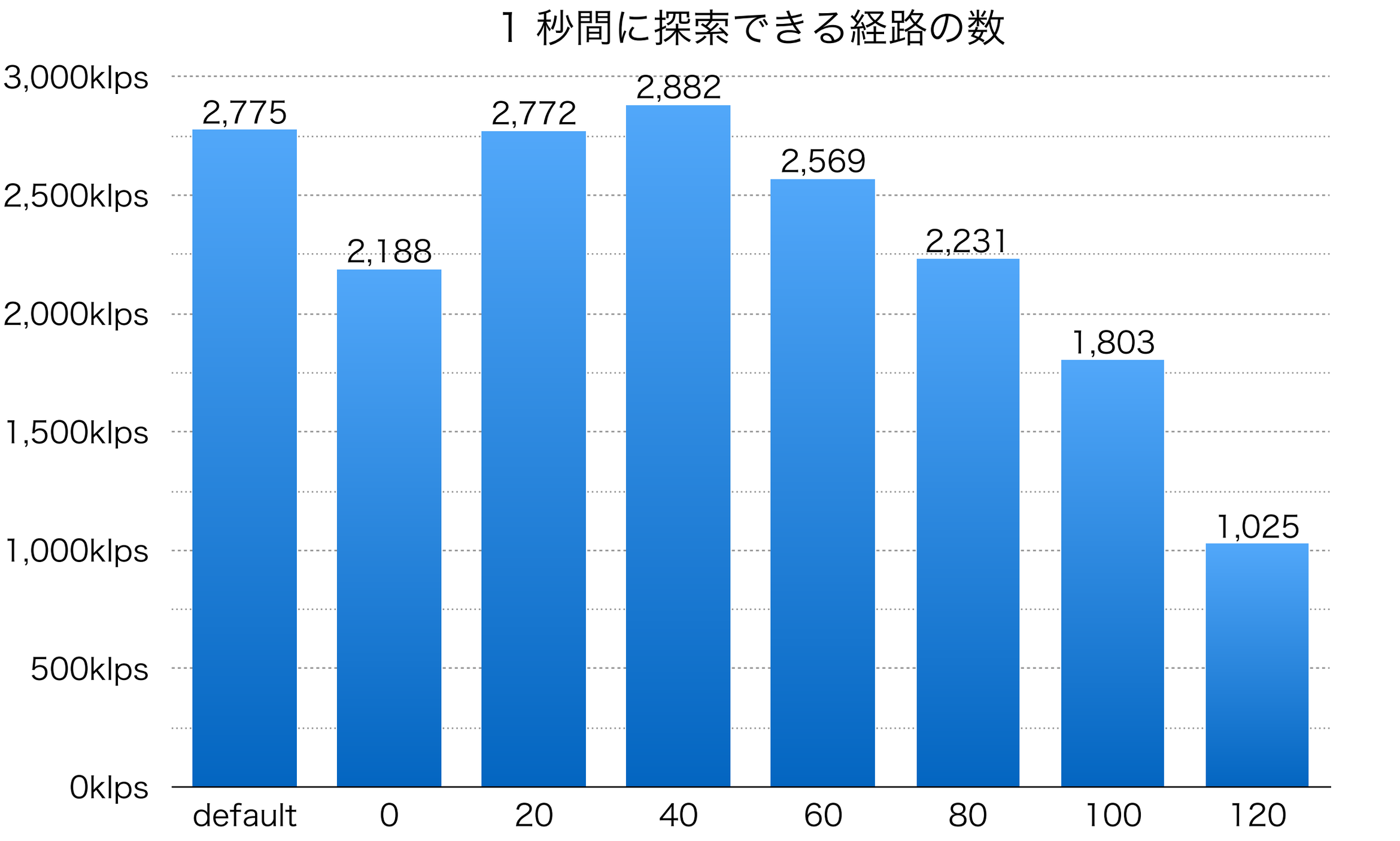
```
ssize_t
read_proc(struct file *filp, char *buf, size_t count, loff_t * offp)
{
    unsigned long before, after, counter;
    struct net *ns;
    struct fib_table *tb;
    struct flowi4 fl;
    struct fib_result res;
...
    srand32(0); // ※
    before = jiffies;
    for (counter = 0; counter < 20000000; ++counter)
    {
redo:
        fl.daddr = rand32(); // ※
        if ((fl.daddr & 0xe0) == 0xe0)
            goto redo;
        // memset(&res, 0, sizeof(res));
        fib_table_lookup(tb, &fl, &res, 0);
        // printk(KERN_INFO "%pI4 -> %p\n", &fl.daddr, res.fi);
    }
    after = jiffies;
    count = snprintf (msg, MSGLEN, "%ld\n", (after - before));
    copy_to_user (buf, msg, count);
...
}
```

※ http://www.bumblebeagle.org/polyominoes/tilingcounting/rand32.c

参考) レベル圧縮する閾値による違い

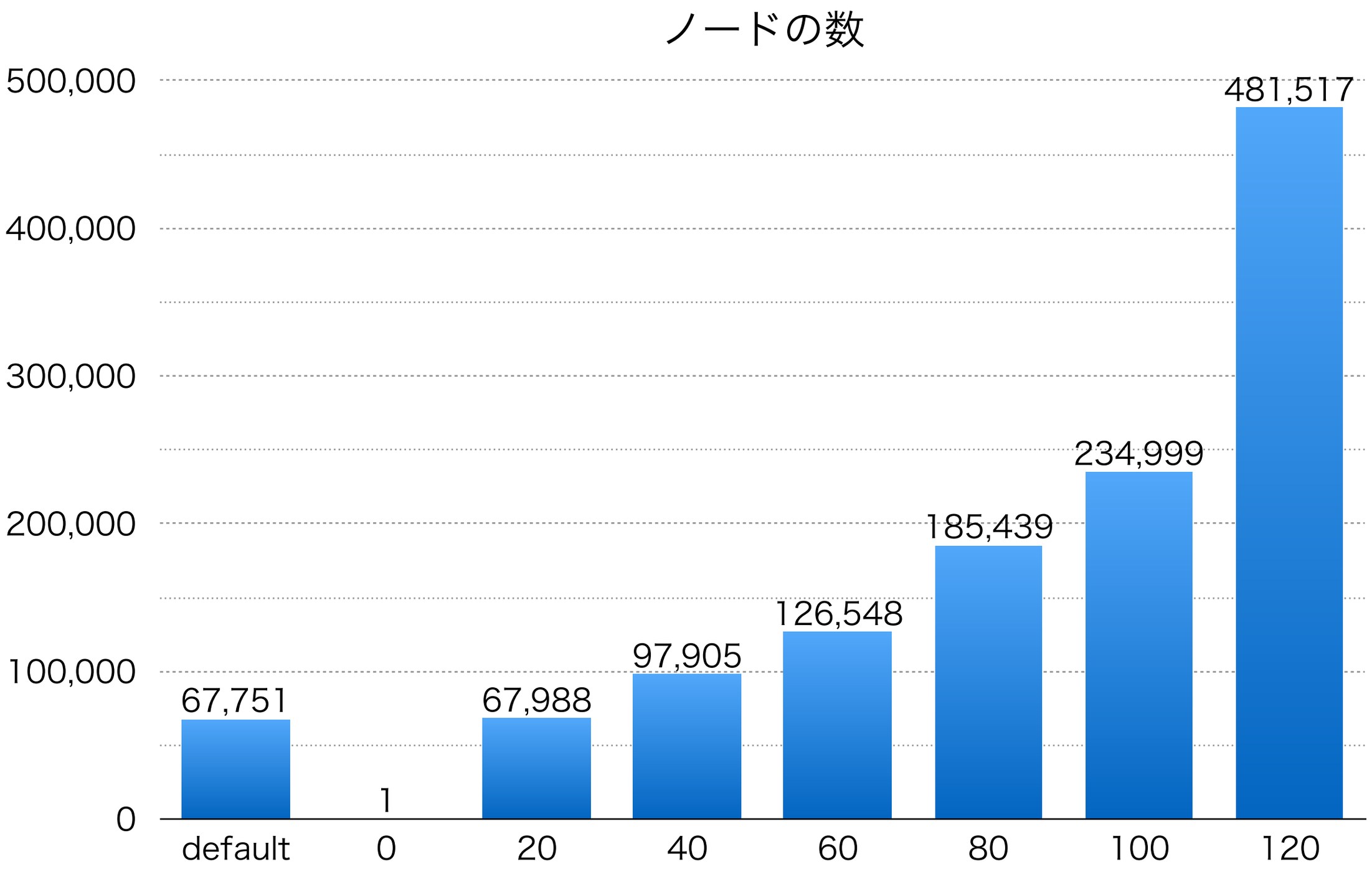
	意味	備考
default	50%以上の子が埋まるならノードを圧縮 (ただし root だけは 30%以上で圧縮)	Linux kernel の設定(固定)
0	0%以上の子が埋まるならノードを圧縮 (1 つのとてつもなく大きなノード)	Relaxed LPC-Trie
20	20%以上の子が埋まるならノードを圧縮	Relaxed LPC-Trie
40	40%以上の子が埋まるならノードを圧縮	Relaxed LPC-Trie
60	60%以上の子が埋まるならノードを圧縮	Relaxed LPC-Trie
80	80%以上の子が埋まるならノードを圧縮	Relaxed LPC-Trie
100	100%以上の子が埋まるならノードを圧縮	Perfect LPC-Trie
120	いかなる場合でもノードを圧縮しない	Patricia Trie

参考) レベル圧縮する閾値による違い

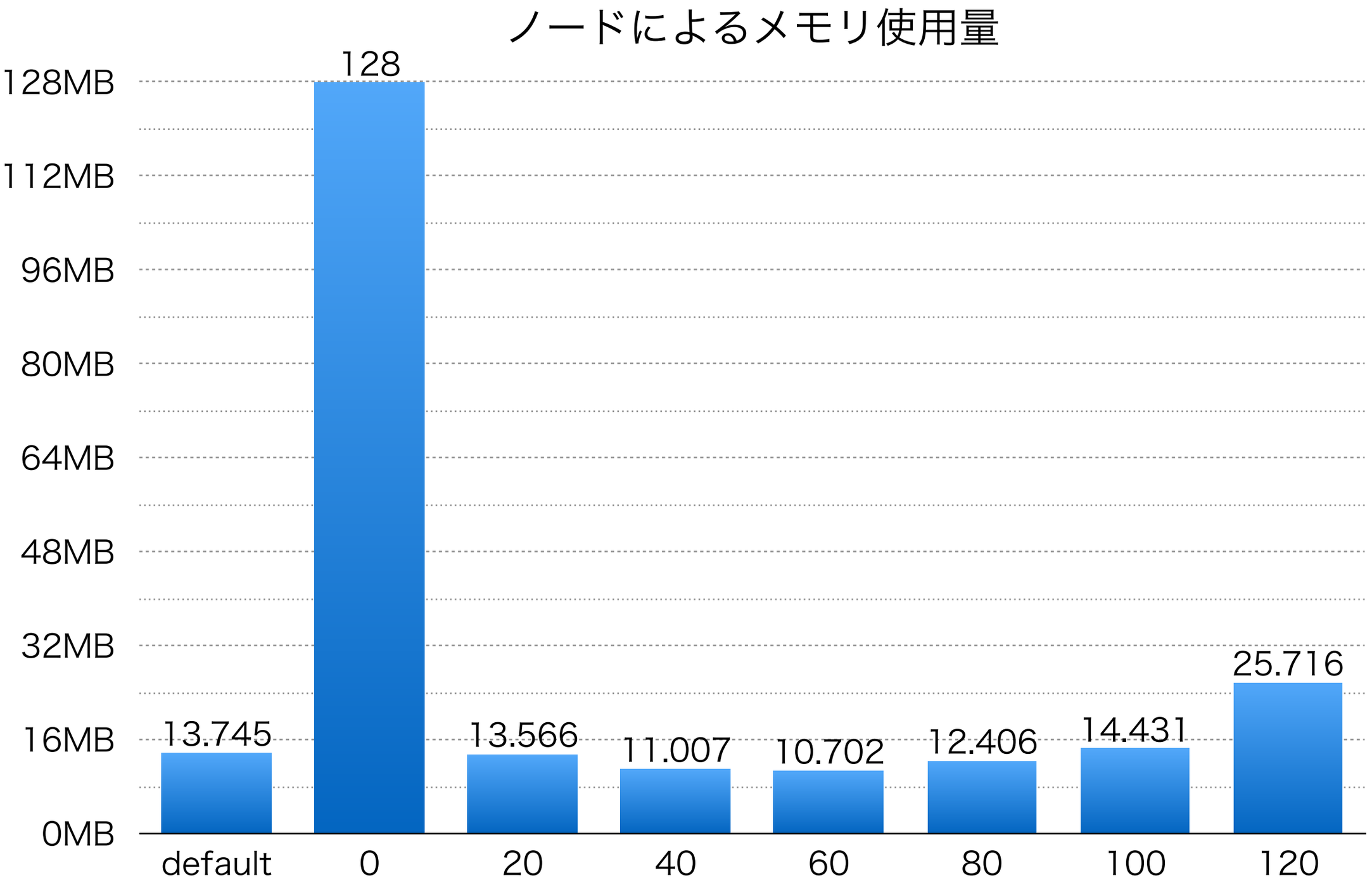


※ デフォルトルートのみの時は 27,174klps

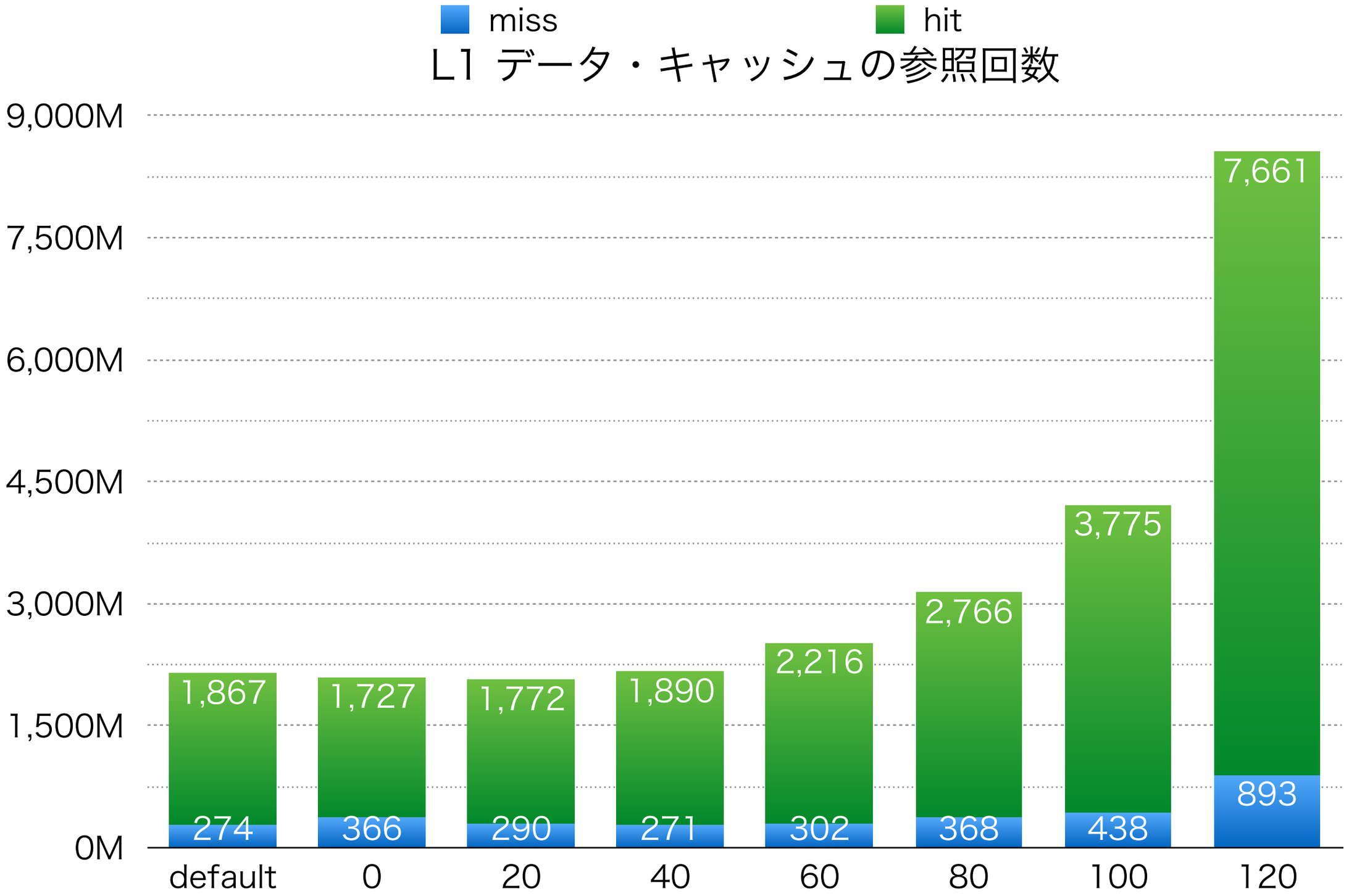
参考) レベル圧縮する閾値による違い



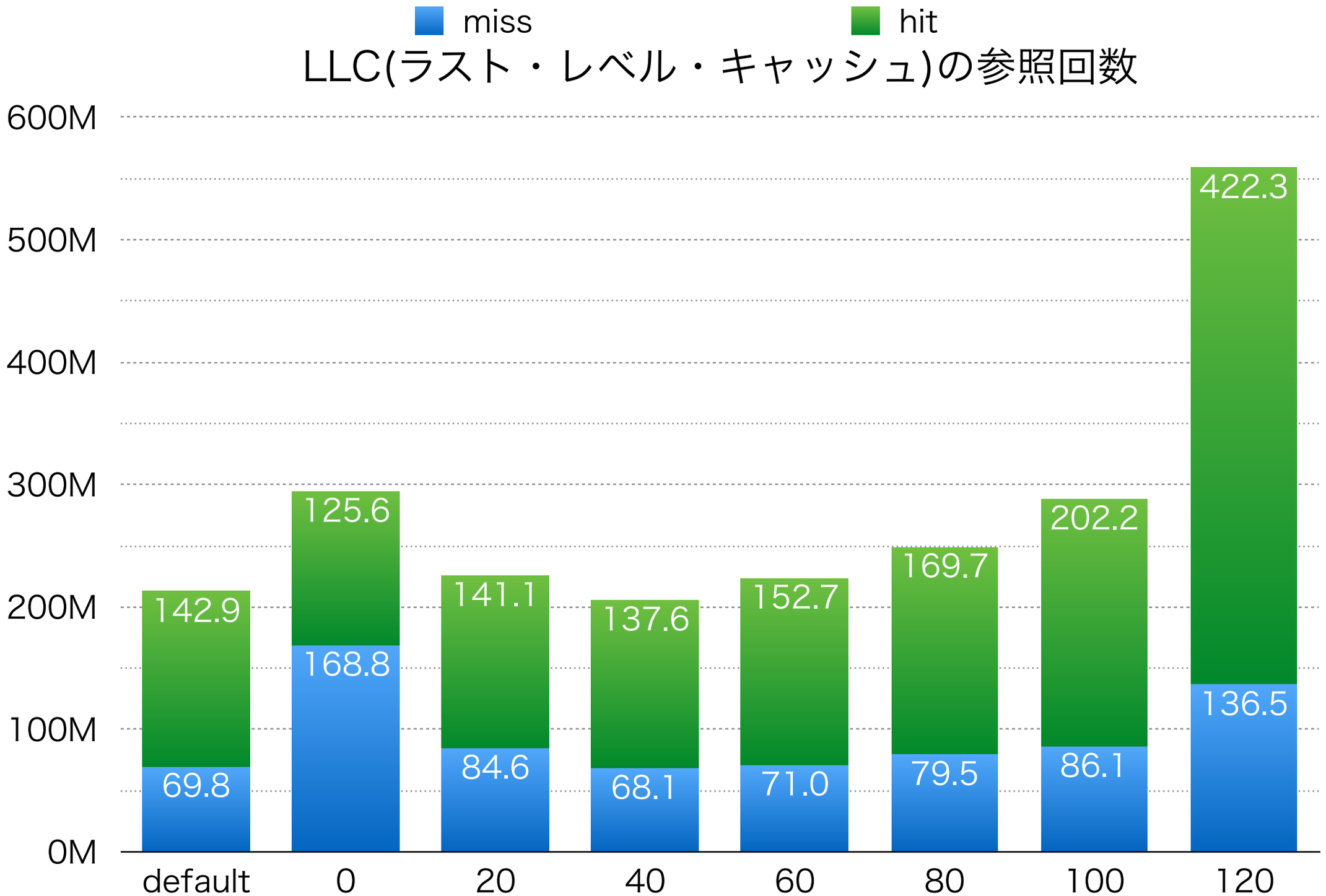
参考) レベル圧縮する閾値による違い



参考) レベル圧縮する閾値による違い



参考) レベル圧縮する閾値による違い



参考) レベル圧縮する閾値による違い

部品	スペック
Blade	HP ProLiant BL460c G6
CPU	Intel(R) Xeon(R) CPU L5520 @ 2.27GHz
L1 cache	128KiB
L2 cache	1MiB
L3 cache	8MiB
Memory	DIMM DDR3 Synchronous 1333 MHz × 3