2026 年 3 月 24 日 – IIJ Lab Seminar

カーネルバイパスによる通信高速化の基本

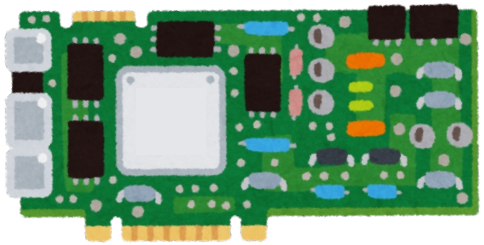
IIJ 技術研究所　安形

カーネルバイパスとは？

カーネルバイパスとは？

カーネルバイパスとは？

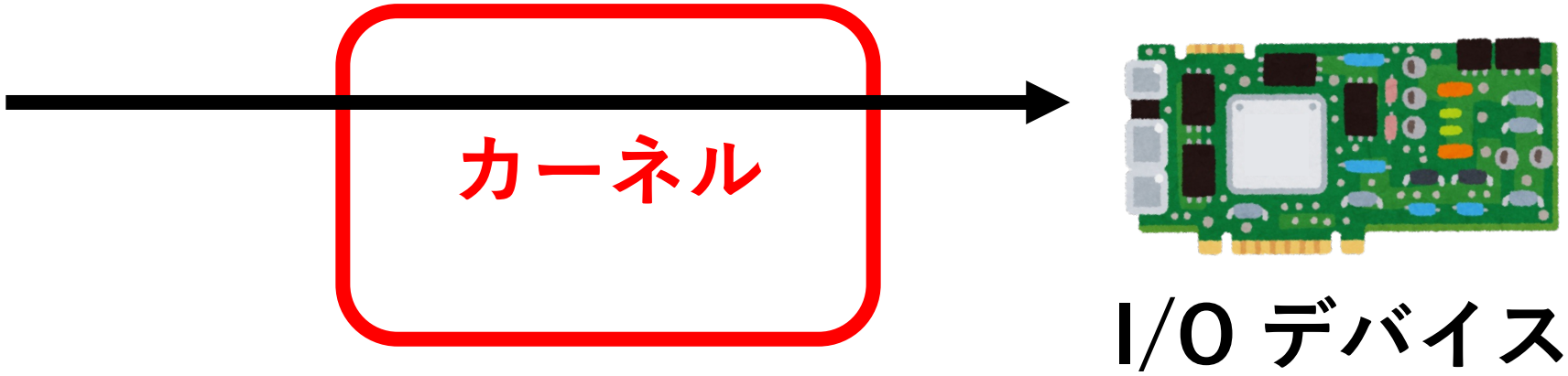
カーネル



I/O デバイス

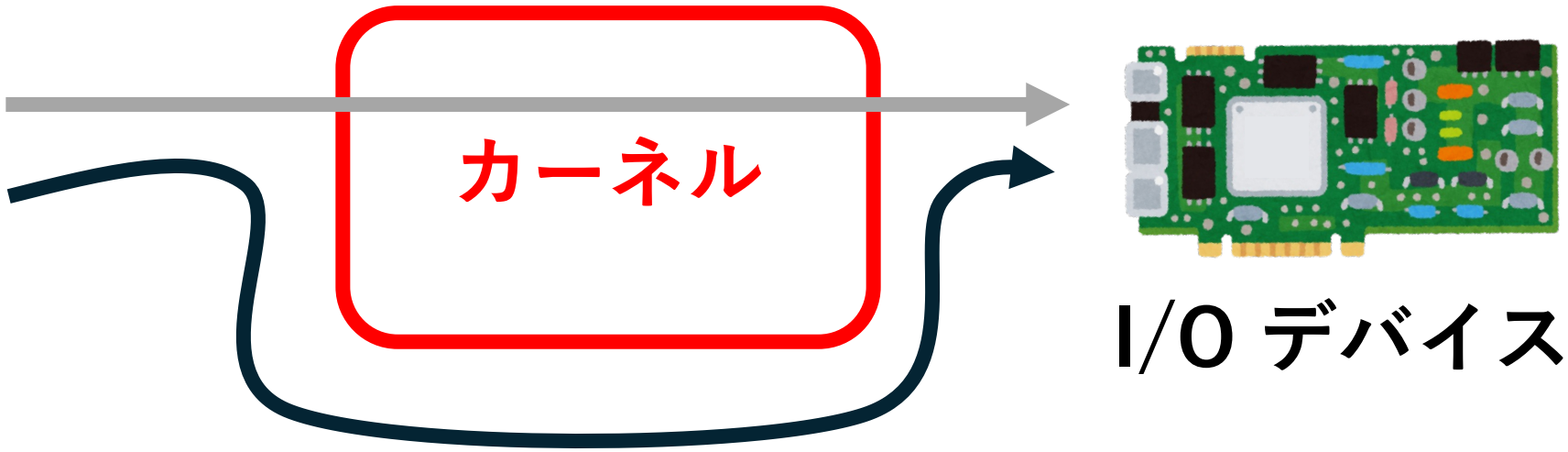
カーネルバイパスとは？

- 基本的にカーネルを経由していたところ

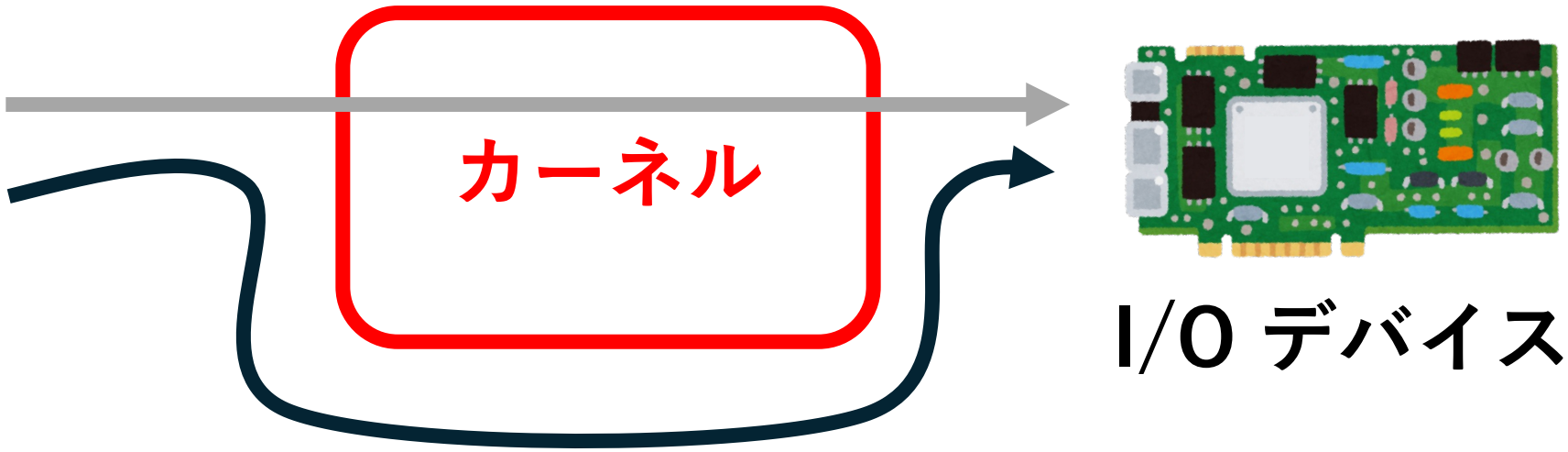


カーネルバイパスとは？

- 基本的にカーネルを経由していたところ、迂回するようにする

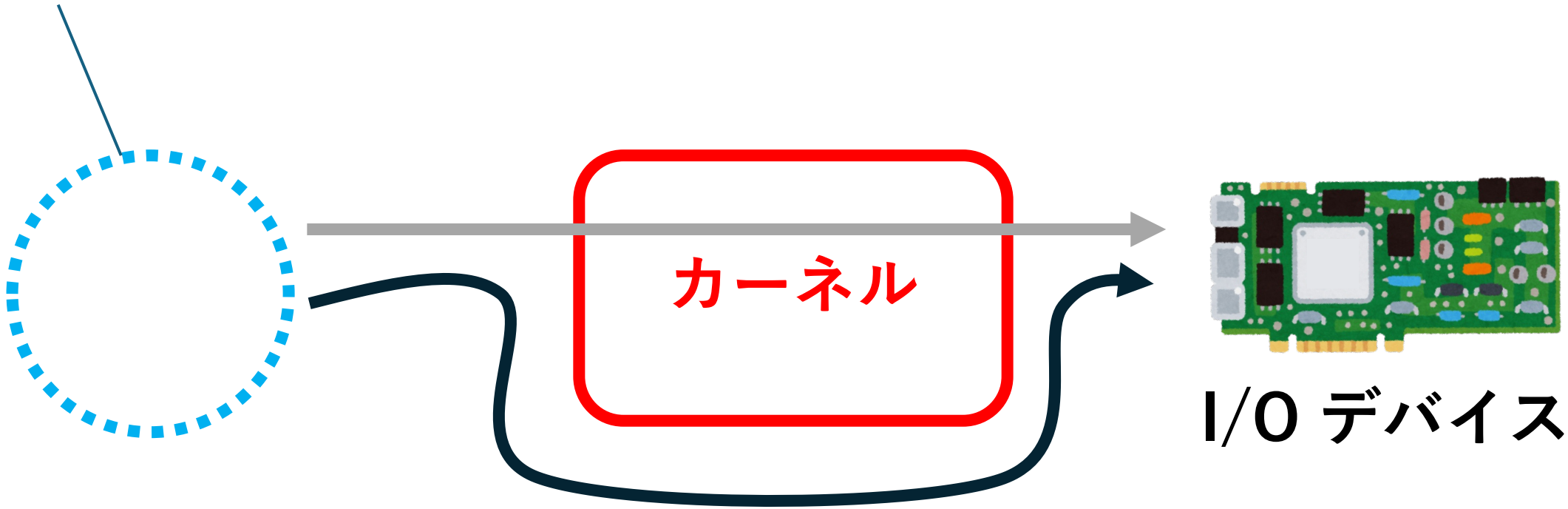


カーネルバイパスとは？



カーネルバイパスとは？

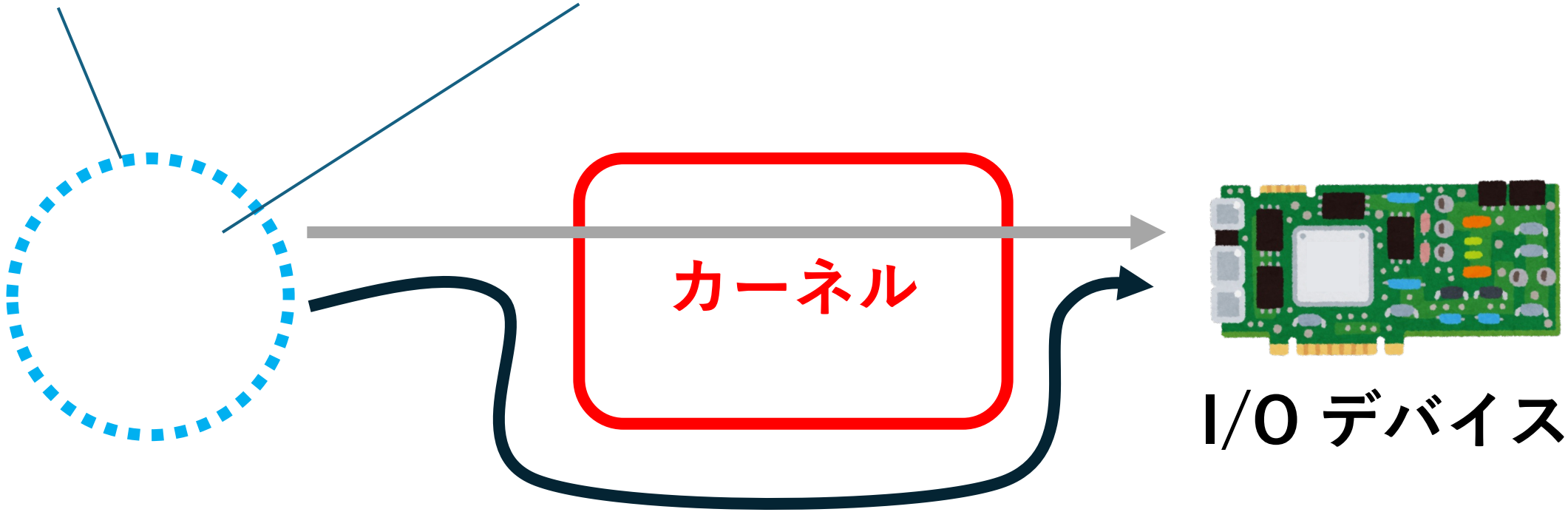
Q. これは何？



カーネルバイパスとは？

Q. これは何？

Q. 何がカーネルを中継しなくなる？



カーネルバイパスとは？

Q. これは何？

Q. 何がカーネルを中継しなくなる？

Q. デバイスへの
アクセスとは
具体的に何？

カーネルバイパスとは？

Q. これは何？

Q. 何がカーネルを中継しなくなる？

Q. デバイスへの
アクセスとは
具体的に何？

Q. 何故通常はカーネルを経由する？

カーネルバイパスとは？

Q. これは何？

Q. 何がカーネルを中継しなくなる？

Q. デバイスへの
アクセスとは
具体的に何？

I/O デバイス

Q. 何故通常はカーネルを経由する？

Q. どうやってバイパスする？

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

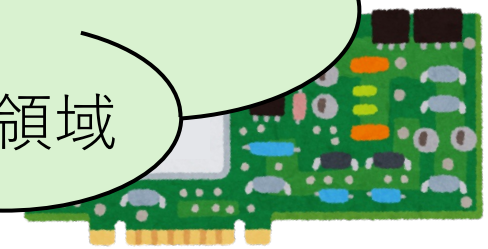
カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセ[illegible]
I/O

Q. デバイスへの
アクセスとは
具体的に何？
[illegible]的には
[illegible]の読み書き

Q. ユーザー空間とは何？
A. CPU が非特権モードで
動作している間に
アクセス可能なメモリ領域



I/O デバイス

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

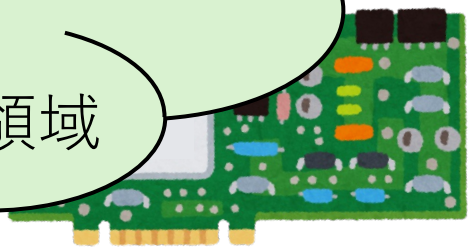
カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセ
I/O

Q. デバイスへの
アクセスとは
具体的に何？
基本的には
Oの読み書き

Q. ユーザー空間とは何？
A. CPU が非特権モードで
動作している間に
アクセス可能なメモリ領域



I/O デバイス

CPU の非特権モードとは？

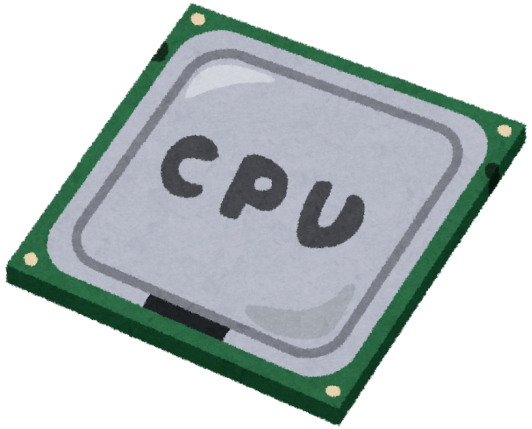
Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

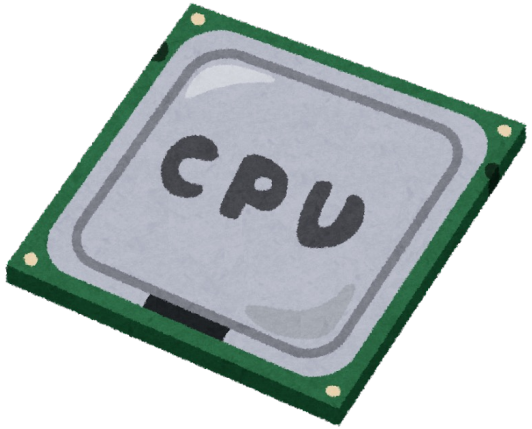


CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**



CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

  この方針が適用されて**いない**場合

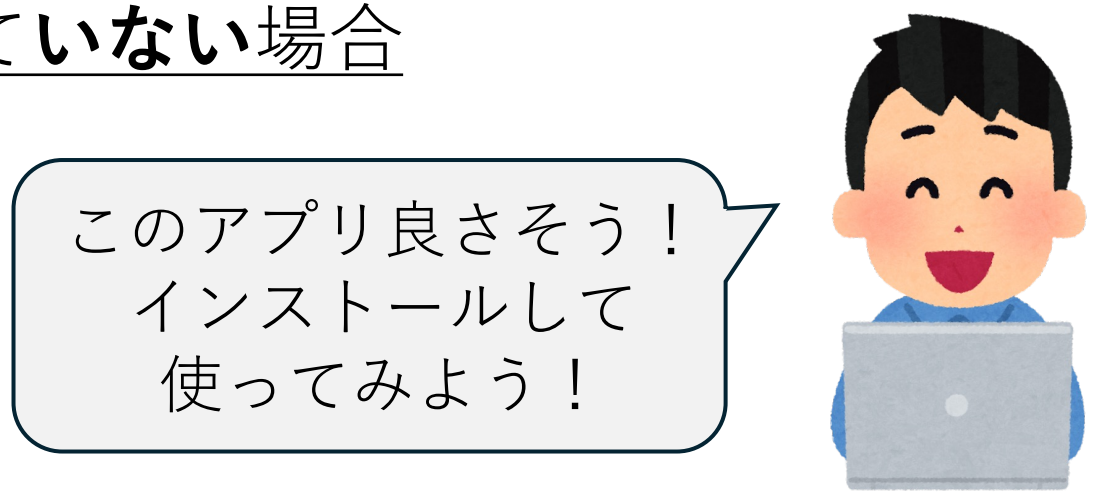
CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

この方針が適用されて**いない**場合



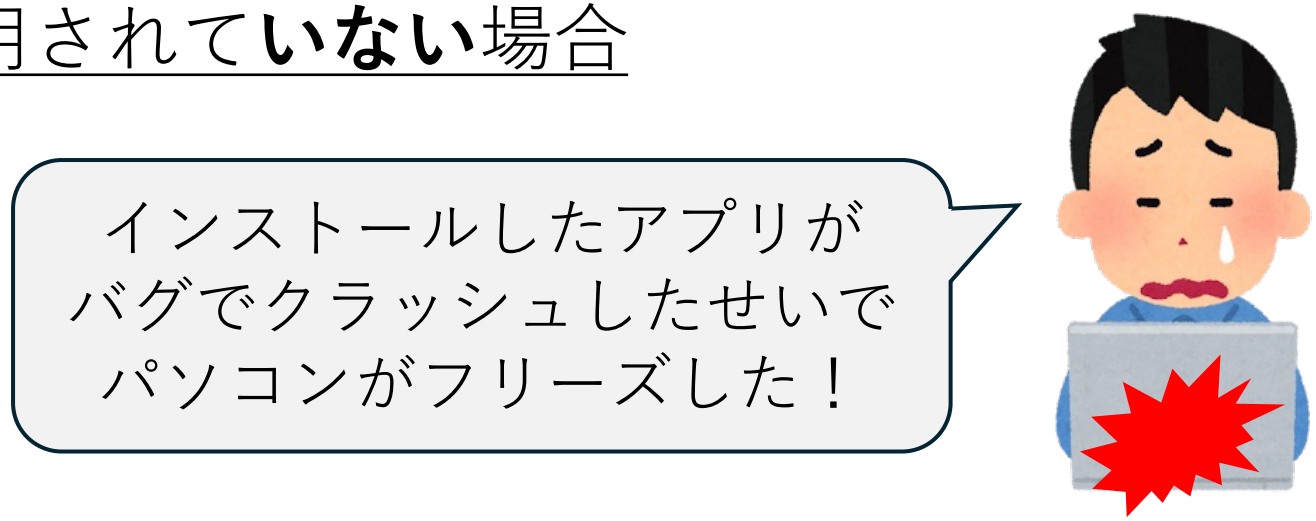
CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

この方針が適用されて**いない**場合



CPU のモードと基本的な運用方針

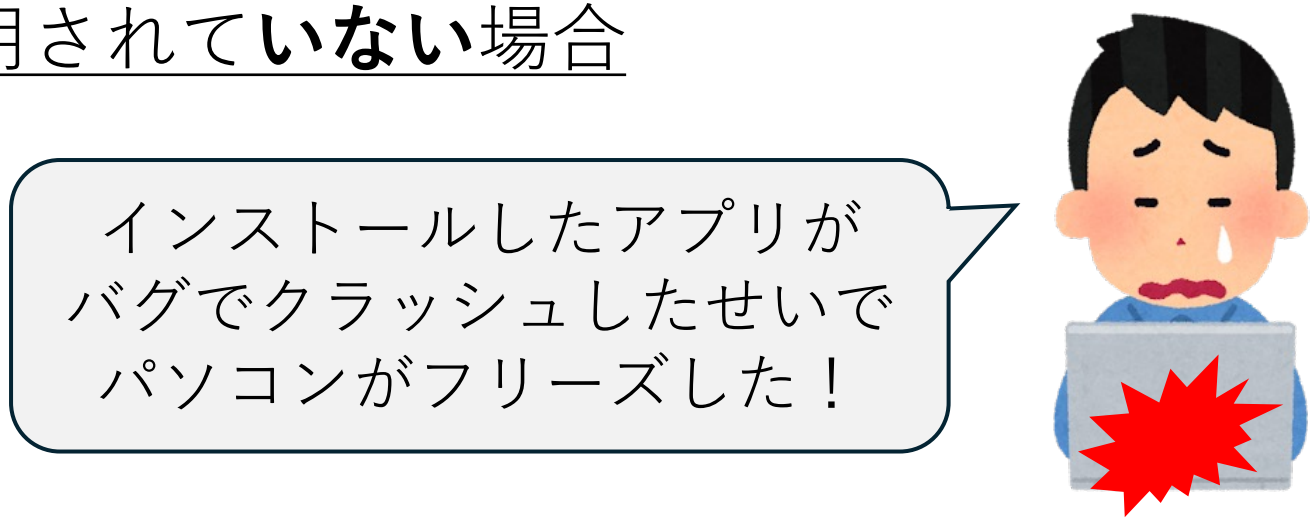
- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

この方針が適用されて**いない**場合

アプリのクラッシュがパソコン全体に影響を与えている



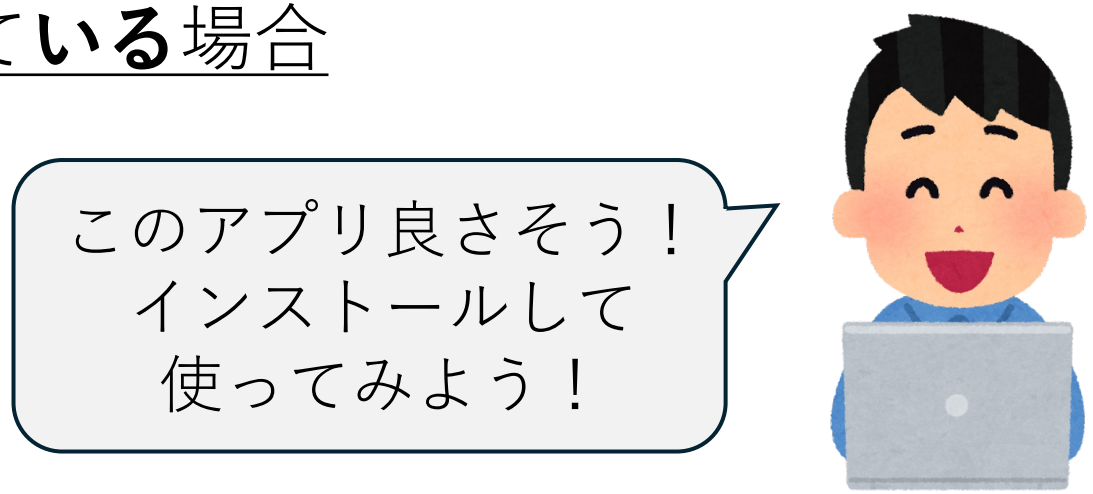
CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

この方針が適用されて**いる**場合



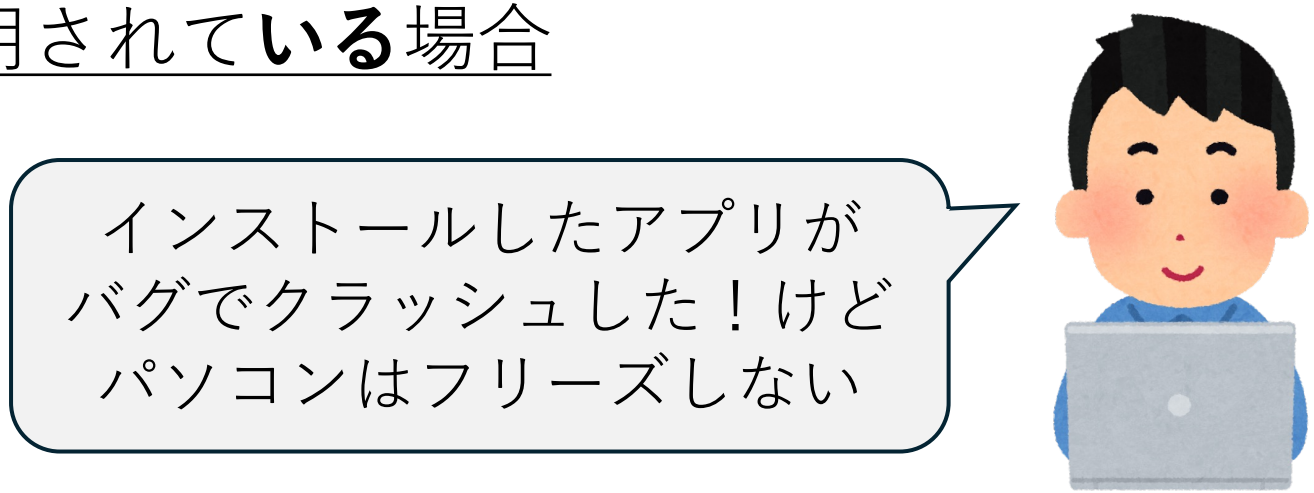
CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

この方針が適用されて**いる**場合



CPU のモードと基本的な運用方針

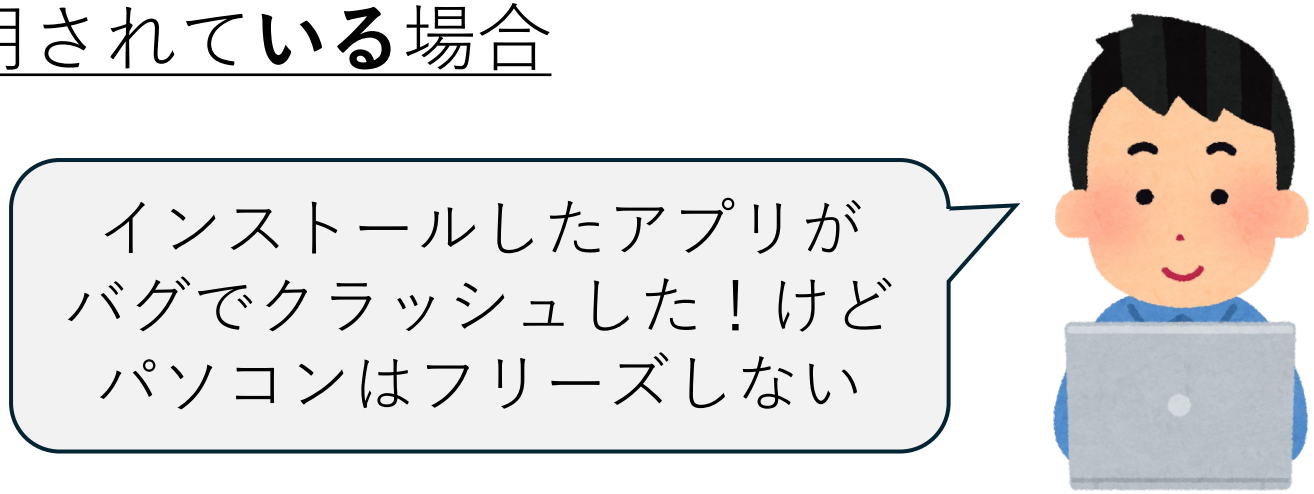
- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

この方針が適用されて**いる**場合

アプリのクラッシュの影響はパソコン全体には波及しない



CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**
  - ほとんどのプログラムを**非特権モード**で実行（破壊操作が難しい）
  - 限られたプログラムのみを**特権モード**で実行

CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**
  - ほとんどのプログラムを**非特権モード**で実行（破壊操作が難しい）
  - 限られたプログラムのみを**特権モード**で実行

- 雑な「カーネル」の定義：**特権モード**で実行されるプログラム

OS の設計によっては必ずしもこの限りではないと思われます

CPU のモードと基本的な運用方針

- CPU のモードは大まかに分けて二
  1. **特権モード**：大体の CPU 機能を使
  2. **非特権モード**：できることが制限

特権モードはカーネルモード、非特権モードはユー

非特権モードで動作するアプリが
クラッシュしても
パソコンはフリーズしないが
特権モードで動作するカーネルが
完全にクラッシュすると
パソコンはフリーズする

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**
  - ほとんどのプログラムを**非特権モード**で実行（破壊操作が難しい）
  - 限られたプログラムのみを**特権モード**で実行

- 雑な「カーネル」の定義：**特権モード**で実行されるプログラム

OS の設計によっては必ずしもこの限りではないと思われます

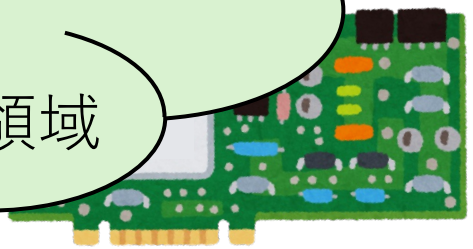
カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセ
I/O

Q. デバイスへの
アクセスとは
具体的に何？
基本的には
Oの読み書き

Q. ユーザー空間とは何？
A. CPU が**非特権モード**で
動作している間に
アクセス可能なメモリ領域



CPU の非特権モードとは？

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

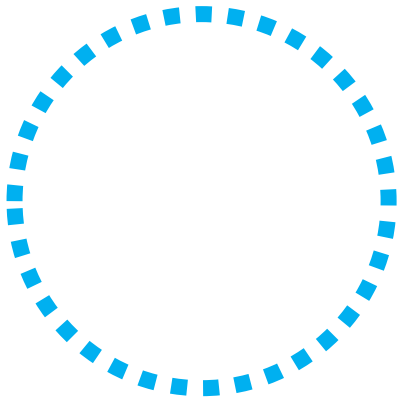
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

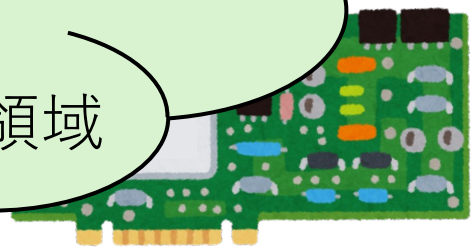
Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセ
I/O

Q. デバイスへの
アクセスとは
具体的に何？
本的には
Iの読み書き



Q. ユーザー空間とは何？
A. CPU が非特権モードで
動作している間に
アクセス可能なメモリ領域



そもそもメモリアクセスとは？

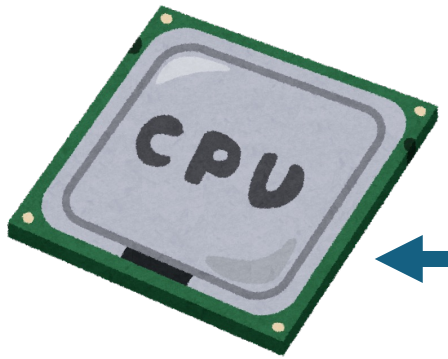
Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

プログラムによるメモリアクセス

プログラムによるメモリアクセス

メモリ書き込みをするには
どんなプログラムを書けば良いか？



プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

プログラムによるメモリアクセス

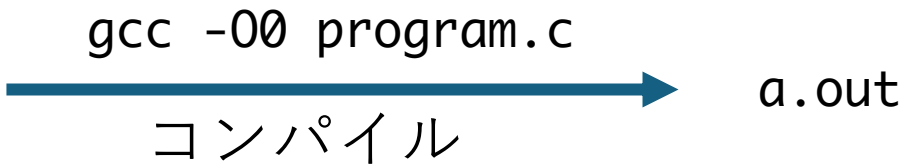
例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

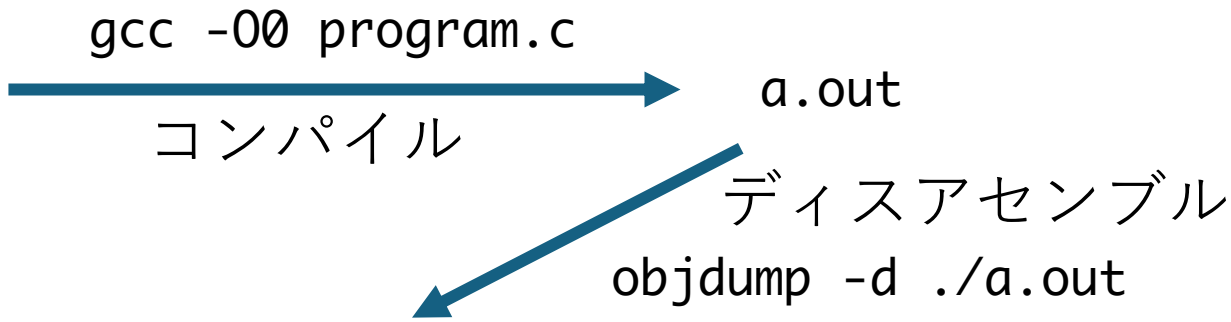
```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```



プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```



プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```

gcc -O0 program.c

コンパイル

a.out

ディスアセンブル

objdump -d ./a.out

```
0000000000001129 <main>:
    1129:       f3 0f 1e fa             endbr64
    112d:       55                      push   %rbp
    112e:       48 89 e5                mov    %rsp,%rbp
    1131:       48 b8 00 00 00 00 01    movabs $0x100000000,%rax
    1138:       00 00 00
    113b:       c7 00 78 56 34 12       movl   $0x12345678,(%rax)
    1141:       b8 00 00 00 00          mov    $0x0,%eax
    1146:       5d                      pop    %rbp
    1147:       c3                      retq
    1148:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
    114f:       00
```

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```

gcc -O0 program.c

コンパイル

a.out

ディスアセンブル

objdump -d ./a.out

```
0000000000001129 <main>:
    1129:  f3 0f 1e fa                  endbr64
    112d:  55                           push   %rbp
    112e:  48 89 e5                     mov    %rsp,%rbp
    1131:  48 b8 00 00 00 00 01         movabs $0x100000000,%rax
    1138:  00 00 00
    113b:  c7 00 78 56 34 12            movl   $0x12345678,(%rax)
    1141:  b8 00 00 00 00               mov    $0x0,%eax
    1146:  5d                           pop    %rbp
    1147:  c3                           retq
    1148:  0f 1f 84 00 00 00 00 00      nopl   0x0(%rax,%rax,1)
    114f:
```

機械語

アセンブリ言語

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```

gcc -O0 program.c

コンパイル

a.out

ディスアセンブル

objdump -d ./a.out

```
0000000000001129 <main>:
    1129:       f3 0f 1e fa             endbr64
    112d:       55                      push   %rbp
    112e:       48 89 e5                mov    %rsp,%rbp
    1131:       48 b8 00 00 00 00 01    movabs $0x100000000,%rax
    1138:       00 00 00
    113b:       c7 00 78 56 34 12       movl   $0x12345678,(%rax)
    1141:       b8 00 00 00 00          mov    $0x0,%eax
    1146:       5d                      pop    %rbp
    1147:       c3                      retq
    1148:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
    114f:       00
```

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```

gcc -O0 program.c

コンパイル

a.out

ディスアセンブル

objdump -d ./a.out

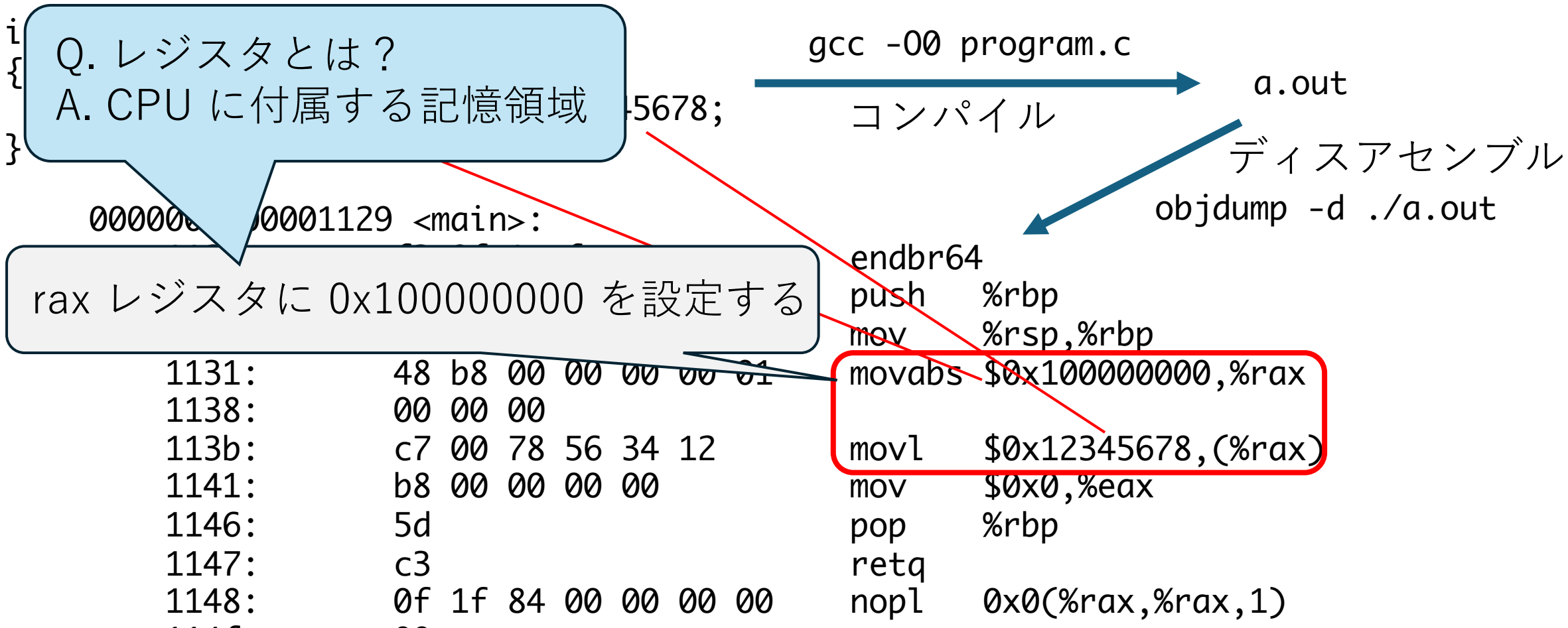
```
0000000000001129 <main>:
```

rax レジスタに 0x100000000 を設定する

```
1131:       48 b8 00 00 00 00 01      endbr64
1138:       00 00 00
113b:       c7 00 78 56 34 12         push   %rbp
1141:       b8 00 00 00 00            mov    %rsp,%rbp
1146:       5d                        movabs $0x100000000,%rax
1147:       c3
1148:       0f 1f 84 00 00 00 00      movl   $0x12345678,(%rax)
114f:       00                        mov    $0x0,%eax
                                      pop    %rbp
                                      retq
                                      nopl   0x0(%rax,%rax,1)
```

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

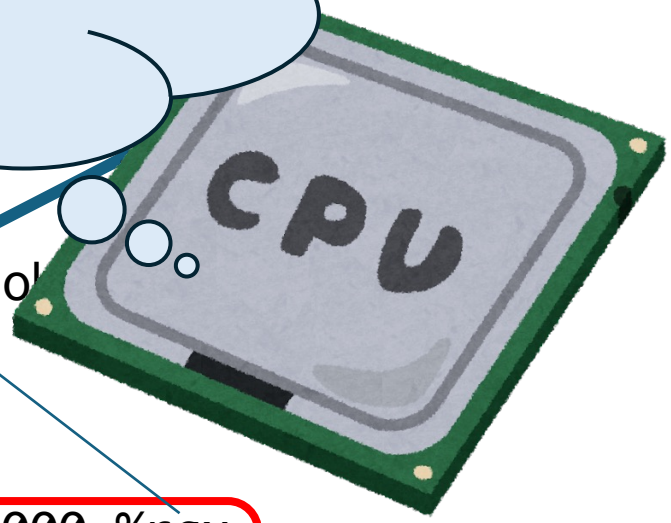


プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 ～ グラム

Q. レジスタとは？
A. CPU に付属する記憶領域

rax, rbx, rcx, rdx, …
rbp, rsp, …



レジスタそれぞれに名前がついており
操作のための専用の CPU 命令が
CPU に実装されている

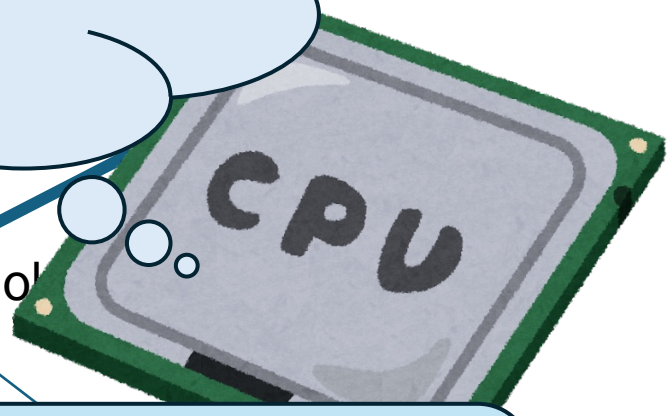
```
0000000000001129 <main>:
    1129:       f3 0f 1e fa             endbr64
    112d:       55                      push   %rbp
    112e:       48 89 e5                mov    %rsp,%rbp
    1131:       48 b8 00 00 00 00 01    movabs $0x100000000,%rax
    1138:       00 00 00
    113b:       c7 00 78 56 34 12       movl   $0x12345678,(%rax)
    1141:       b8 00 00 00 00          mov    $0x0,%eax
    1146:       5d                      pop    %rbp
    1147:       c3                      retq
    1148:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
    114f:       00
```

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 [illegible] グラム

Q. レジスタとは？
A. CPU に付属する記憶領域

rax, rbx, rcx, rdx, …
rbp, rsp, …



レジスタそれぞれに名前がついており
操作のための専用の CPU 命令が
CPU に実装されている

ポイント

- CPU 付属の高速な記憶領域
- CPU 命令から直接操作可能

```
1146:       5d                      pop    %rbp
1147:       c3                      retq
1148:       0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
114f:       00
```

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```

gcc -O0 program.c

コンパイル

a.out

ディスアセンブル

objdump -d ./a.out

```
0000000000001129 <main>:
    1131:       48 b8 00 00 00 00 01
    1138:       00 00 00
    113b:       c7 00 78 56 34 12
    1141:       00
    endbr64
    push   %rbp
    mov    %rsp,%rbp
    movabs $0x100000000,%rax

    movl   $0x12345678,(%rax)
    mov    $0x0,%eax
    pop    %rbp
    retq
    nopl   0x0(%rax,%rax,1)
```

rax レジスタに 0x100000000 を設定する

rax レジスタの値により参照される
メモリアドレス（ 0x100000000 ）へ
0x12345678 を書き込む

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```

gcc -O0 program.c

コンパイル

a.out

ディスアセンブル

objdump -d ./a.out

```
0000000000001129 <main>:
    1131:       48 b8 00 00 00 00 01
    1138:       00 00 00
    113b:       c7 00 78 56 34 12
    1141:       00
```

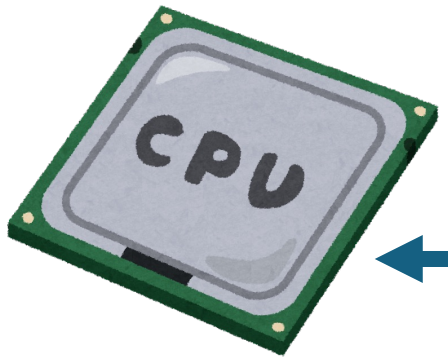
```
endbr64
push   %rbp
mov    %rsp,%rbp
movabs $0x100000000,%rax

movl   $0x12345678,(%rax)
mov    $0x0,%eax
pop    %rbp
retq
nopl   0x0(%rax,%rax,1)
```

rax レジスタに 0x100000000 を設定する

rax レジスタの値により参照される
メモリアドレス（0x100000000）へ
0x12345678 を書き込む

プログラムによるメモリアクセス

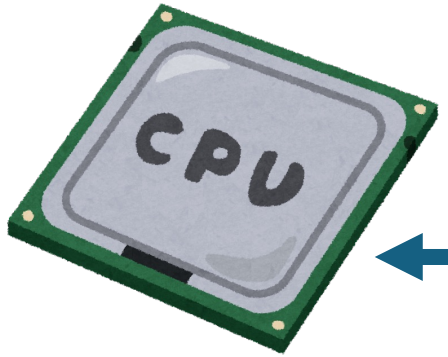
メモリ書き込みをするには
どんなプログラムを書けば良いか？



プログラムによるメモリアクセス

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



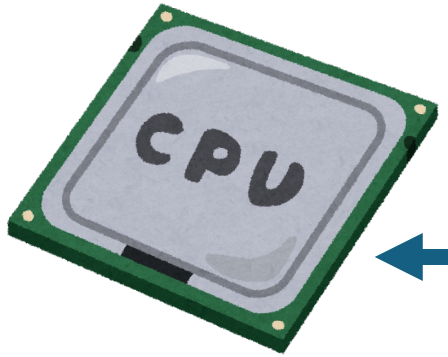
この命令の実行時に CPU から
メモリアクセスが試みられる

```
rax:0x100000000
```

プログラムによるメモリアクセス

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```




```
rax:0x100000000
```

この命令の実行時に CPU から
メモリアクセスが試みられる

ポイント

CPU は以下のような **CPU 命令**を実装している

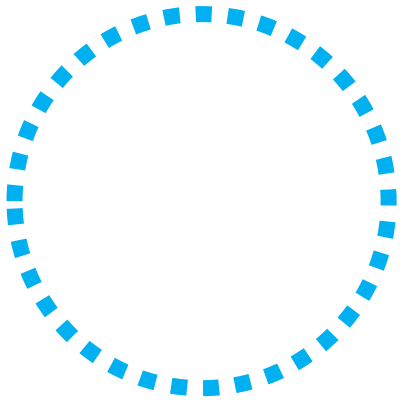
- CPU レジスタの値を読み書きする
- メモリのアドレスを指定して読み書きを行う

カーネルバイパスとは？

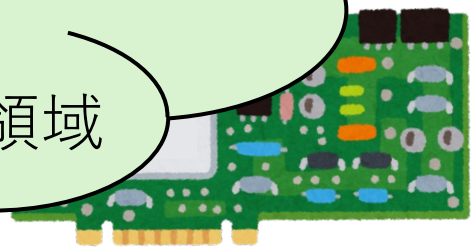
Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセ
I/O

Q. デバイスへの
アクセスとは
具体的に何？
本的には
Iの読み書き



Q. ユーザー空間とは何？
A. CPU が非特権モードで
動作している間に
アクセス可能なメモリ領域



そもそもメモリアクセスとは？

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

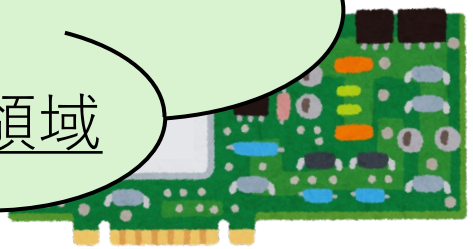
カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスなど

Q. デバイスへの
アクセスとは
具体的に何？
基本的には
I/O の読み書き

Q. ユーザー空間とは何？
A. CPU が非特権モードで
動作している間に
アクセス**可能**なメモリ領域



デバイス

アクセスの可否？

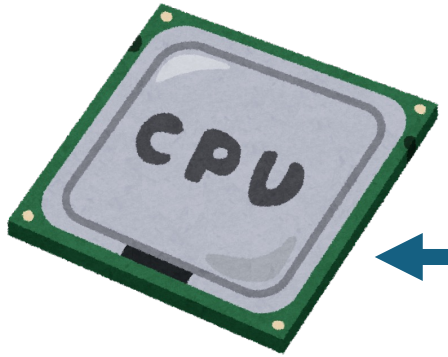
Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

プログラムによるメモリアクセス

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



この命令の実行時に CPU から
メモリアクセスが試みられる

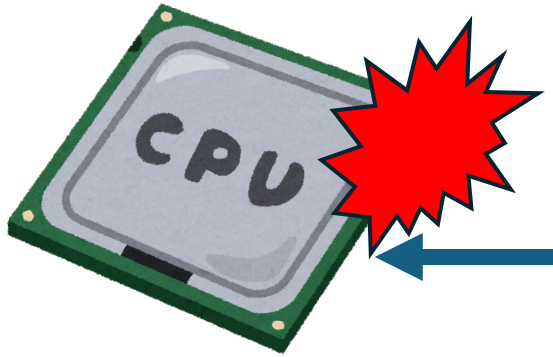
```
rax:0x100000000
```

プログラムによるメモリアクセス

実行するとおそらく
Segmentation fault という
エラーで停止して
メモリアクセスができない

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



この命令の実行時に CPU から
メモリアクセスが試みられる

```
rax:0x100000000
```

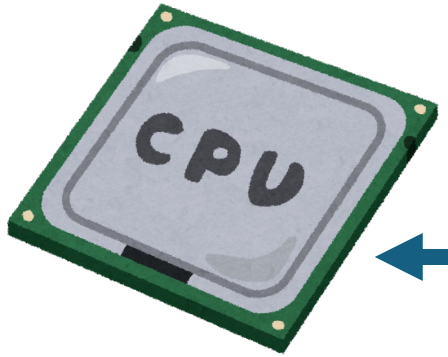
プログラムによるメモリアクセス

実行するとおそらく
Segmentation fault という
エラーで停止して
メモリアクセスができない

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```

何故？：メモリアクセスが制限されているから



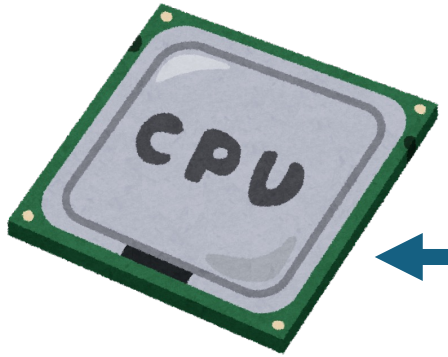

```
rax:0x100000000
```

プログラムによるメモリアクセス

実行するとおそらく
Segmentation fault という
エラーで停止して
メモリアクセスができない

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```




```
rax:0x100000000
```

何故？：メモリアクセスが制限されているから

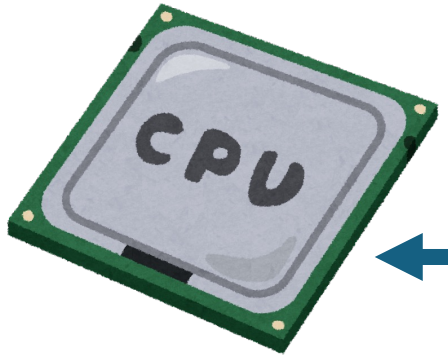
何によって？：カーネル
どうやって？：**MMU** の機能を利用

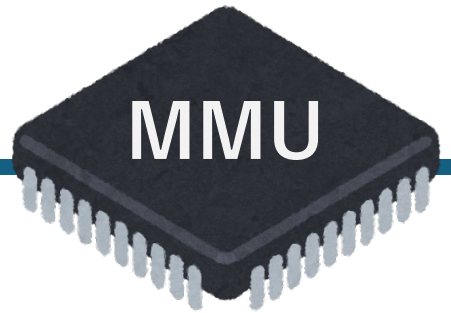
プログラムによるメモリアクセス

CPU からメモリへのアクセスには
MMU (Memory Management Unit) という部品が介在する

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





```
rax:0x100000000
```

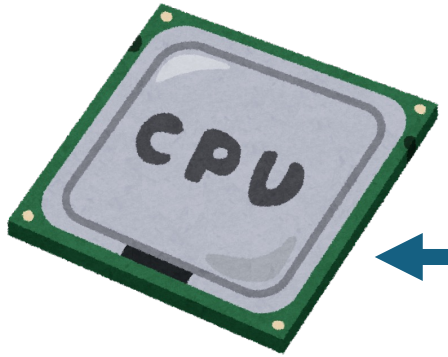
プログラムによるメモリアクセス

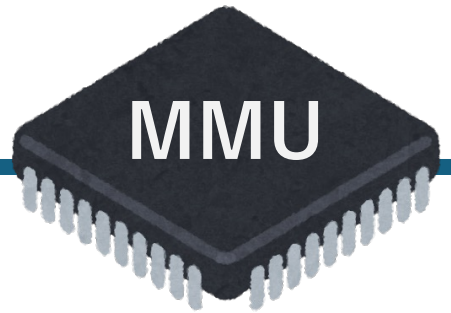
CPU からメモリへのアクセスには
MMU (Memory Management Unit) という部品が介在する

MMU は 仮想アドレスから物理アドレスへの変換を担当

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





```
rax:0x100000000
```

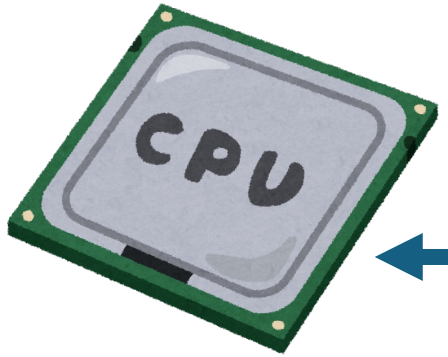
プログラムによるメモリアクセス

CPU からメモリへのアクセスには
MMU (Memory Management Unit) という部品が介在する

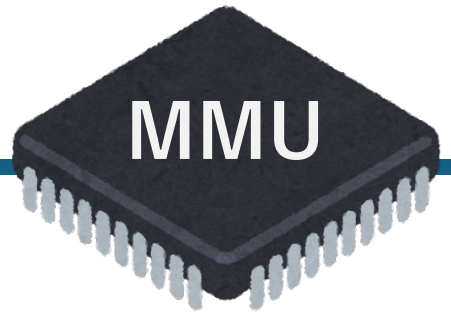
MMU は 仮想アドレスから物理アドレスへの変換を担当

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```




```
rax:0x100000000
```



ポイント
一般的なプログラムが参照する
メモリアドレスは仮想メモリアドレス

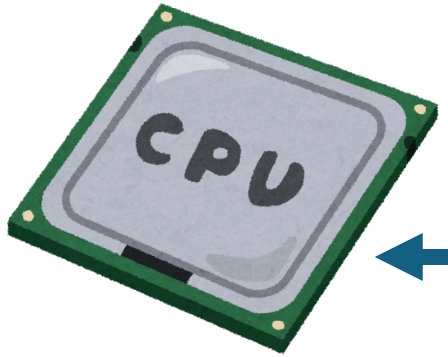
プログラムによるメモリアクセス

CPU からメモリへのアクセスには
MMU (Memory Management Unit) という部品が介在する

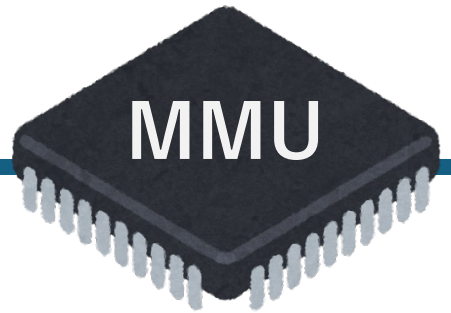
MMU は 仮想アドレスから物理アドレスへの変換を担当

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```




```
rax:0x100000000
```



ポイント

一般的なプログラムが参照する
メモリアドレスは仮想メモリアドレス

OS が起動時に初期化の早い段階で
仮想メモリアドレスを基本とする
CPU モードを有効にする

ちなみに
カーネルも仮想メモリアドレスを
基本として動作しています

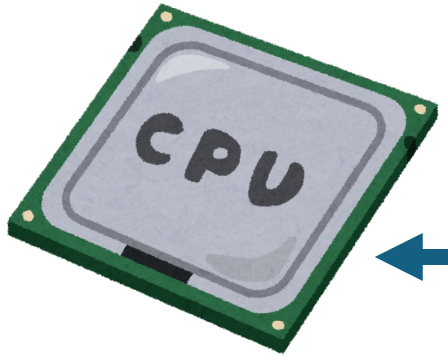
プログラムによるメモリアクセス

CPU からメモリへのアクセスには
MMU (Memory Management Unit) という部品が介在する

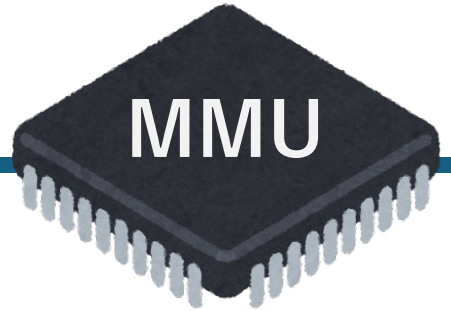
MMU は 仮想アドレスから物理アドレスへの変換を担当

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



rax:0x100000000



ポイント

一般的なプログラムが参照する
メモリアドレスは**仮想**メモリアドレス

OS が起動時に初期化の早い段階で
仮想メモリアドレスを基本とする
CPU モードを有効にする

ちなみに
カーネルも**仮想**メモリアドレスを
基本として動作しています

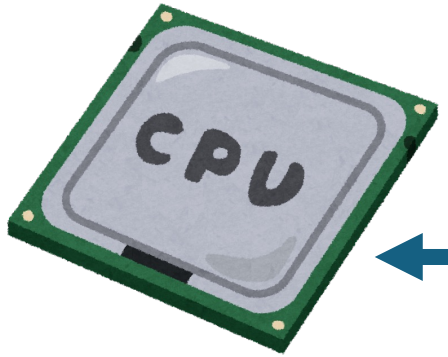
プログラムによるメモリアクセス

CPU からメモリへのアクセスには
MMU (Memory Management Unit) という部品が介在する

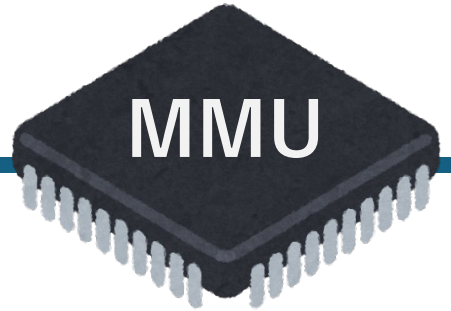
MMU は 仮想アドレスから物理アドレスへの変換を担当

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```




```
rax:0x100000000
```



ポイント

一般的なプログラムが参照する
メモリアドレスは**仮想**メモリアドレス

OS が起動時に初期化の早い段階で
仮想メモリアドレスを基本とする
CPU モードを有効にする

ちなみに
カーネルも**仮想**メモリアドレスを
基本として動作しています

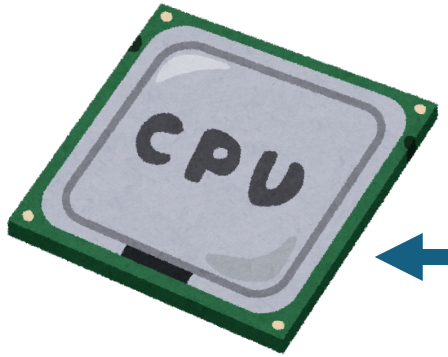
プログラムによるメモリアクセス

CPU からメモリへのアクセスには
MMU (Memory Management Unit) という部品が介在する

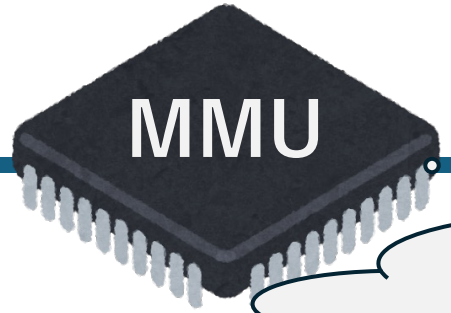
MMU は 仮想アドレスから物理アドレスへの変換を担当

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```




```
rax:0x100000000
```



仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

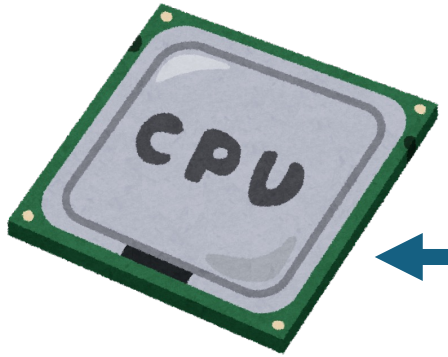
プログラムによるメモリアクセス

CPU からメモリへのアクセスには
MMU (Memory Management Unit) という部品が介在する

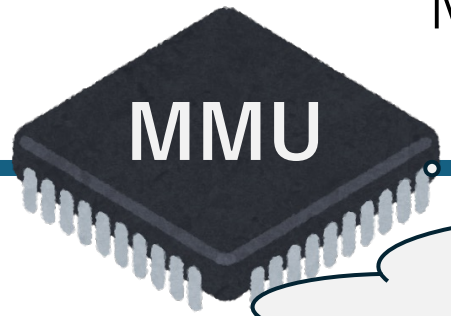
MMU は 仮想アドレスから物理アドレスへの変換を担当

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```




```
rax:0x100000000
```



MMU はアドレス変換に際して
ページテーブルを参照する

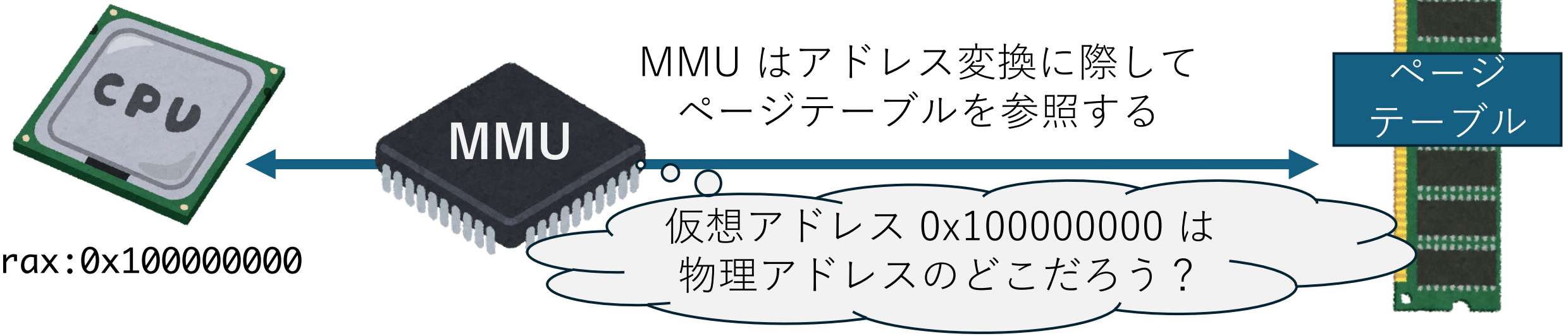
仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

プログラムによるメモリアクセス

ページテーブルはソフトウェアにより
メモリ上に用意される**仮想**アドレスと
物理アドレスの対応を保持するテーブル
（テーブルのフォーマットはハードウェア依存）

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



rax:0x100000000

プログラムによるメモリアクセス

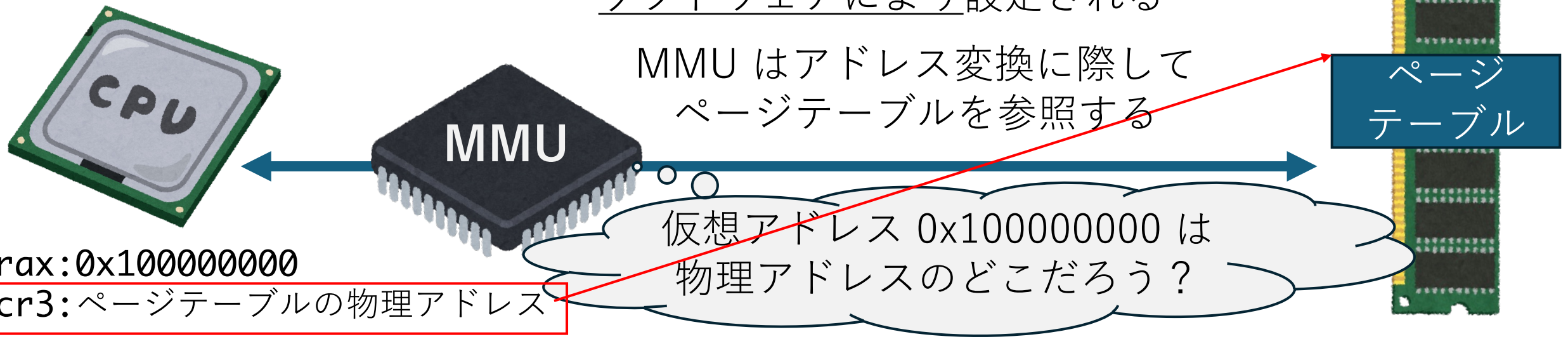
ページテーブルはソフトウェアにより
メモリ上に用意される**仮想**アドレスと
物理アドレスの対応を保持するテーブル
（テーブルのフォーマットはハードウェア依存）

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```

MMU が参照すべき
ページテーブルは
cr3 レジスタを通して
ソフトウェアにより設定される

MMU はアドレス変換に際して
ページテーブルを参照する




```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

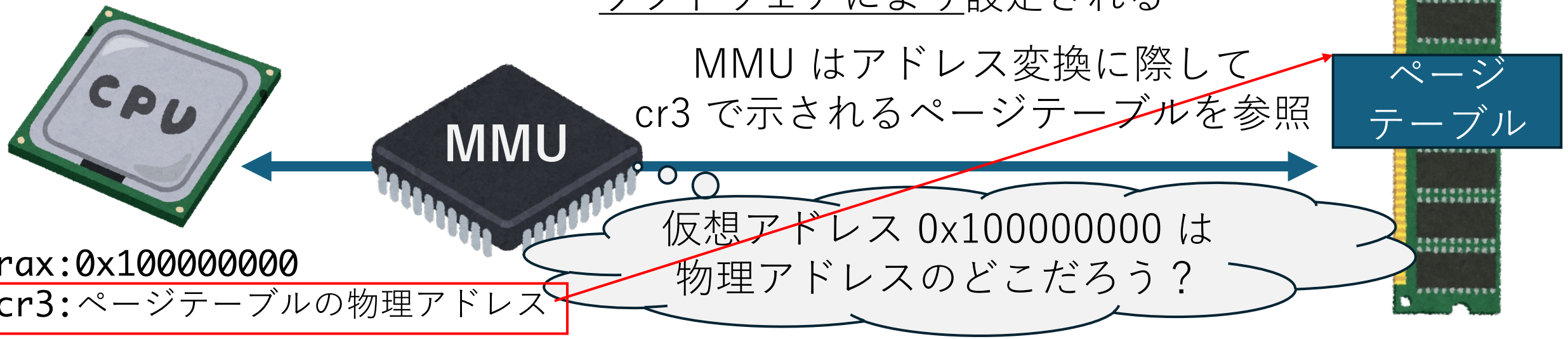
ページテーブルはソフトウェアにより
メモリ上に用意される**仮想**アドレスと
物理アドレスの対応を保持するテーブル
（テーブルのフォーマットはハードウェア依存）

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```

MMU が参照すべき
ページテーブルは
cr3 レジスタを通して
ソフトウェアにより設定される

MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



rax:0x100000000
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

ポイント
ページテーブル自体は
ソフトウェアが用意できる

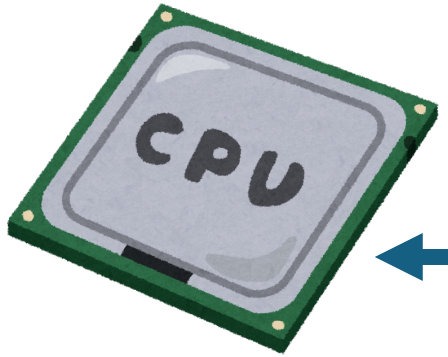
ページテーブルは**ソフトウェアにより**
メモリ上に用意される**仮想**アドレスと
物理アドレスの対応を保持するテーブル
（テーブルのフォーマットはハードウェア依存）

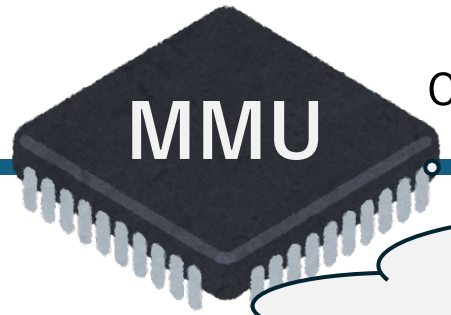
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

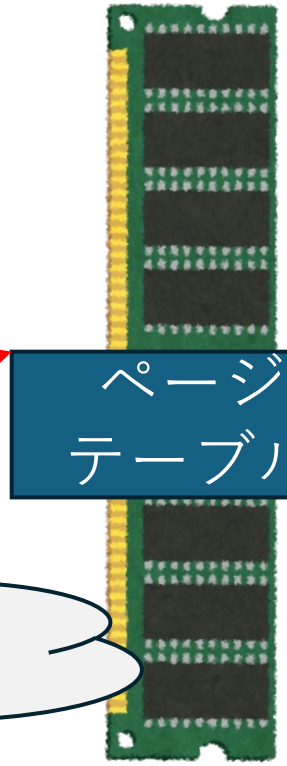
```
movl $0x12345678,(%rax)
```

MMU が参照すべき
ページテーブルは
cr3 レジスタを通して
ソフトウェアにより設定される

MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照







ページ
テーブル

仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

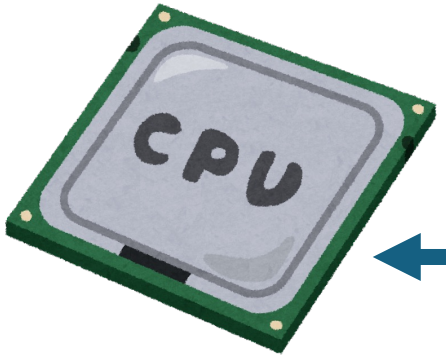
```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

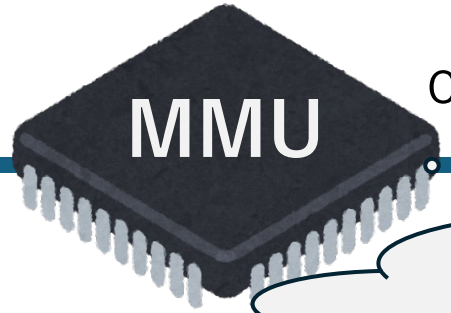
プログラムによるメモリアクセス

仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

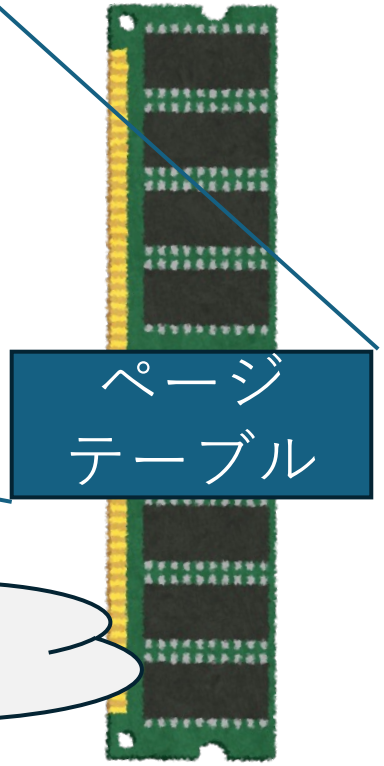
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

rax:0x100000000
cr3:ページテーブルの物理アドレス

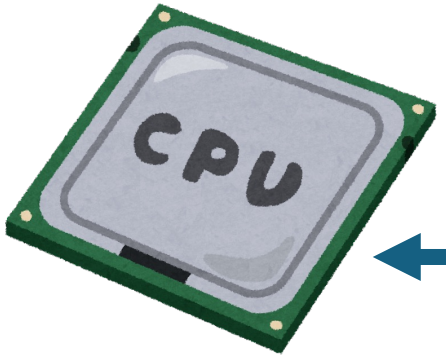
プログラムによるメモリアクセス

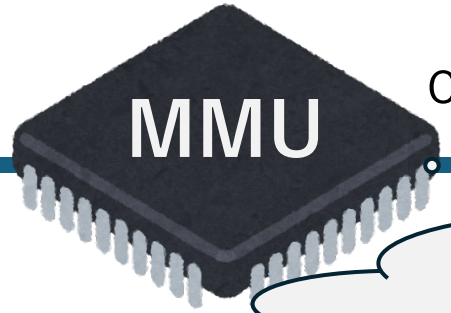
仮想アドレスと物理アドレスの
対応の設定は 4 KB ごとに設定
(4 KB は 0x1000 B)

仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

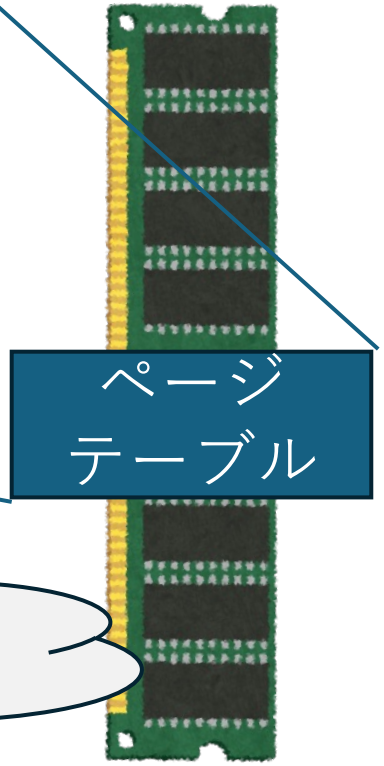
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

rax:0x100000000
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

仮想アドレスと物理アドレスの
対応の設定は 4 KB ごとに設定
(4 KB は 0x1000 B)

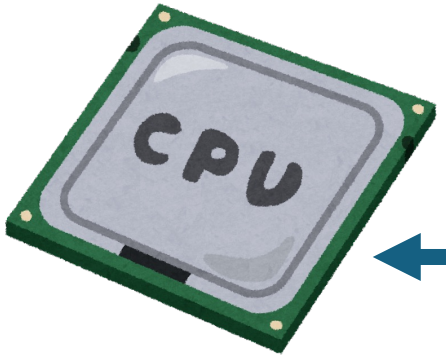
仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

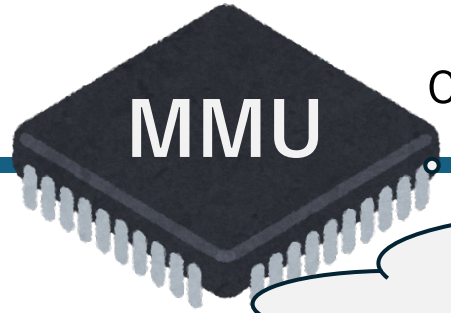
256 TB や 128 PB まで

仮想アドレス空間は設定と
ハードウェアに依存しますが
0 ~ 256 TB や 0 ~ 128 PB が
多いかもしれません

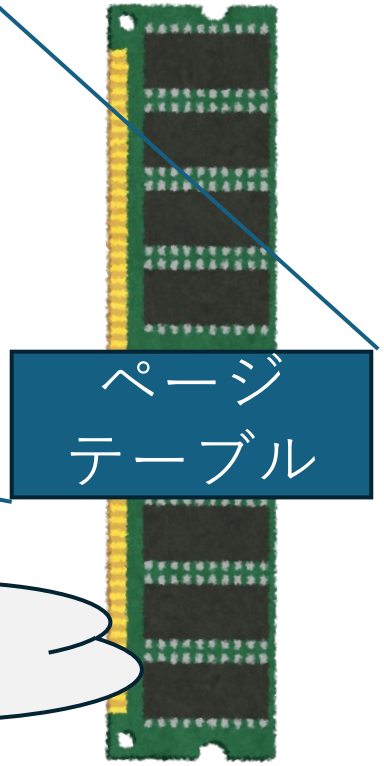
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

rax:0x100000000
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

仮想アドレスと物理アドレスの
対応の設定は 4 KB ごとに設定
(4 KB は 0x1000 B)

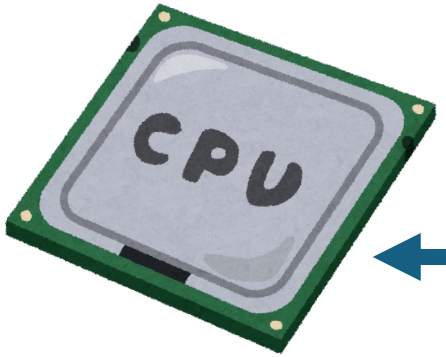
仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

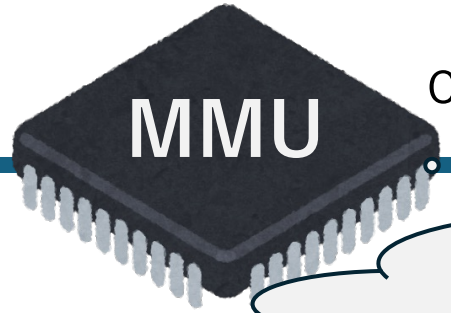
仮想アドレス空間は設定と
ハードウェアに依存しますが
0 ~ 256 TB や 0 ~ 128 PB が
多いかもしれません

仮想アドレス空間の全体に
対応する物理アドレスを
設定する必要はないです

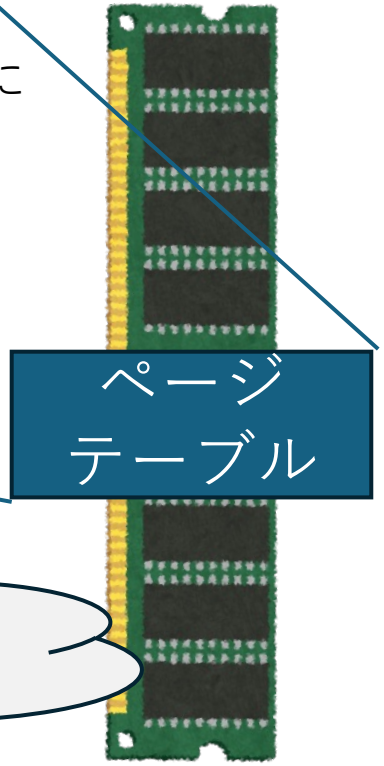
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

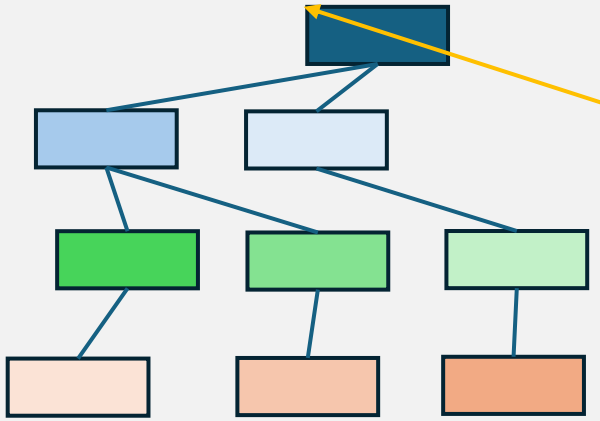
rax:0x100000000
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

仮想アドレスと物理アドレスの

仮想 | 物理

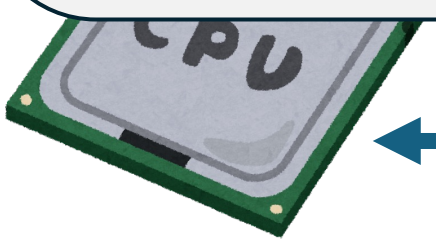
仮想アドレス空間は設定と
アに依存しますが
や 0 ~ 128 PB が
しれません

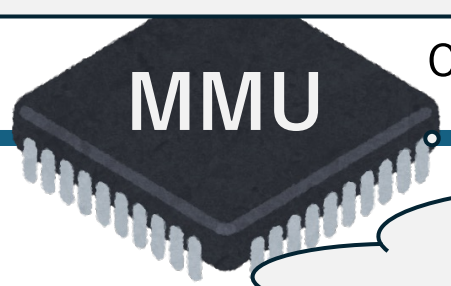


ページテーブルのフォーマットはハードウェア依存ですが
x86-64 では 4 KB のページを物理メモリアドレスの参照を
通して接続した木構造のようなデータ構造になっています

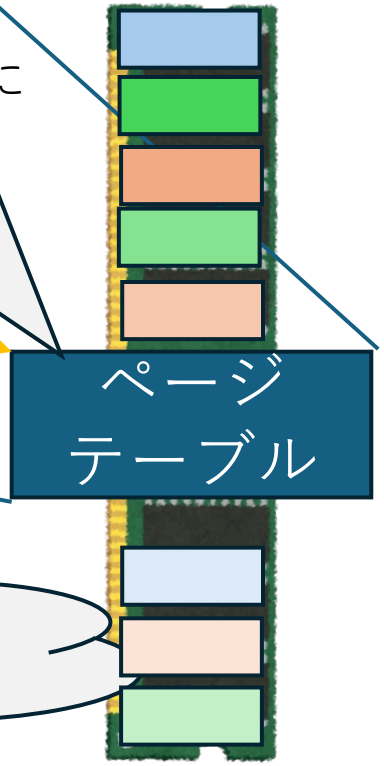
cr3 レジスタに設定するのは木構造の root として扱う
ページの物理メモリアドレスです

より詳細な説明はこちらをご参照ください
https://yasukata.hatenablog.com/entry/2023/04/10/085714





cr3 で示されるページテーブルを参照



ページ
テーブル

仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

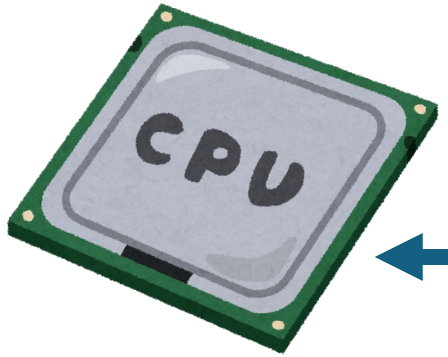
rax:0x100000000
cr3:ページテーブルの物理アドレス

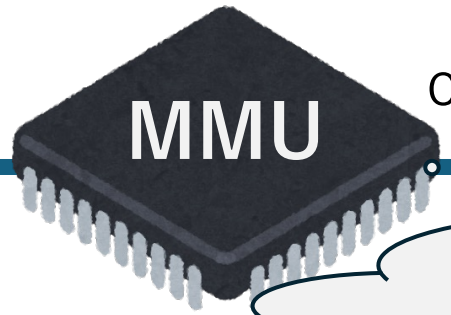
プログラムによるメモリアクセス

仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

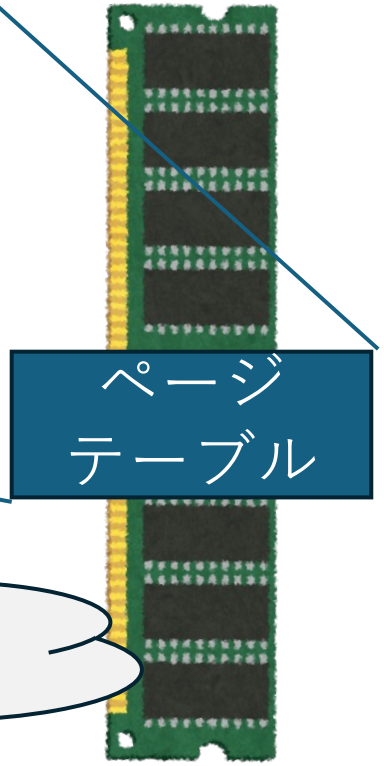
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

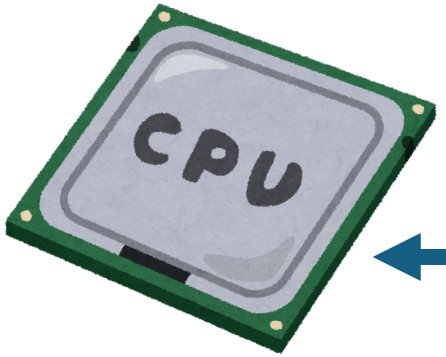
rax:0x100000000
cr3:ページテーブルの物理アドレス

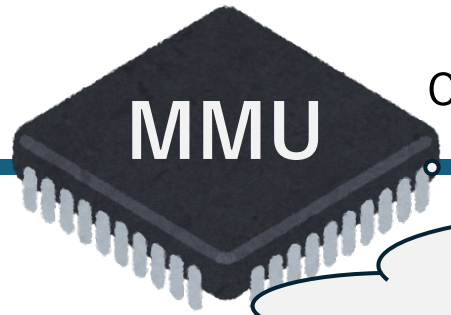
プログラムによるメモリアクセス

仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

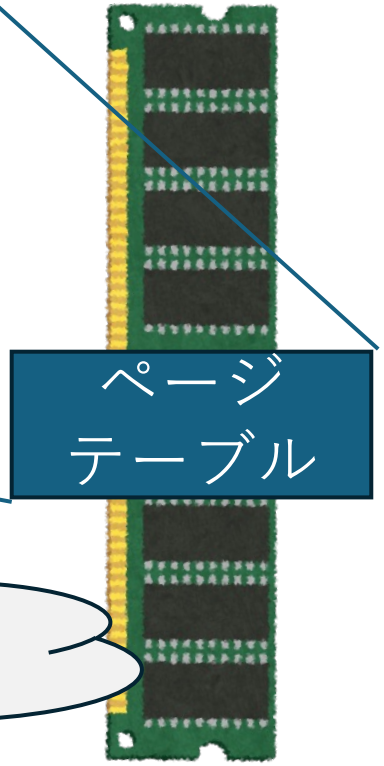
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 と
対応する物理アドレスがない

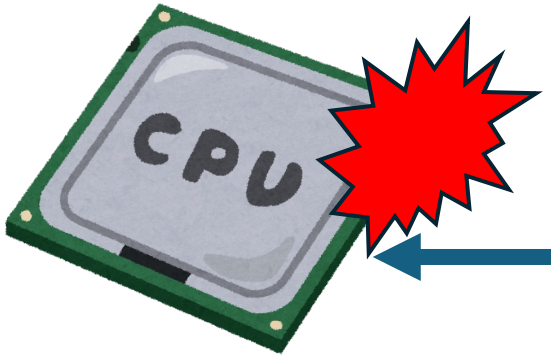
rax:0x100000000
cr3:ページテーブルの物理アドレス

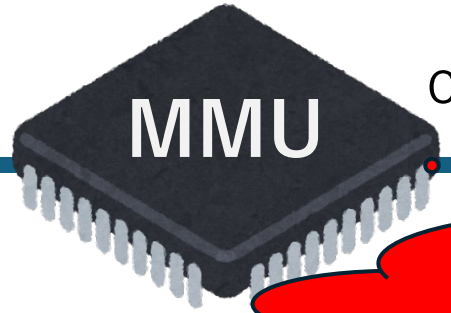
プログラムによるメモリアクセス

仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

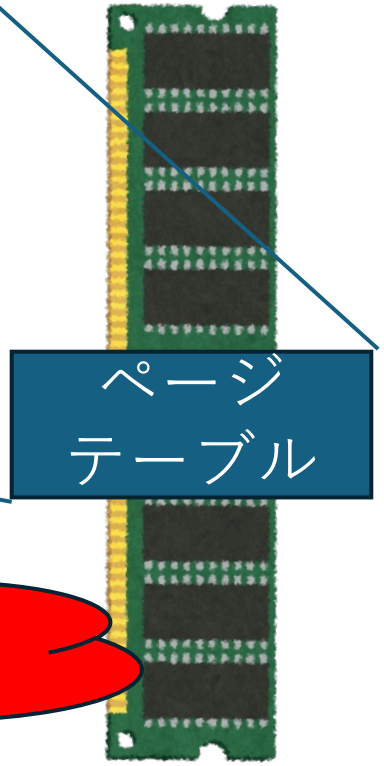
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



Segmentation fault を起動

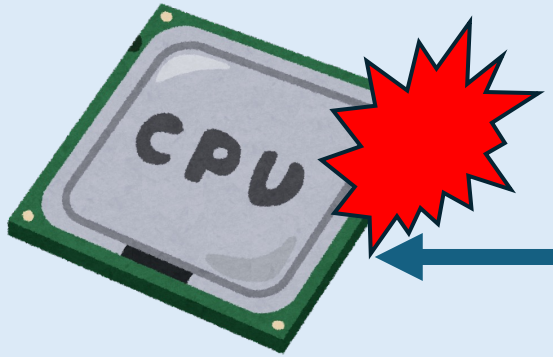
rax:0x100000000
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

実行するとおそらく
Segmentation fault という
エラーで停止して
メモリアクセスができない

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



この命令の実行時に CPU から
メモリアクセスが試みられる

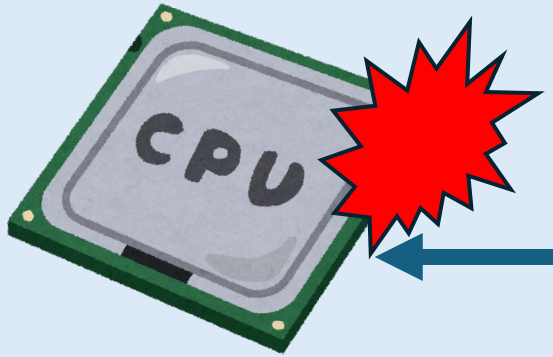
```
rax:0x100000000
```

プログラムによるメモリアクセス

実行するとおそらく
Segmentation fault という
エラーで停止して
メモリアクセスができない

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



この命令の実行時に CPU から
メモリアクセスが試みられる

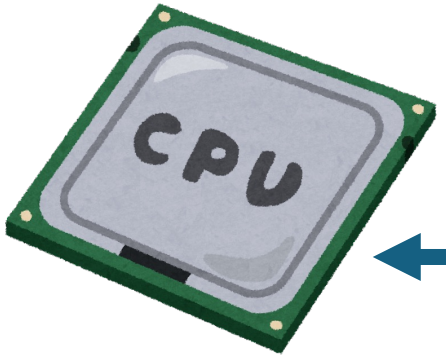
```
rax:0x100000000
```

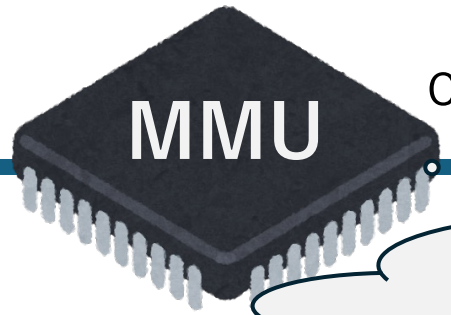
プログラムによるメモリアクセス

仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

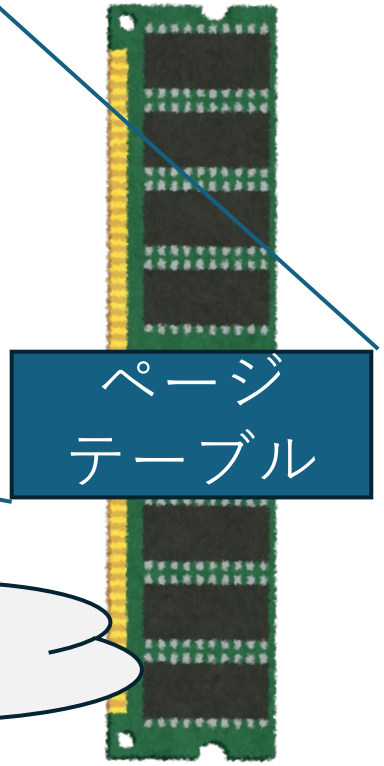
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

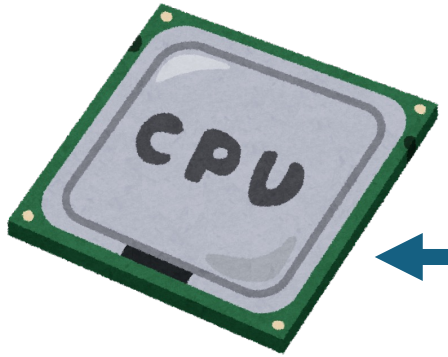
プログラムによるメモリアクセス

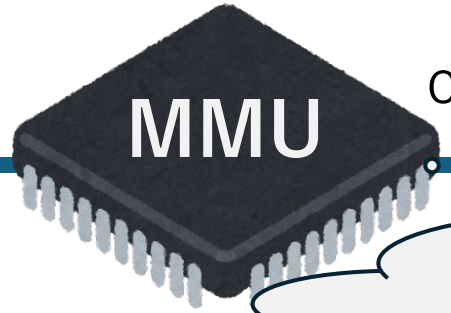
対応する**物理アドレス**として
0x2000 が設定されていた場合

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

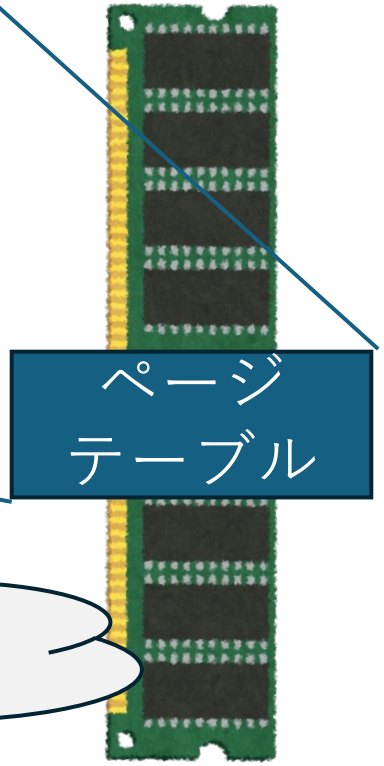
```
movl $0x12345678,(%rax)
```

仮想	物理
0x0000	
0x1000	
…	
0x100000000	0x2000
…	





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスのどこだろう？

```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

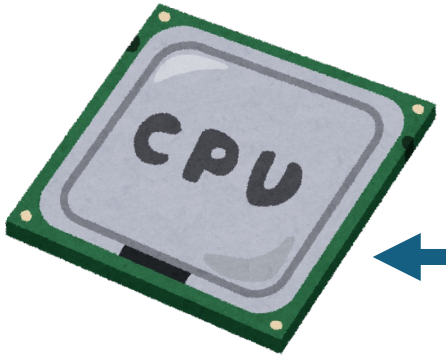
プログラムによるメモリアクセス

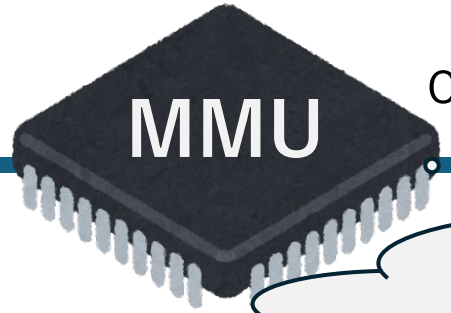
対応する**物理アドレス**として
0x2000 が設定されていた場合

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

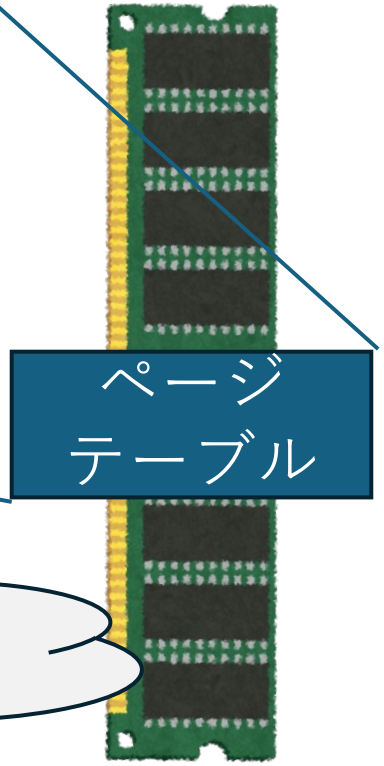
```
movl $0x12345678,(%rax)
```

仮想	物理
0x0000	
0x1000	
…	
0x100000000	0x2000
…	





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスの 0x2000

```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

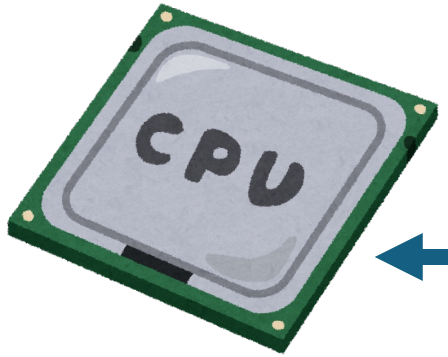
対応する**物理アドレス**として
0x2000 が設定されていた場合

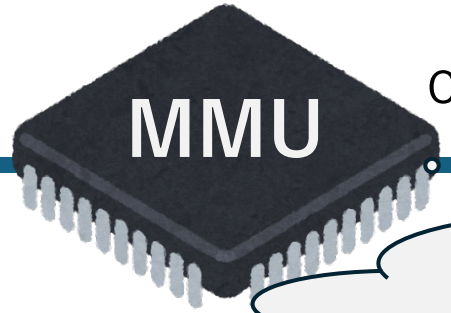
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```

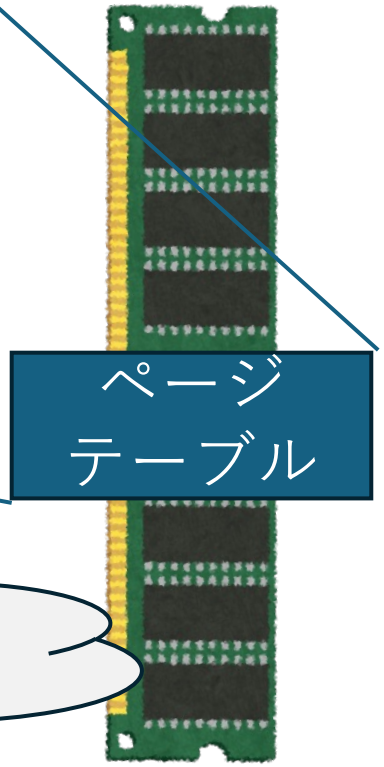
仮想	物理
0x0000	
…	
0x100000000	0x2000
…	

物理アドレス 0x2000 へ
0x12345678 を書き込み





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスの 0x2000

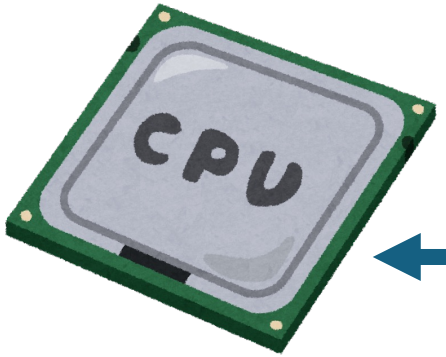
```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

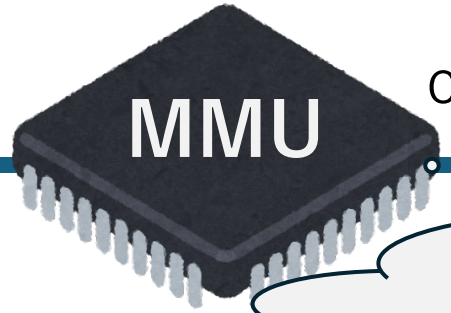
プログラムによるメモリアクセス

対応する**物理アドレス**として
0x2000 が設定されていた場合

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



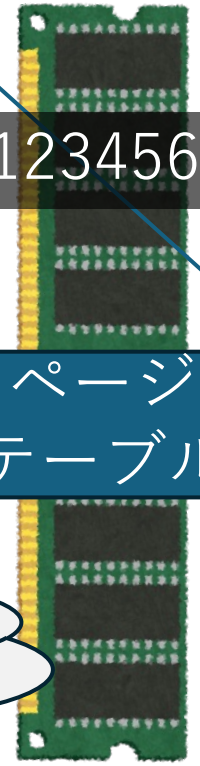



```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

仮想	物理
0x0000	
…	
0x100000000	0x2000
…	

物理アドレス 0x2000 へ
0x12345678 を書き込み

0x2000 0x12345678



ページ
テーブル

MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照

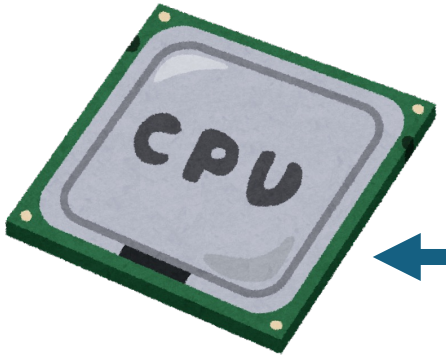
仮想アドレス 0x100000000 は
物理アドレスの 0x2000

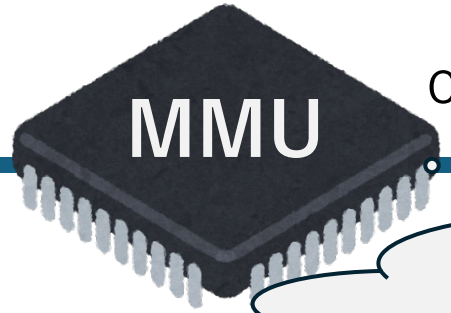
プログラムによるメモリアクセス

仮想	物理
0x0000	
0x1000	
…	
0x100000000	0x2000
…	

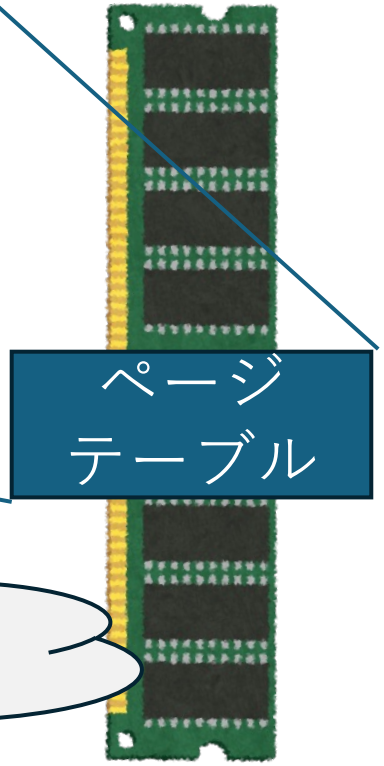
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスの 0x2000

```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

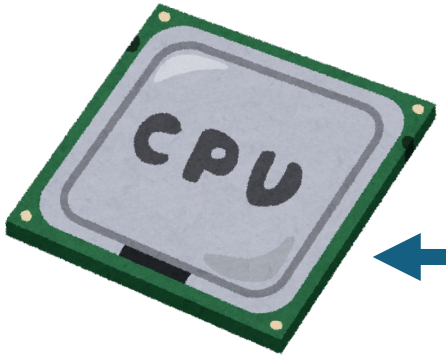
プログラムによるメモリアクセス

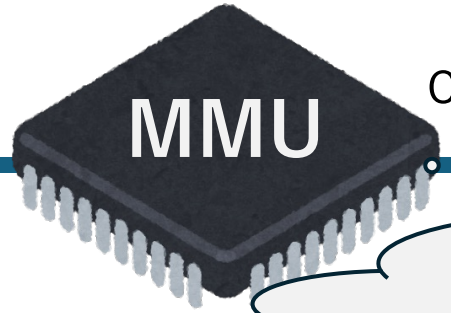
非特権モード時の
アクセスの可否も
設定できます

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

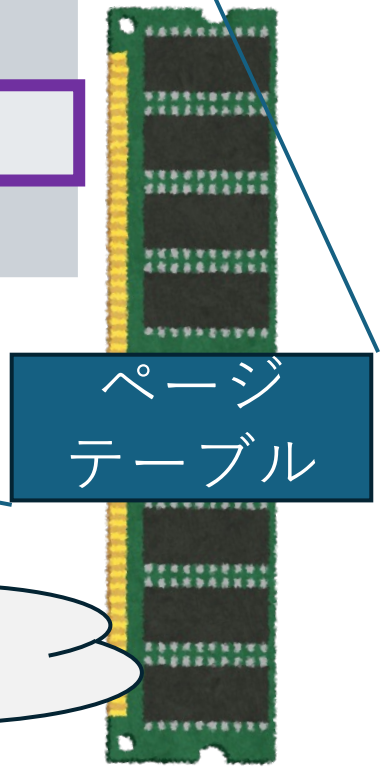
```
movl $0x12345678,(%rax)
```

仮想	物理	非特権モード時
0x0000		
0x1000		
…		
0x100000000	0x2000	アクセス可能
…		





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスの 0x2000

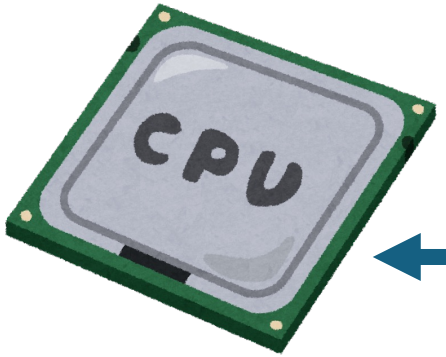
`rax:0x100000000`
`cr3:`ページテーブルの物理アドレス

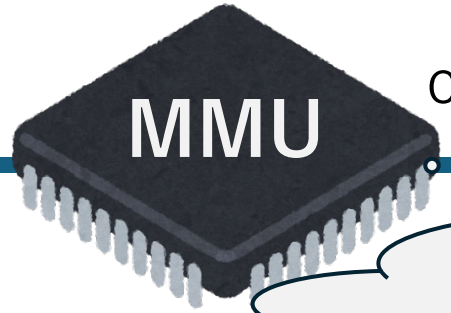
プログラムによるメモリアクセス

仮想	物理
0x0000	
0x1000	
…	
0x100000000	0x2000
…	

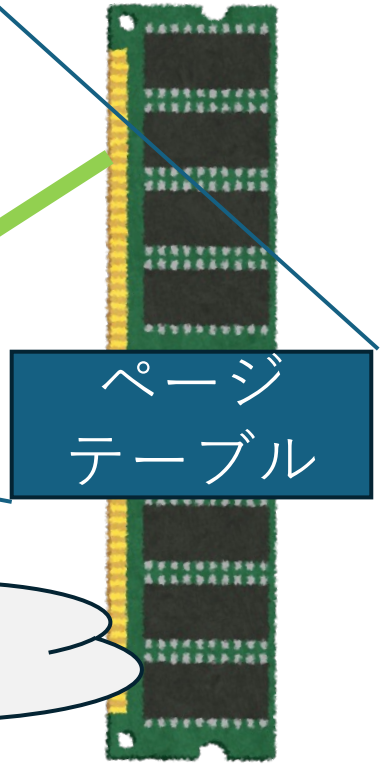
仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスの 0x2000

rax:0x100000000
cr3:ページテーブルの物理アドレス

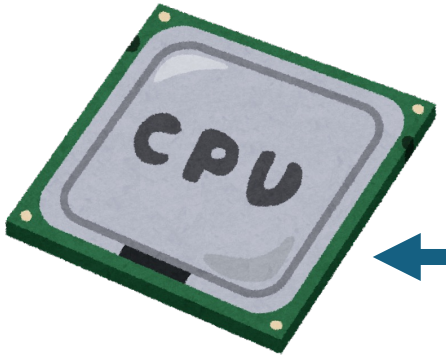
プログラムによるメモリアクセス

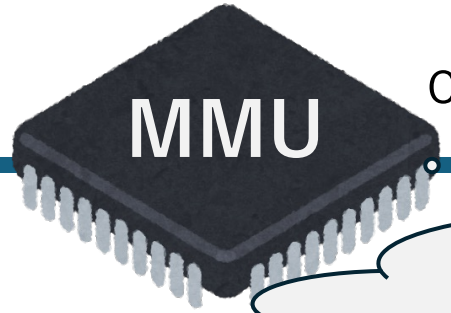
ポイント１

実行中のプログラムは
MMU が参照している
ページテーブルに記載のある
物理メモリアドレスしか
アクセスできない

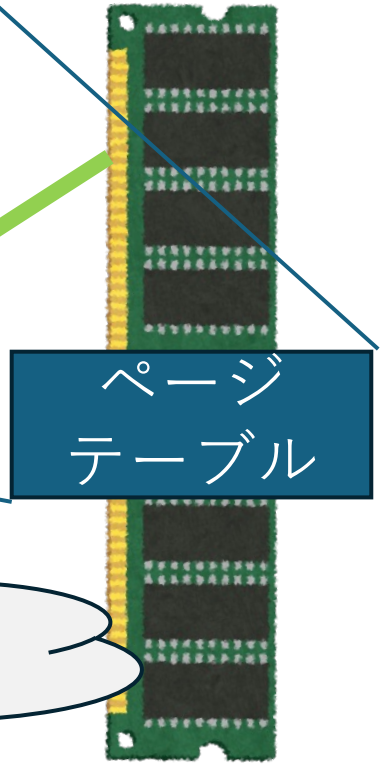
仮想	物理
0x0000	
0x1000	
…	
0x100000000	0x2000
…	

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスの 0x2000

```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

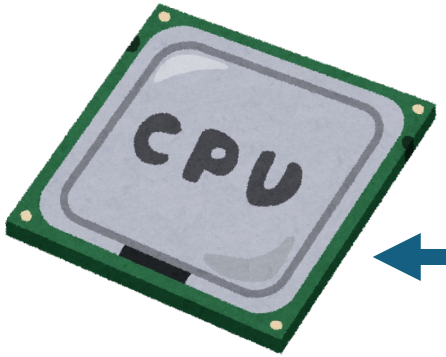
ポイント１

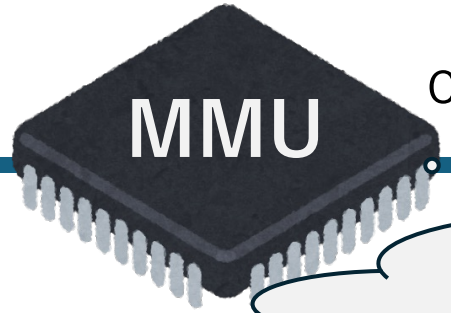
実行中のプログラムは
MMU が参照している
ページテーブルに記載のある
物理メモリアドレスしか
アクセスできない

仮想	物理
0x0000	
0x1000	
…	
0x100000000	
…	

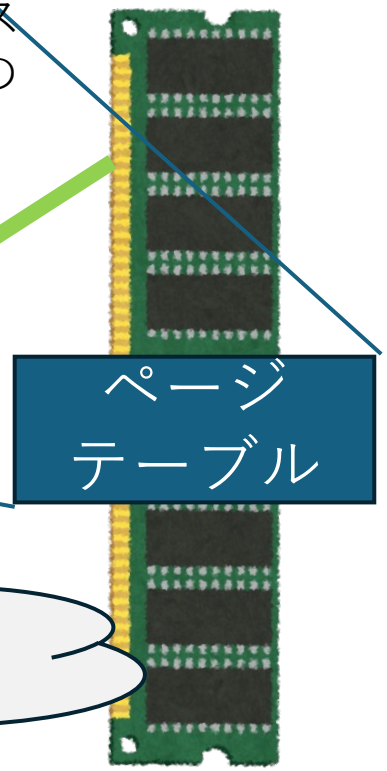
例えば物理メモリアドレスが
一つも設定されていなければ
プログラムが仮想メモリアドレス
0 ~ 上限（256 TB や 128 PB）の
どこにアクセスしても
Segmentation fault になる

```
movl $0x12345678,(%rax)
```





MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスの 0x2000

```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

ポイント１

実行中のプログラムは
MMU が参照している
ページテーブルに記載のある
物理メモリアドレスしか
アクセスできない

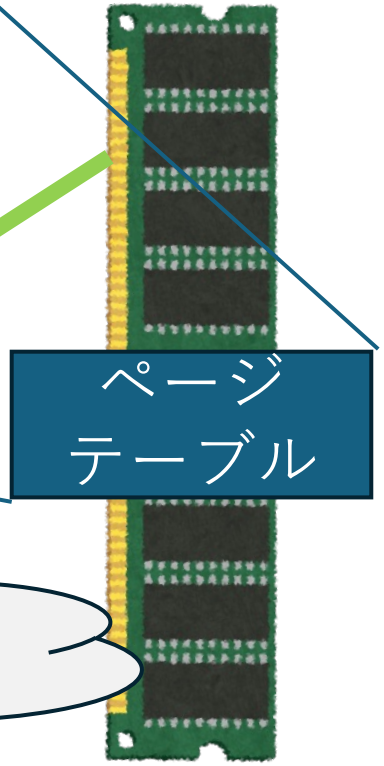
仮想	物理
0x0000	
0x1000	
…	
0x100000000	0x2000

mov

ポイント２

非特権モードでは cr3 レジスタの値を
変更する CPU 命令を実行できない
（**特権モード**であれば可能）

はアドレス変換に際して
されるページテーブルを参照



仮想アドレス 0x100000000 は
物理アドレスの 0x2000

rax:0x100000000
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

ポイント１

実行中のプログラムは
MMU が参照している
ページテーブルに記載のある
物理メモリアドレスしか
アクセスできない

仮想	物理
0x0000	
0x1000	
…	
0x10000000	

特権モードで動作するカーネルは
ページテーブルの操作を通して
非特権モードで動作するプログラムが
アクセス可能なメモリ領域を制限できる

mov

ポイント２

非特権モードでは cr3 レジスタの値を
変更する CPU 命令を実行できない
（**特権モード**であれば可能）

はアドレス変換に際して
されるページテーブルを参照

ページ
テーブル

仮想アドレス 0x100000000 は
物理アドレスの 0x2000

rax:0x100000000
cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

ポイント１

実行中のプログラムは
MMU が参照している
ページテーブルに記載のある
物理メモリアドレスしか
アクセスできない

仮想	物理
0x0000	
0x1000	
…	
0x10000000	

特権モードで動作するカーネルは
ページテーブルの操作を通して
非特権モードで動作するプログラムが
アクセス可能なメモリ領域を制限できる

ポイント２

非特権モードでは cr3 レジスタの値を
変更する CPU 命令を実行できない
（**特権モード**であれば可能）

制限に必要なポイント

ページテーブルが置かれた
物理メモリアドレスを
非特権モード時に参照される
ページテーブルに記載して
アクセス可能にしないこと

mov

はア

される

仮想アドレス

物理アド

rax:0x100000000

cr3:ページテーブルの物理アドレス

プログラムによるメモリアクセス

ポイント１

実行中のプログラムは
MMU が参照している
ページテーブルに記載のある
物理メモリアドレスしか
アクセスできない

仮想	物理
0x0000	
0x1000	
…	
0x10000000	

特権モードで動作するカーネルは
ページテーブルの操作を通して
非特権モードで動作するプログラムが
アクセス可能なメモリ領域を制限できる

ポイント２

非特権モードでは cr3 レジスタの値を

cr3 は変更できなくても
ページテーブルを編集できる場合には
アクセスしたい物理メモリアドレスを
書き込めばアクセスできてしまうので

制限に必要なポイント

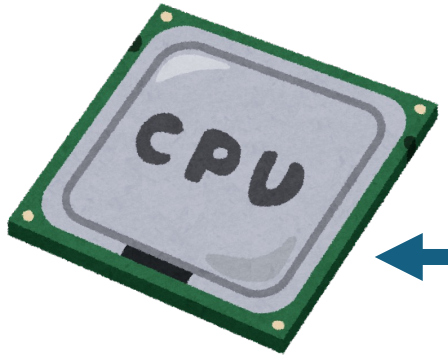
ページテーブルが置かれた
物理メモリアドレスを
非特権モード時に参照される
ページテーブルに記載して
アクセス可能にしないこと

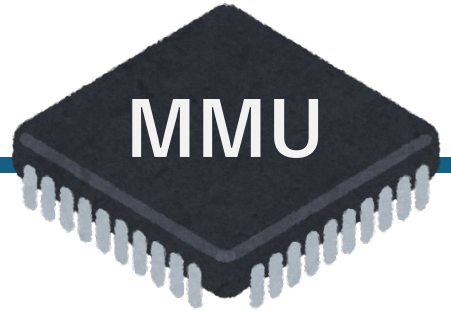
mmap システムコールでカーネルに仮想アドレス 0x100000000 へ対応する物理メモリの確保と設定をリクエストした場合

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
int main(void)
{
    void *mem = mmap((void *) 0x100000000,
                     0x1000, PROT_READ|PROT_WRITE,
                     MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,
                     -1, 0);
    if (mem != (void *) 0x100000000 || mem == MAP_FAILED) {
        printf("mmap failed %p¥n", mem);
        exit(1);
    }
    *((int *) 0x100000000) = 0x12345678;
    printf("success¥n");
}
```

セグメンテーション違反が発生せず success と表示されれば成功

一般的な OS での運用

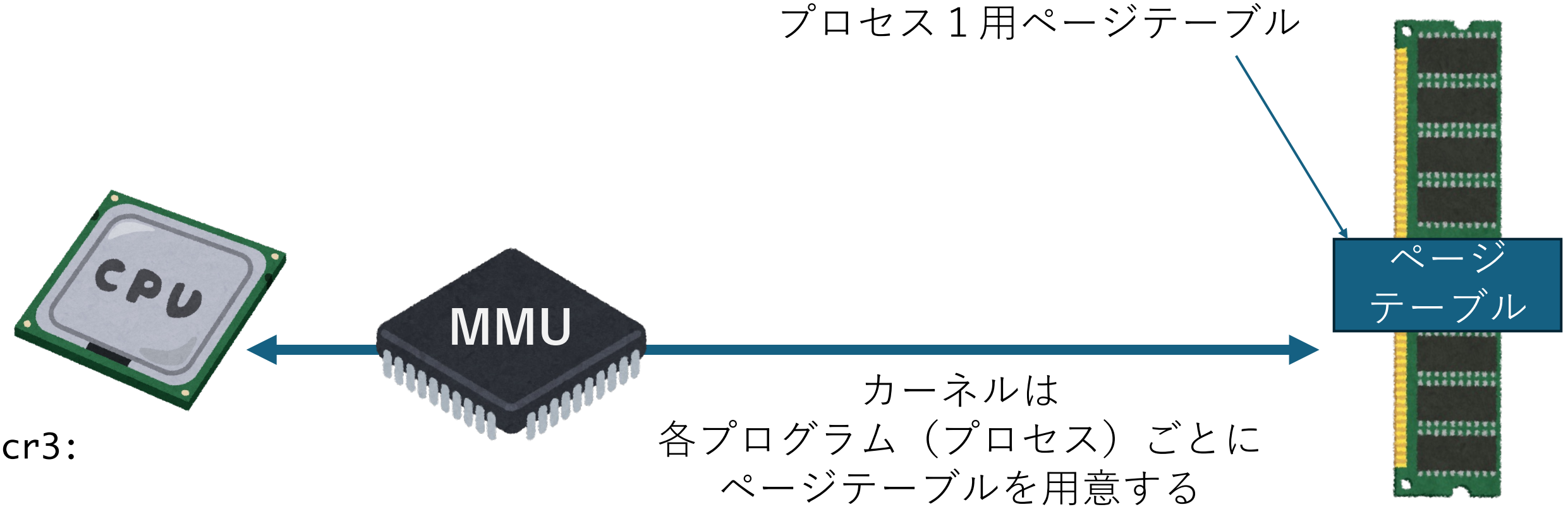




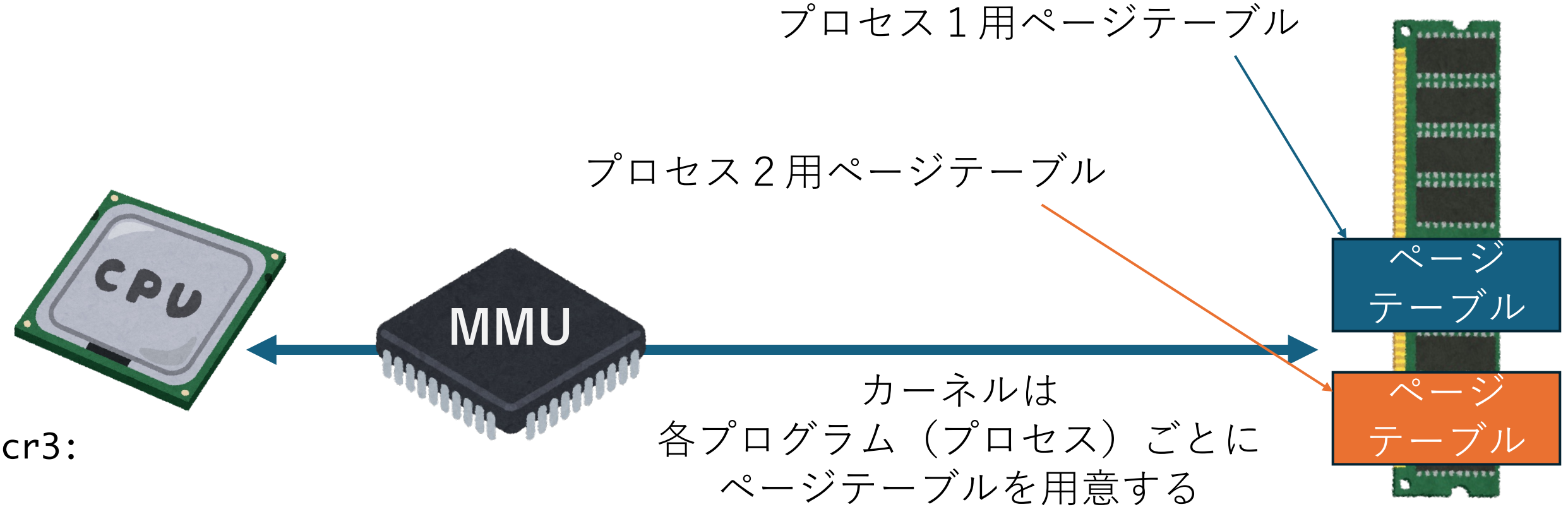
cr3:

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

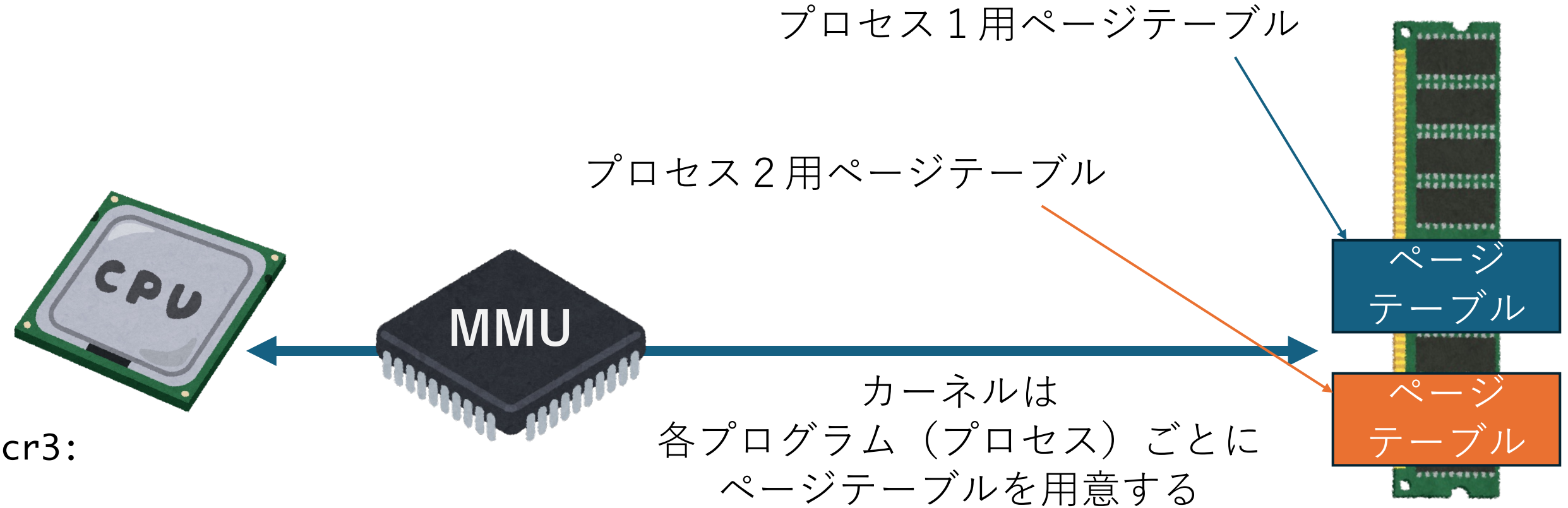

プロセス１用ページテーブル
CPU
MMU
ページ
テーブル
cr3:
カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用


プロセス１用ページテーブル
プロセス２用ページテーブル
CPU
MMU
ページ
テーブル
ページ
テーブル
cr3:
カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

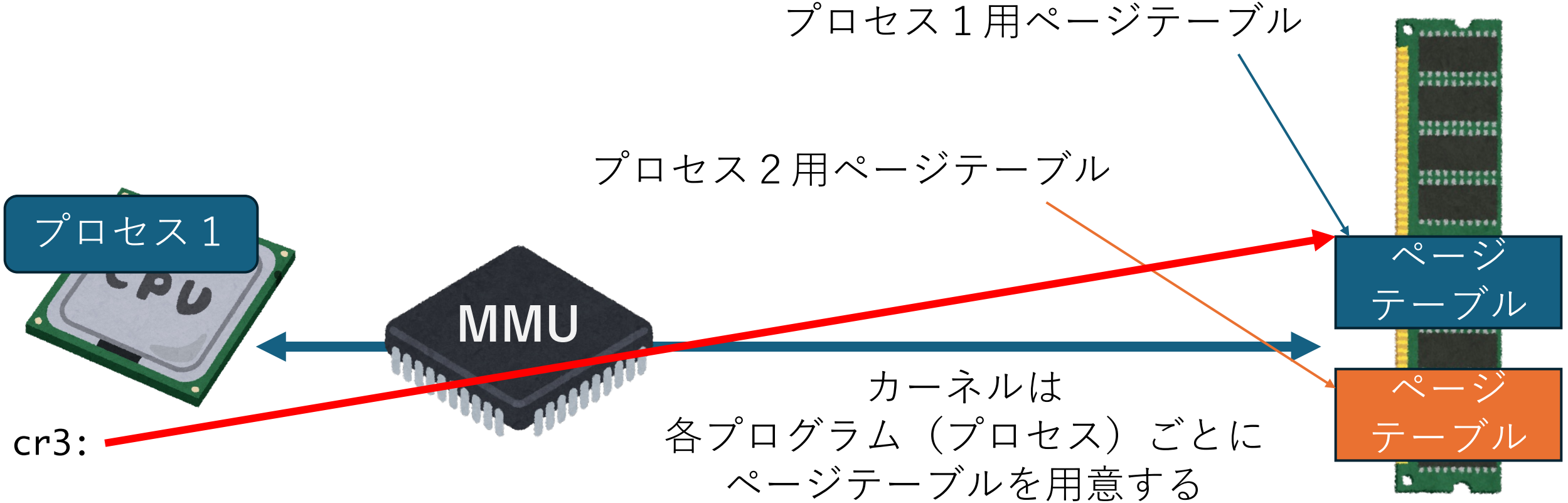
一般的な OS での運用

カーネルは実行しているプロセスの切り替え毎に
cr3 の値を書き換えてMMU が参照するページテーブルを切り替える



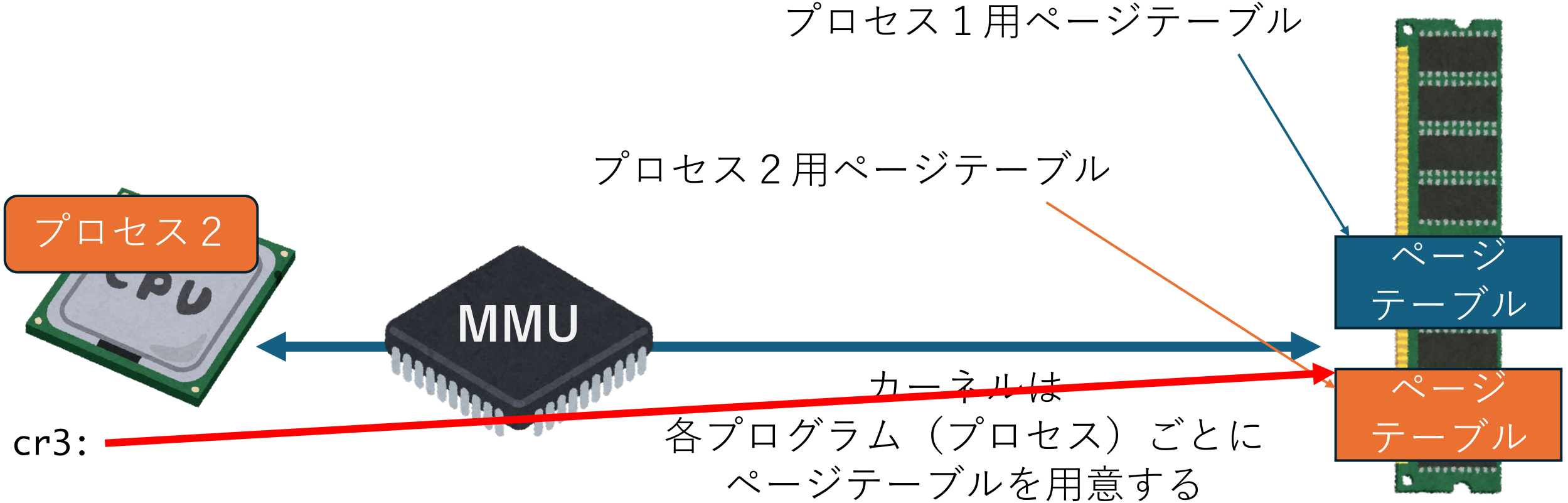
一般的な OS での運用

カーネルは実行しているプロセスの切り替え毎に
cr3 の値を書き換えてMMU が参照するページテーブルを切り替える



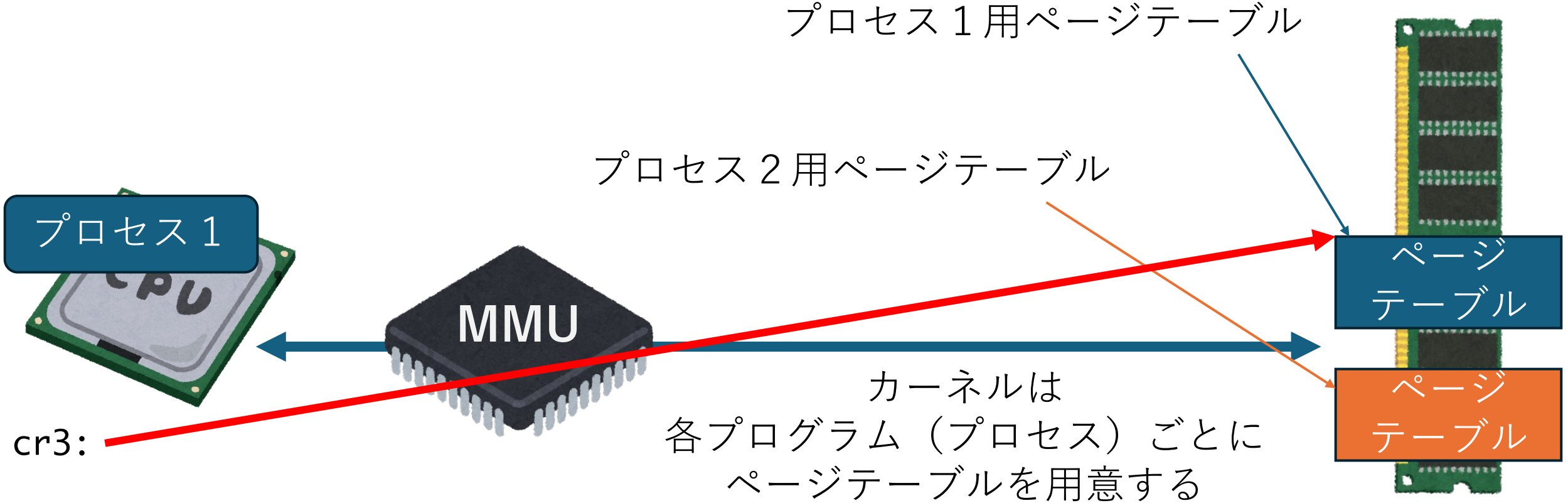
一般的な OS での運用

カーネルは実行しているプロセスの切り替え毎に
cr3 の値を書き換えてMMU が参照するページテーブルを切り替える



一般的な OS での運用

カーネルは実行しているプロセスの切り替え毎に
cr3 の値を書き換えてMMU が参照するページテーブルを切り替える



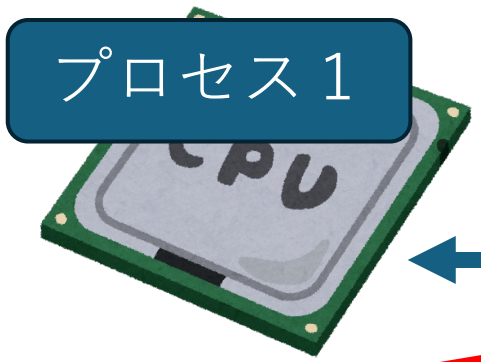
一般的な OS での運用

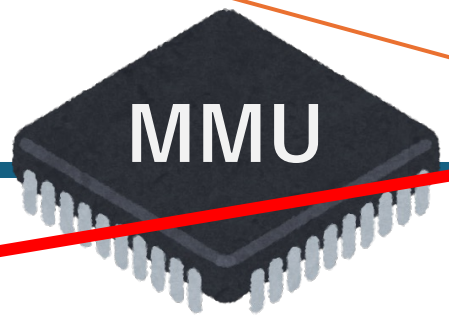
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

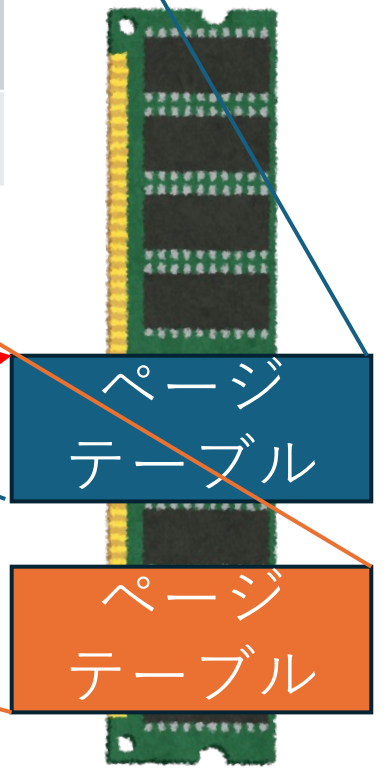
プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル







cr3:

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

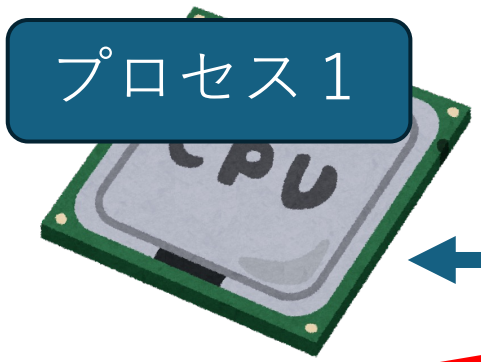
一般的な OS での運用

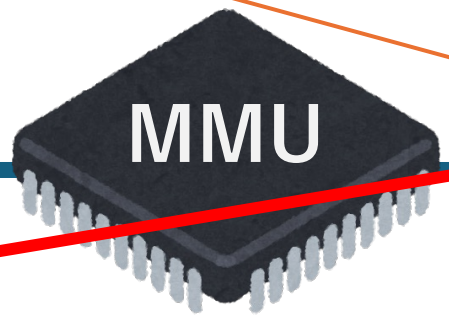
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

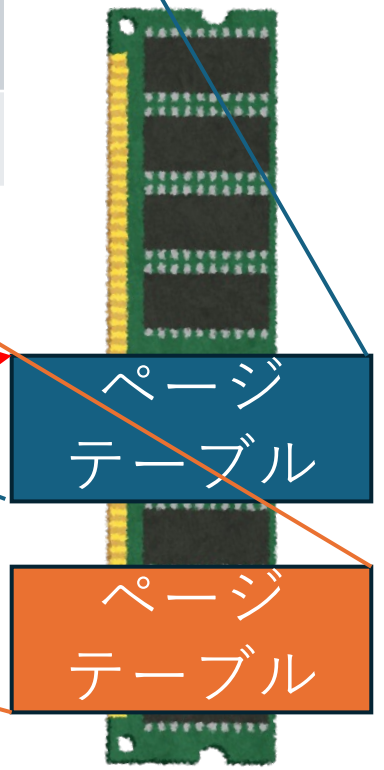
プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル







cr3:

プロセス１が実行中

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

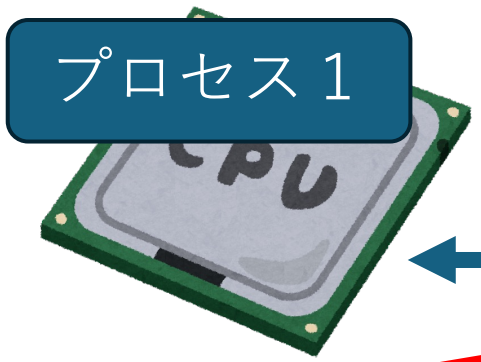
一般的な OS での運用

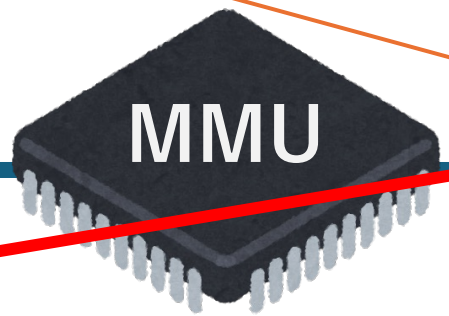
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

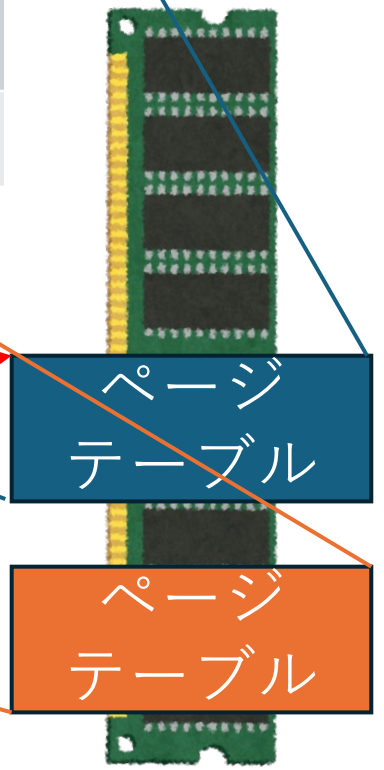
プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル







cr3:
仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

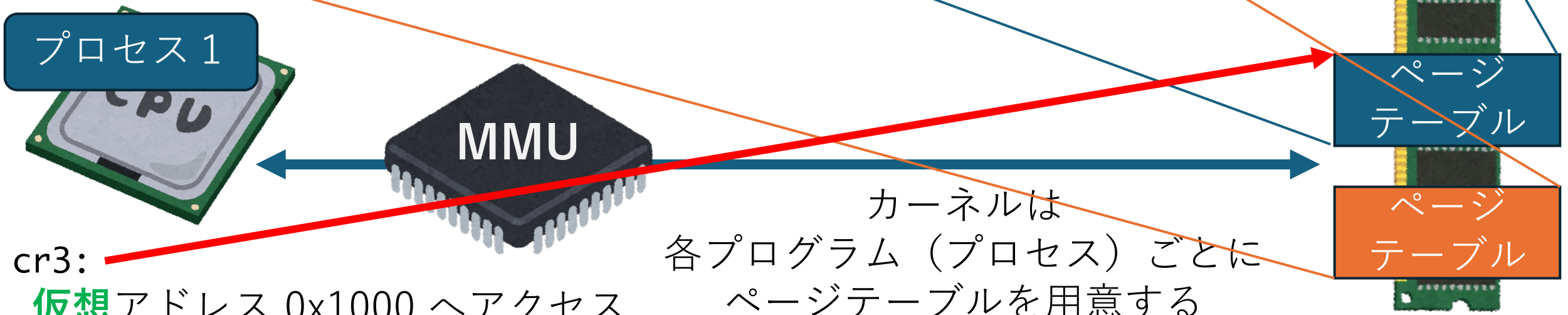
一般的な OS での運用

仮想	物理
0x0000	
0x1000	0x3000
…	
…	

プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル



cr3:

仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

物理アドレス 0x2000 へアクセス

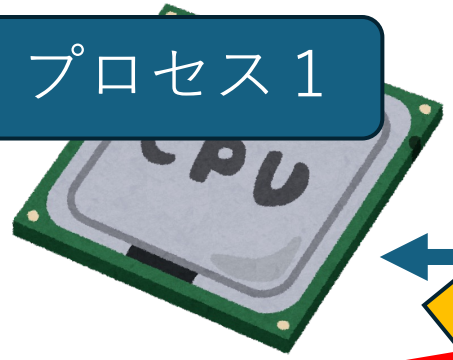
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

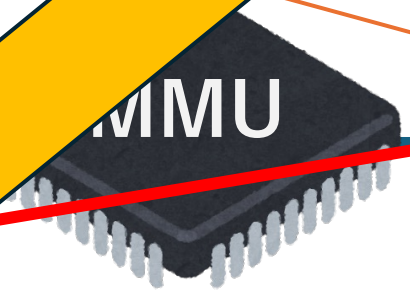
プロセス２用ページテー

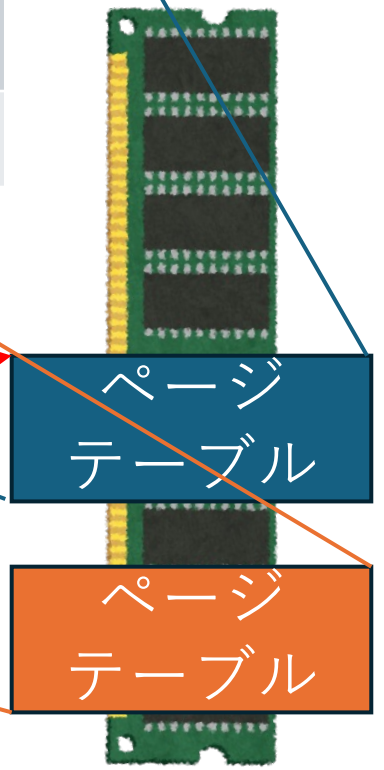
仮想	物理
0x0	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル

プロセス１







ページテーブル

ページテーブル

cr3:

仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

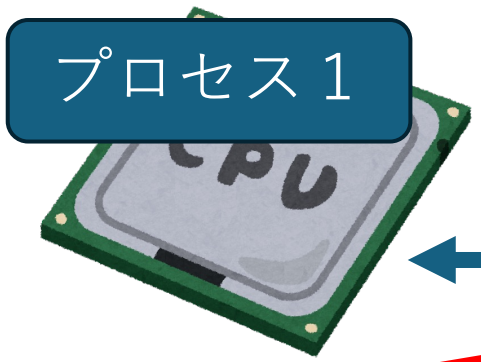
一般的な OS での運用

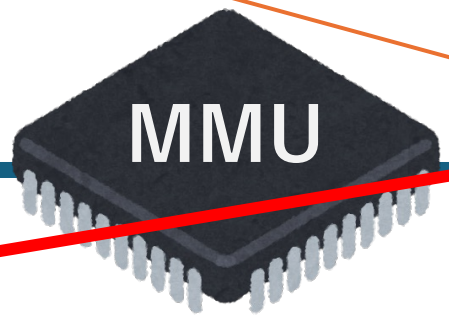
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

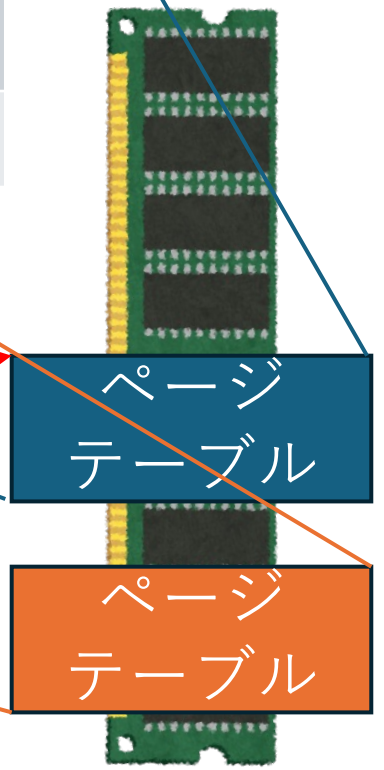
プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル







cr3:

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

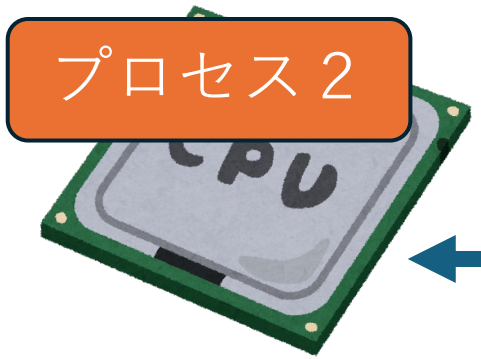
一般的な OS での運用

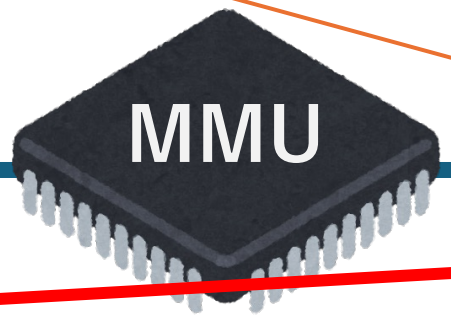
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

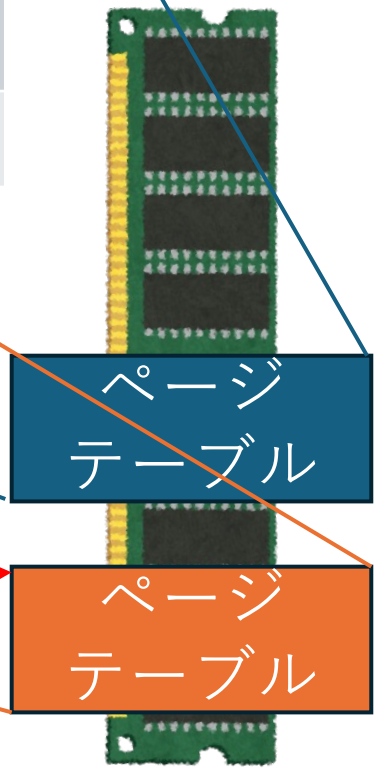
プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル







cr3:
プロセスが切り替えられた

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

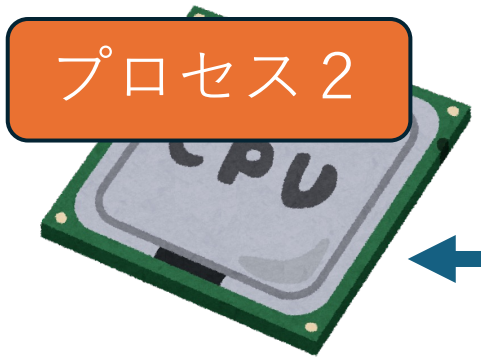
一般的な OS での運用

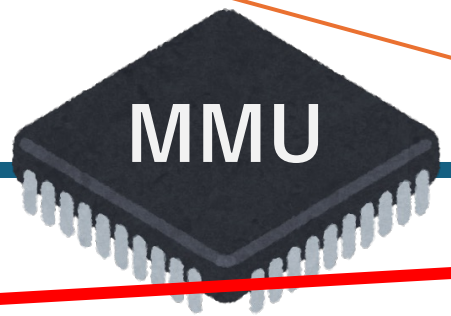
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

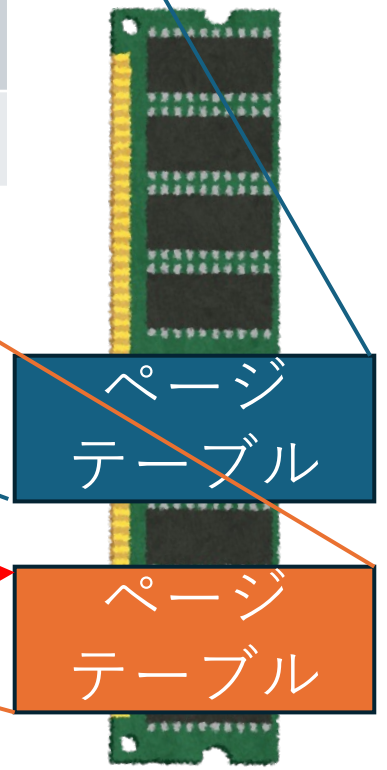
プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル







cr3:

仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

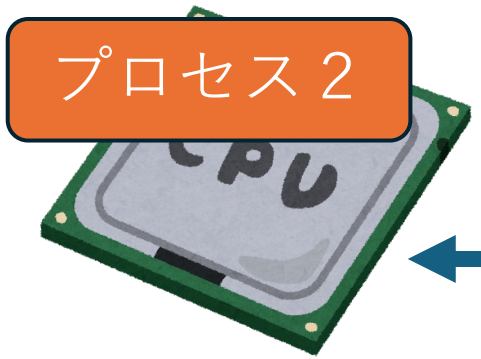
一般的な OS での運用

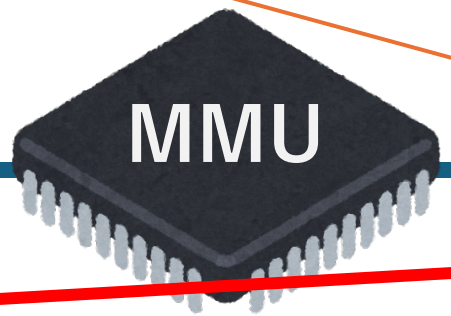
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

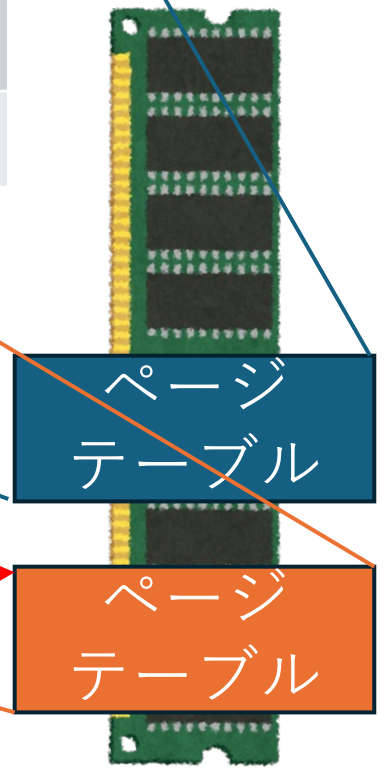
プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル







cr3:

仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

物理アドレス 0x3000 へアクセス

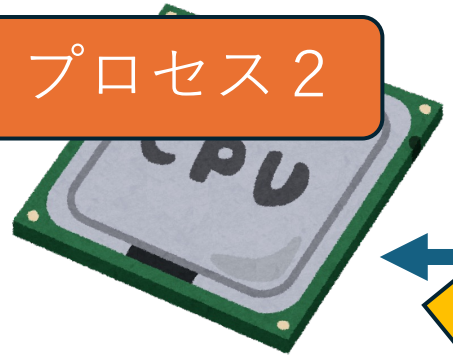
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

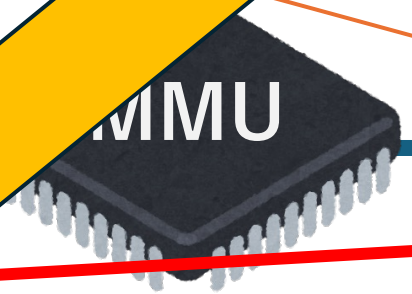
プロセス２用ページテー

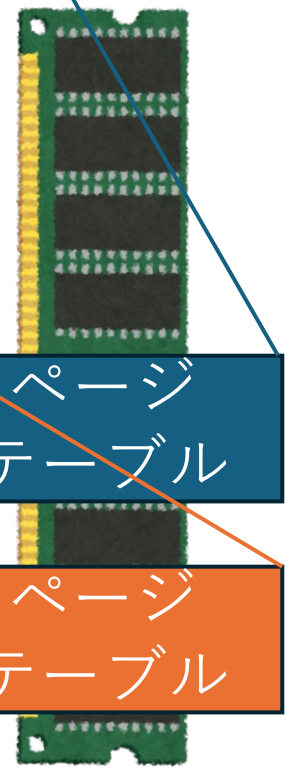
仮	物理
0x	
00	0x2000
.	
…	

プロセス１用ページテーブル

プロセス２







ページテーブル

ページテーブル

cr3:

仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

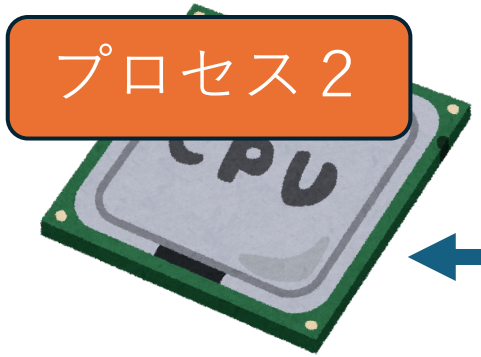
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

仮想	物理
0x0000	
0x1000	0x2000
…	

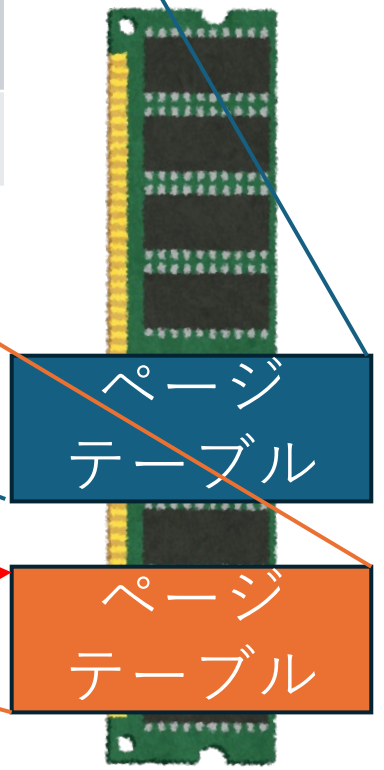
プロセス２用ページテーブル

同じ**仮想**アドレス 0x1000 へのアクセスも
ページテーブルが違えば
別の**物理**アドレスへのアクセスになります

プロセス２



MMU



ページテーブル

ページテーブル

cr3:

仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

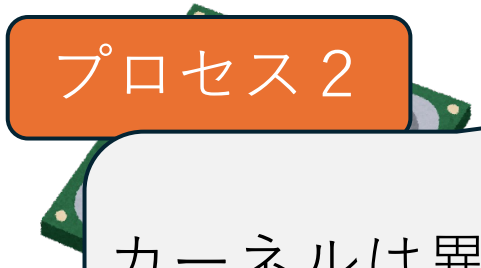
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

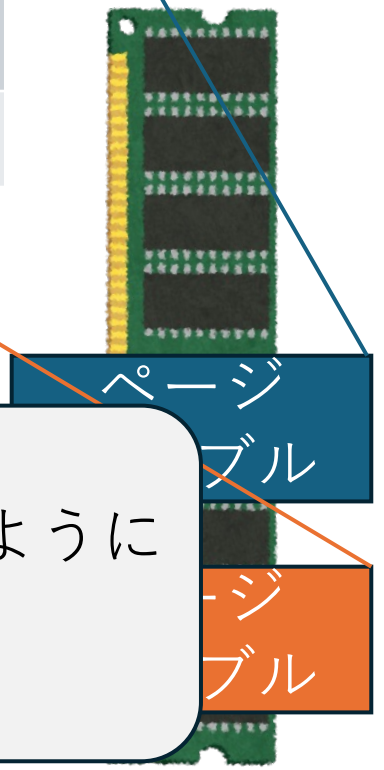
仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス２用ページテーブル

同じ**仮想**アドレス 0x1000 へのアクセスも
ページテーブルが違えば
別の**物理**アドレスへのアクセスになります

プロセス２



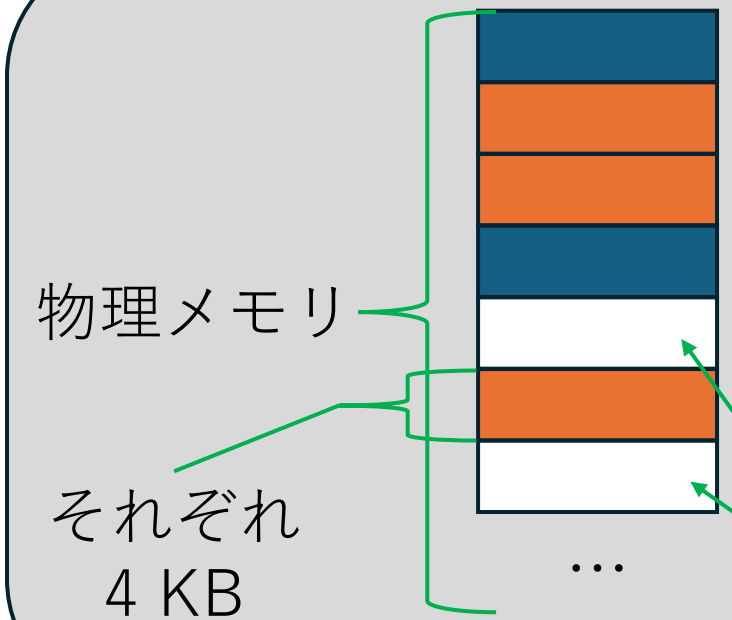


ポイント

カーネルは異なるプロセスが同じ物理メモリ領域へアクセスできないように
注意しながらページテーブルを用意します
これによりプロセス間の分離（isolation）が担保されます

cr

仮想アドレス 0x1000 へアクセス



プロセス１へ割り当て済み
プロセス２へ割り当て済み
未使用

カーネルはどの物理メモリ領域が
利用されているかを把握しており
新たにプロセスに割り当てる時は
未使用の領域から割り当てます

プロセス２

ページ
ブル

ポイント

カーネルは異なるプロセスが同じ物理メモリ領域へアクセスできないように
注意しながらページテーブルを用意します
これによりプロセス間の分離（isolation）が担保されます

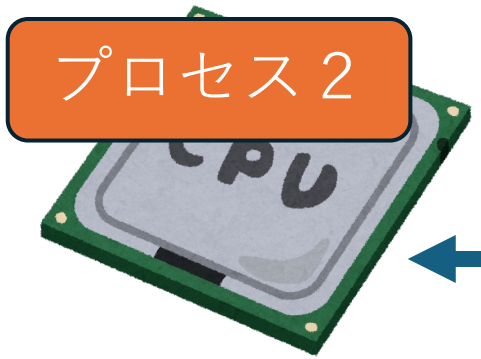
一般的な OS での運用

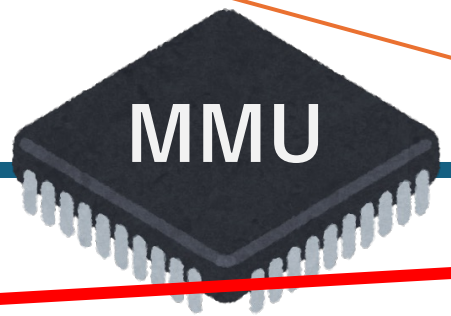
仮想	物理
0x0000	
0x1000	0x3000
…	
…	

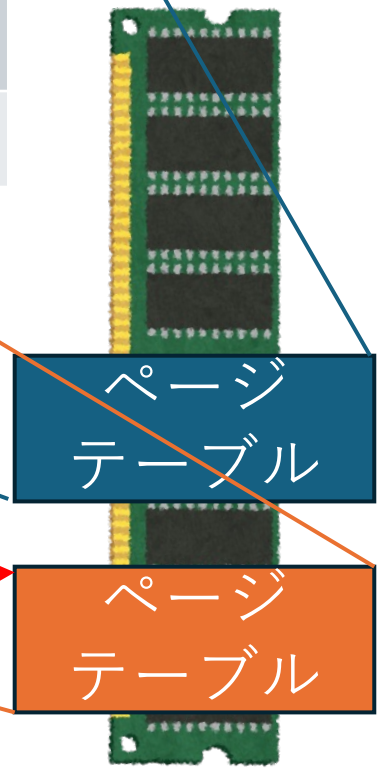
プロセス２用ページテーブル

仮想	物理
0x0000	
0x1000	0x2000
…	
…	

プロセス１用ページテーブル







cr3:

仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

ちなみに、敢えて異なるプロセスが
同じ物理アドレスを参照できるように
設定するのがプロセス間の**共有メモリ**です

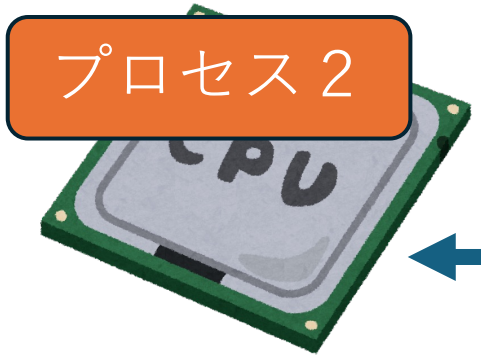
仮想	物理
0x0000	**0x4000**
0x1000	0x3000
…	
…	

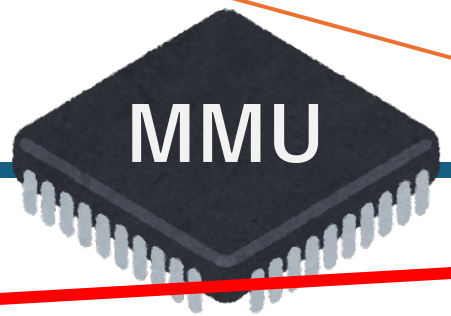
プロセス２用ページテーブル

仮想アドレスはプロセス１とプロセス２で
一致する必要はありません

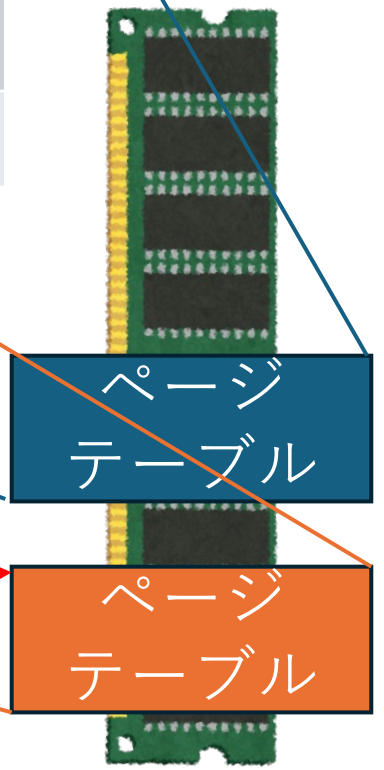
仮想	物理
0x0000	
0x1000	0x2000
0x2000	**0x4000**
…	

プロセス１用ページテーブル





この場合
物理メモリアドレス 0x4000 ~ 0x4fff が
プロセス１とプロセス２で共有されます



cr3:

仮想アドレス 0x1000 へアクセス

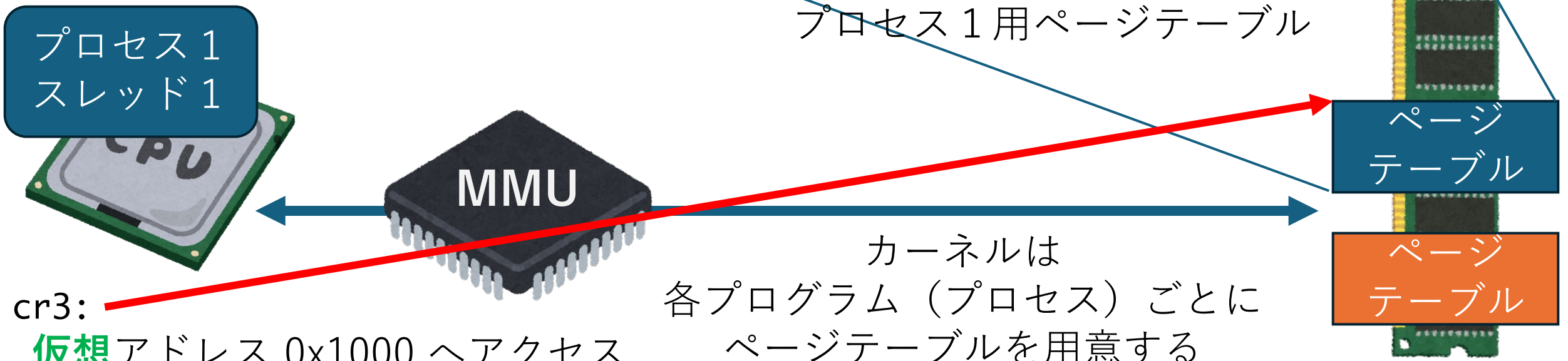
カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

ちなみに、同じプロセスから生成されたスレッドは
実行時に MMU が同じページテーブルを参照するため
スレッド間でメモリ空間が共有されます

仮想	物理
0x0000	
0x1000	0x2000
0x2000	0x4000
…	

プロセス１用ページテーブル



cr3:
仮想アドレス 0x1000 へアクセス

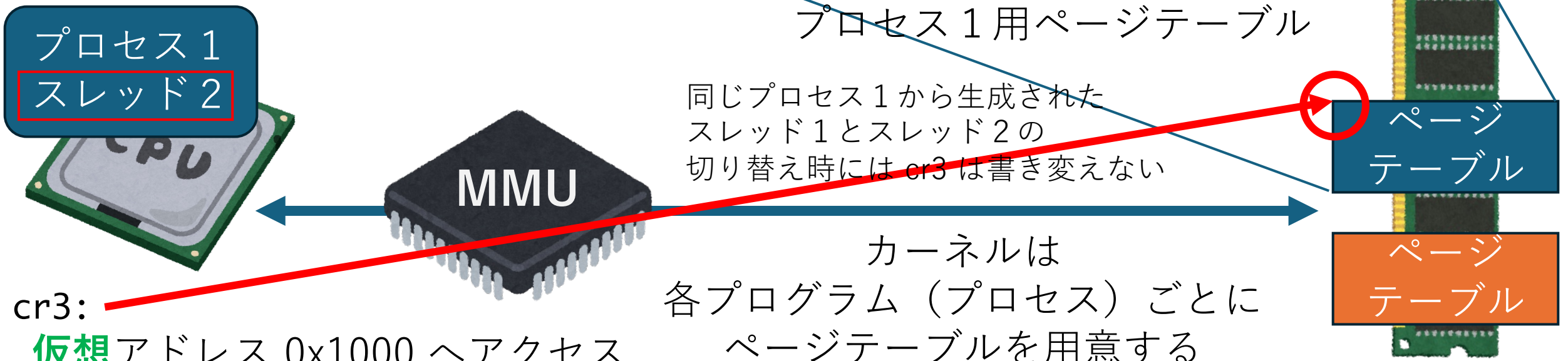
カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

ちなみに、同じプロセスから生成されたスレッドは
実行時に MMU が同じページテーブルを参照するため
スレッド間でメモリ空間が共有されます

仮想	物理
0x0000	
0x1000	0x2000
0x2000	0x4000
…	

プロセス１用ページテーブル



同じプロセス１から生成された
スレッド１とスレッド２の
切り替え時には cr3 は書き変えない

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

cr3:
仮想アドレス 0x1000 へアクセス

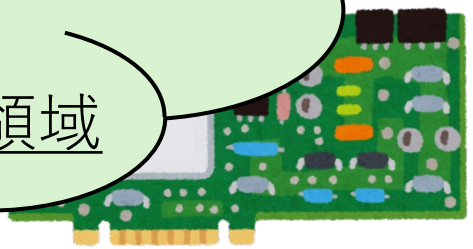
カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスなど

Q. デバイスへの
アクセスとは
具体的に何？
基本的には
I/O の読み書き

Q. ユーザー空間とは何？
A. CPU が非特権モードで
動作している間に
アクセス**可能**なメモリ領域



デバイス

アクセスの可否？

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

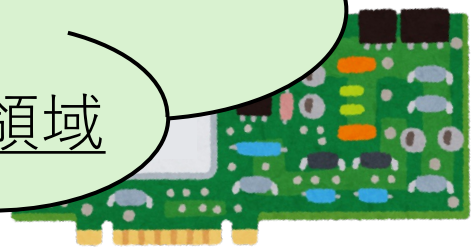
カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスなど

Q. デバイスへの
アクセスとは
具体的に何？
基本的には
I/O の読み書き

Q. ユーザー空間とは何？
A. CPU が非特権モードで
動作している間に
アクセス**可能**なメモリ領域



デバイス

アクセスの可否？

一般的にカーネルが
ページテーブルを通じて
制限を適用している

Q. る？
A. 常では
への
アクセスが許可されていないから

Q. バイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

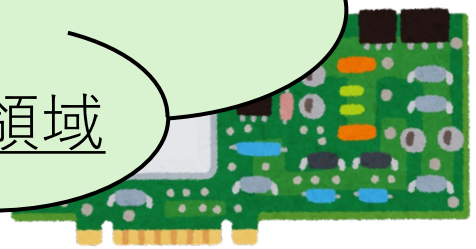
カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスなど

Q. デバイスへの
アクセスとは
具体的に何？
基本的には
I/O の読み書き

Q. ユーザー空間とは何？
A. CPU が非特権モードで
動作している間に
アクセス**可能**なメモリ領域



デバイス

カーネルが
ページテーブルを通じて
アクセスを許可した
物理メモリ領域および
対応する仮想アドレス

アクセスの可否？

一般的にカーネルが
ページテーブルを通じて
制限を適用している

Q. る？
A. 常では
への
アクセスが許可されていないから

Q. デバイスをバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

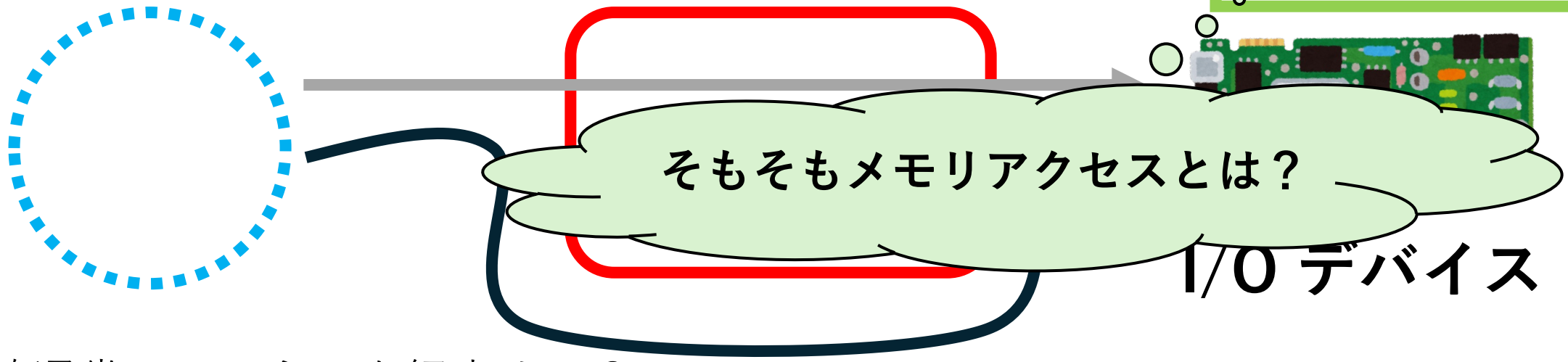
Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き



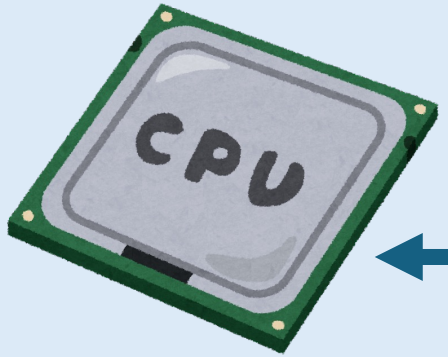
Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

プログラムによるメモリアクセス

0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```



この命令の実行時に CPU から
メモリアクセスが試みられる

```
rax:0x100000000
```

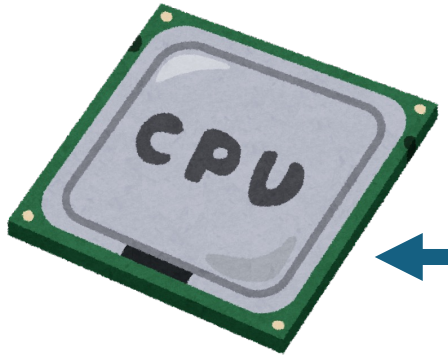
プログラムによるメモリアクセス

<u>ポイント</u>
このような CPU 命令による
メモリの読み書きで
デバイスを操作することができます

0x100000000 へ
0x12345678 を書き込み

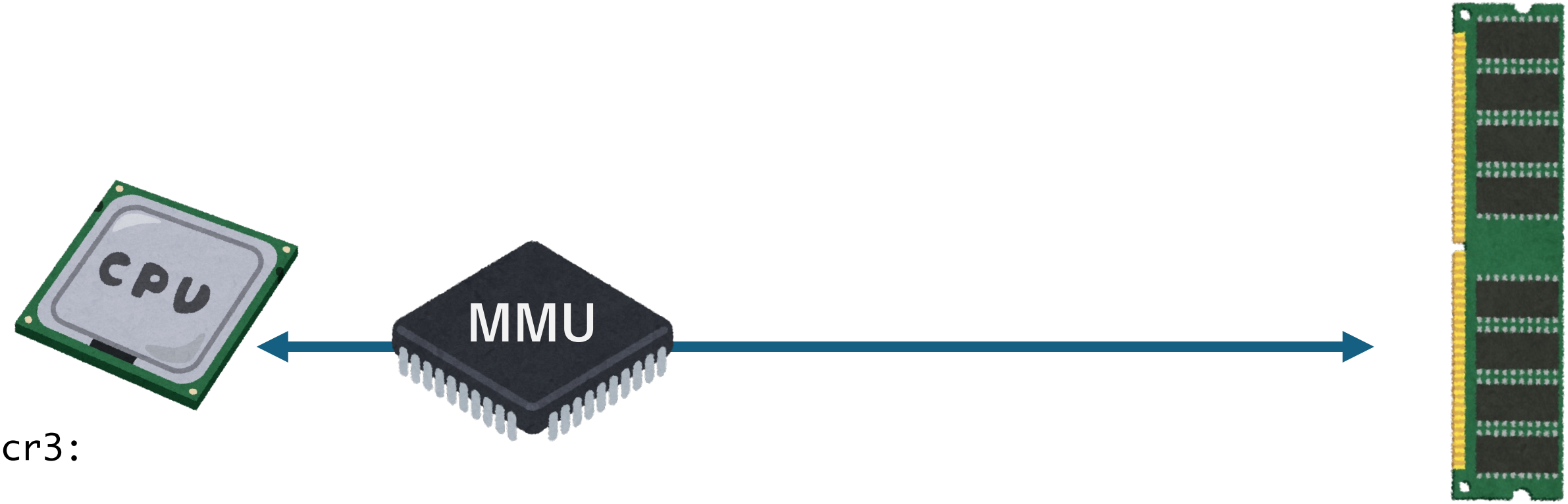
```
movl $0x12345678,(%rax)
```

この命令の実行時に CPU から
メモリアクセスが試みられる



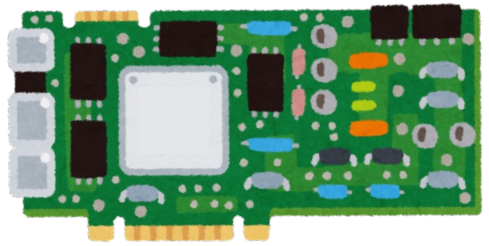

```
rax:0x100000000
```

メモリの読み書きを通じたデバイス操作

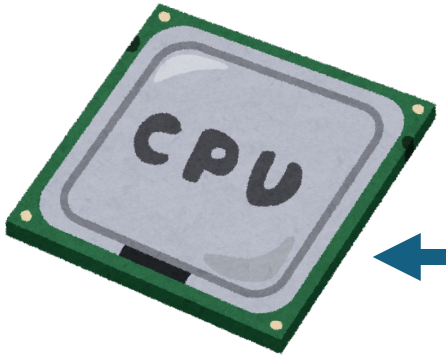


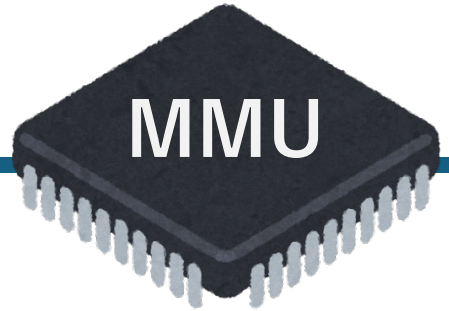
cr3:

メモリの読み書きを通じたデバイス操作



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける

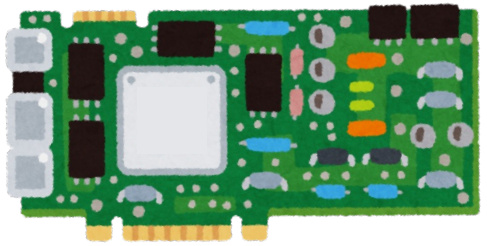




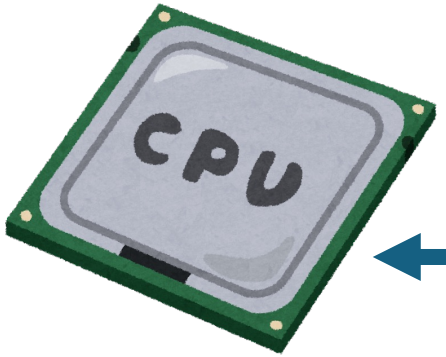
cr3:

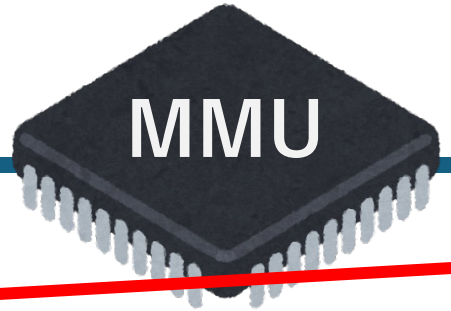
メモリの読み書きを通じたデバイス操作

仮想	物理
0x0000	
0x1000	0x200000
…	
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける



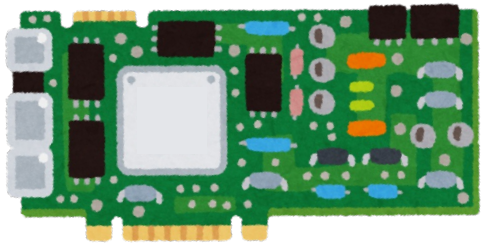


ページ
テーブル

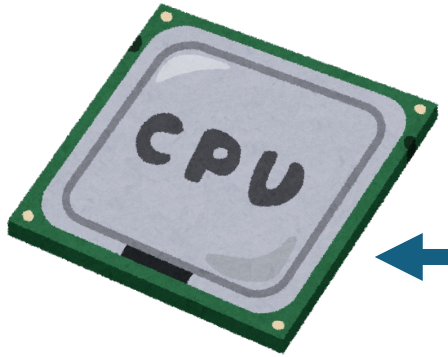
cr3:

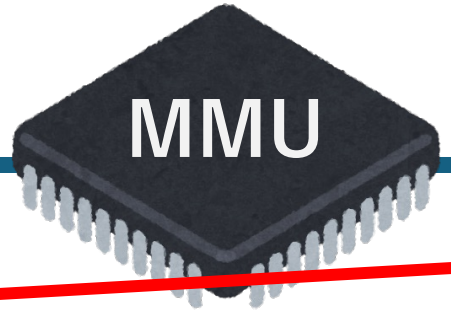
メモリの読み書きを通じたデバイス操作

仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける



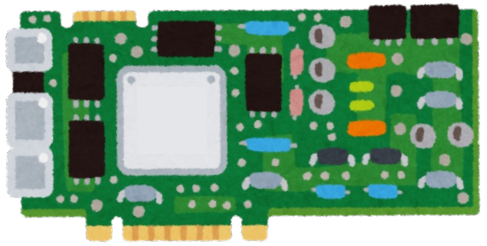


ページ
テーブル

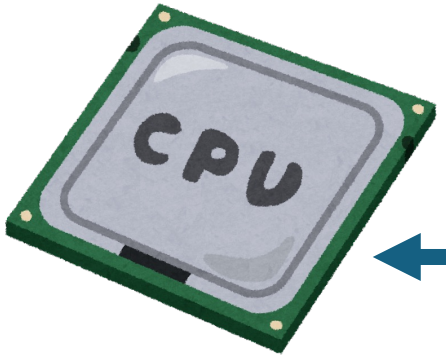
cr3:

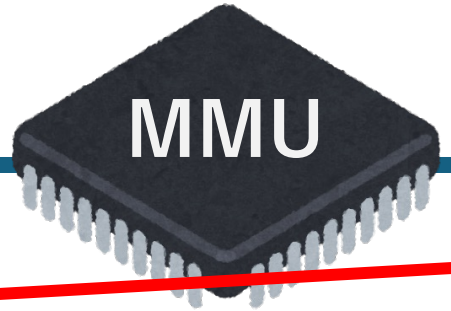
メモリの読み書きを通じたデバイス操作

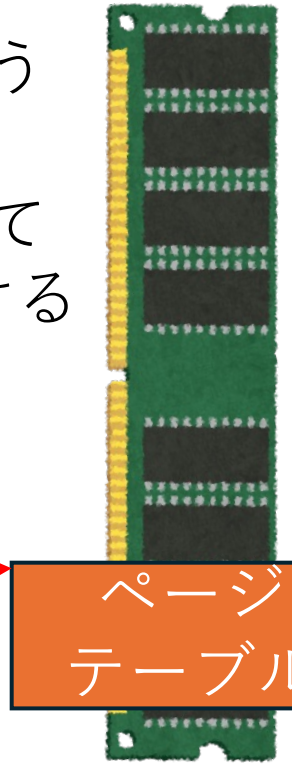
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける







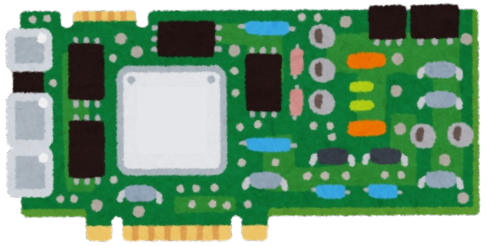
ページ
テーブル

cr3:

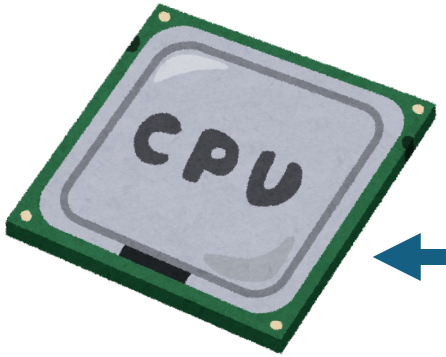
仮想アドレス 0x1000 へアクセス

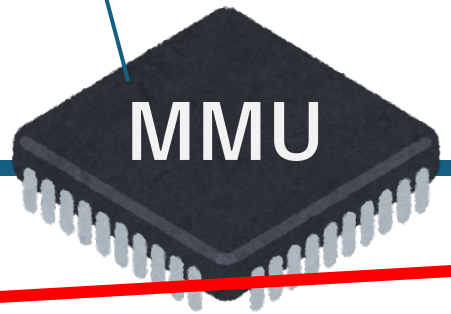
メモリの読み書きを通じたデバイス操作

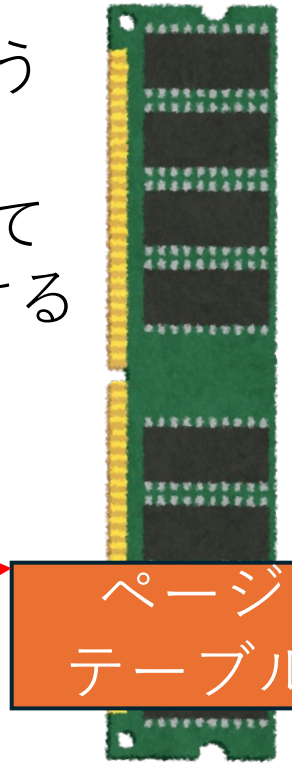
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける







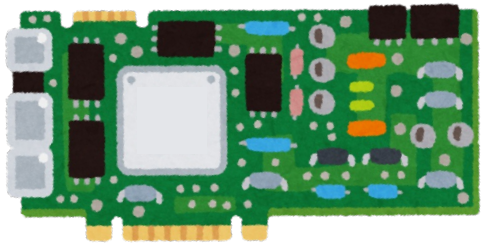
ページ
テーブル

cr3:

仮想アドレス 0x1000 へアクセス

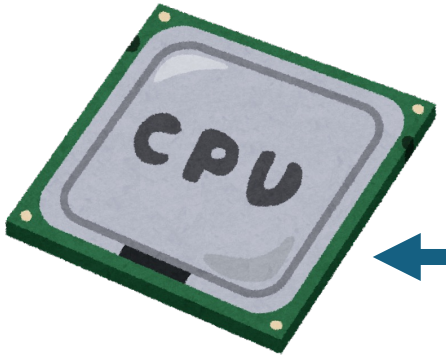
メモリの読み書きを通じたデバイス操作

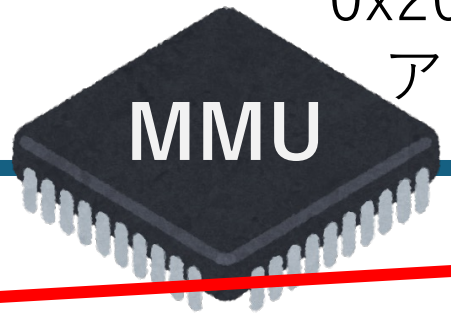
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける

物理アドレス
0x200000 へ
アクセス





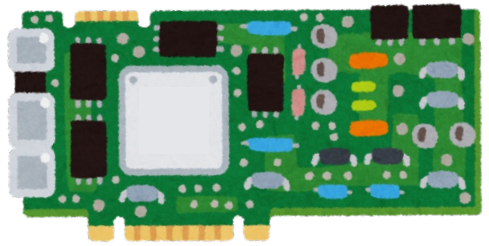
ページ
テーブル

cr3:

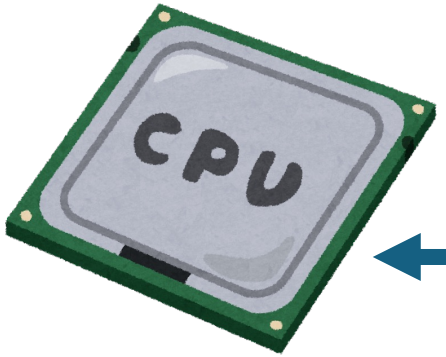
仮想アドレス 0x1000 へアクセス

メモリの読み書きを通じたデバイス操作

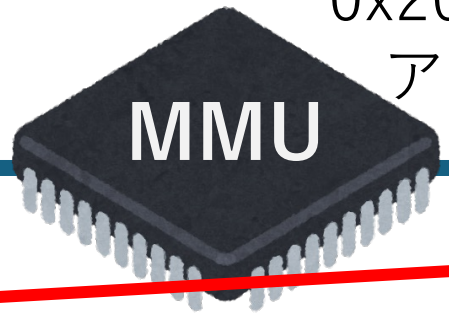
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



0x200000 は
メモリ



物理アドレス
0x200000 へ
アクセス



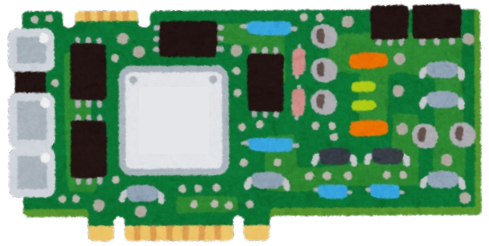
ページ
テーブル

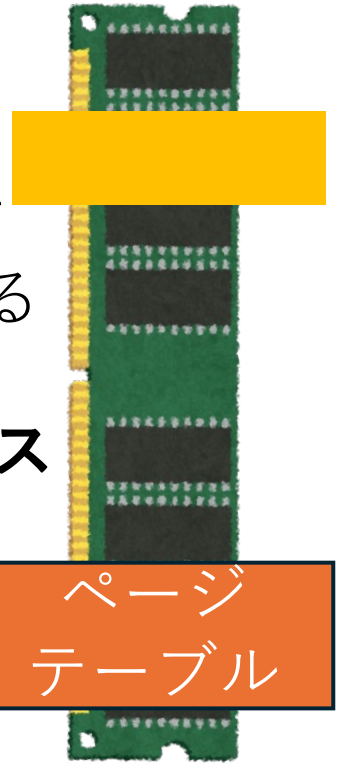
cr3:

仮想アドレス 0x1000 へアクセス

メモリの読み書きを通じたデバイス操作

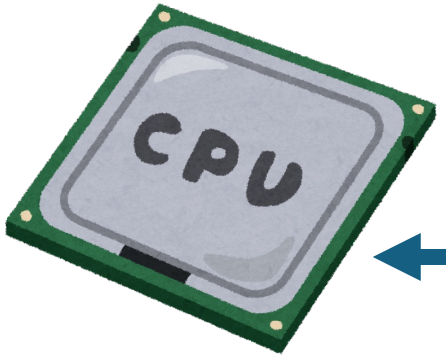
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	

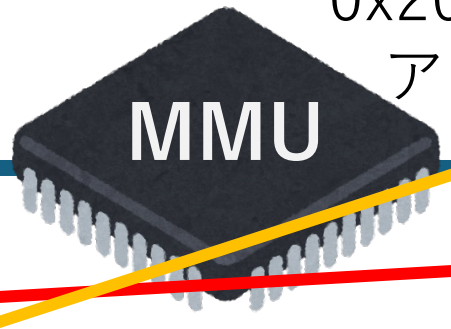




0x200000 は
メモリ

物理アドレス
0x200000 へ
アクセス





メモリへのアクセス

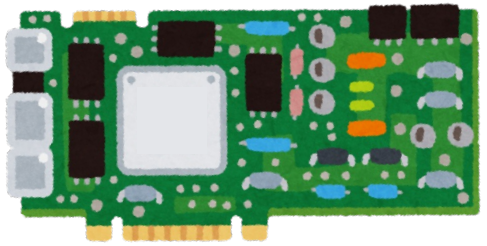
ページ
テーブル

cr3:

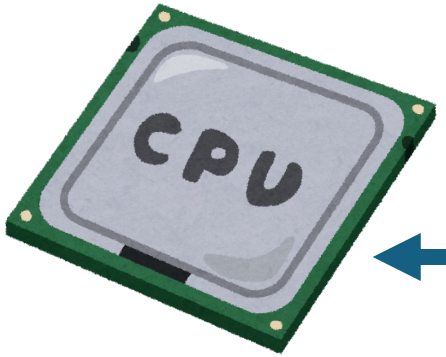
仮想アドレス 0x1000 へアクセス

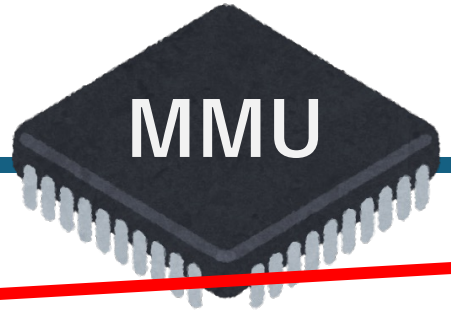
メモリの読み書きを通じたデバイス操作

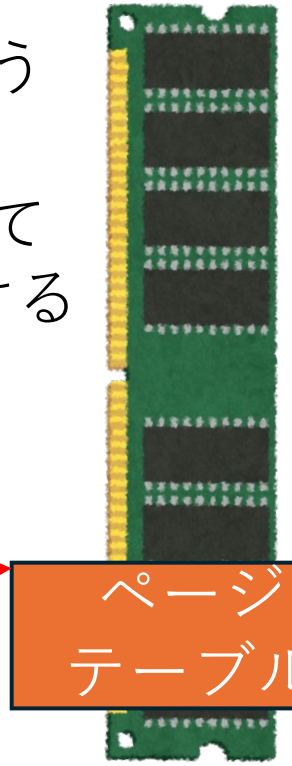
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける







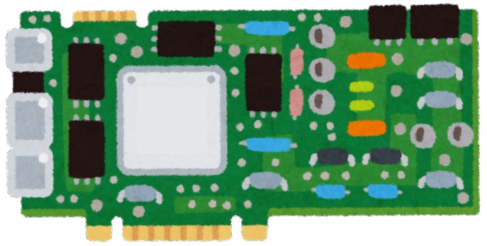
ページ
テーブル

cr3:

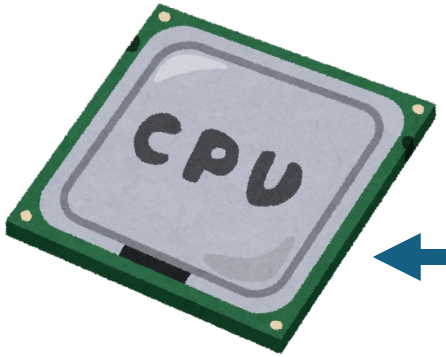
仮想アドレス 0x1000 へアクセス

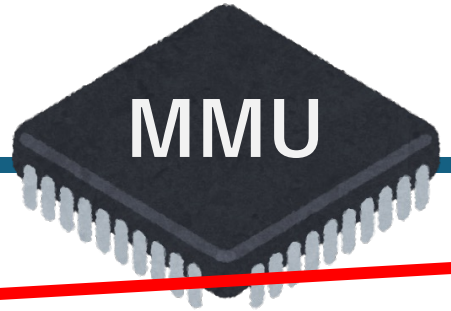
メモリの読み書きを通じたデバイス操作

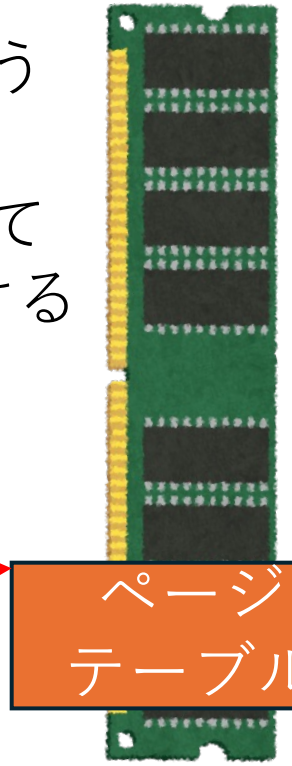
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける







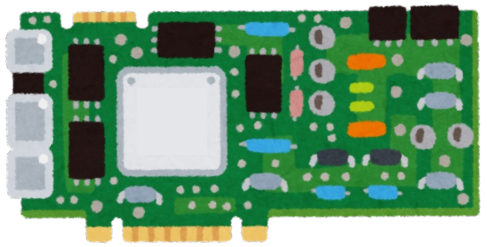
ページ
テーブル

cr3:

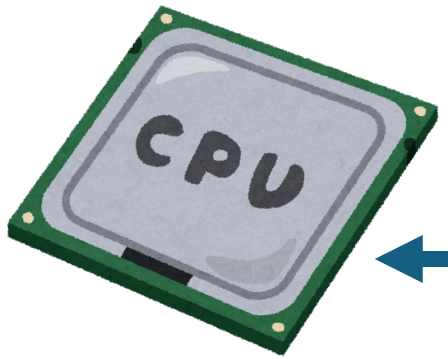
仮想アドレス 0x2000 へアクセス

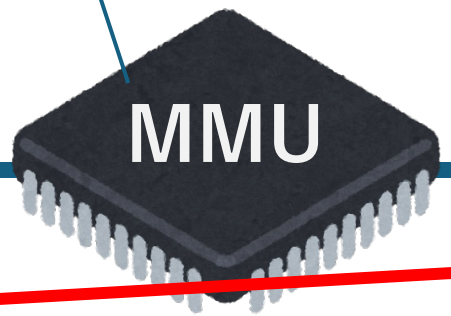
メモリの読み書きを通じたデバイス操作

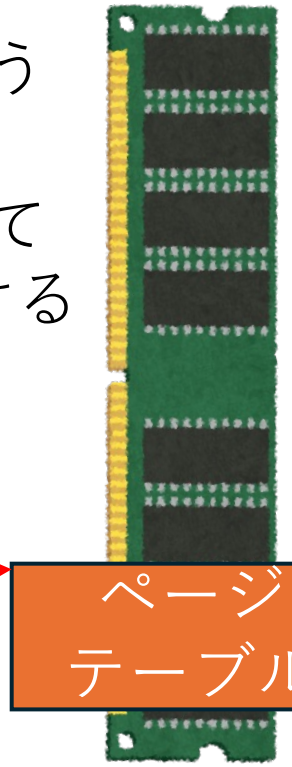
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける







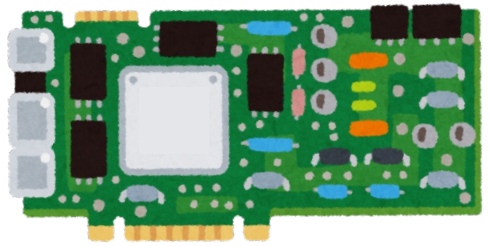
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

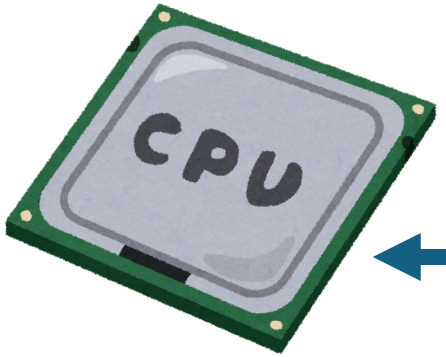
メモリの読み書きを通じたデバイス操作

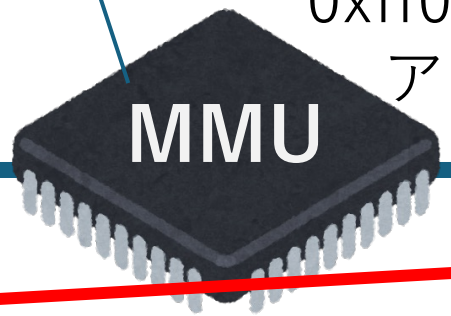
仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	



アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける

物理アドレス
0xff000000 へ
アクセス





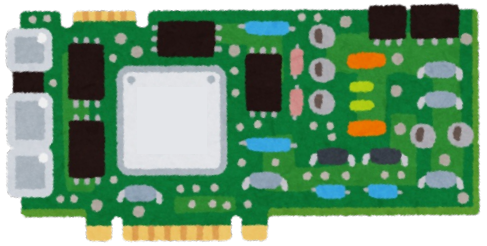
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

メモリの読み書きを通じたデバイス操作

仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	

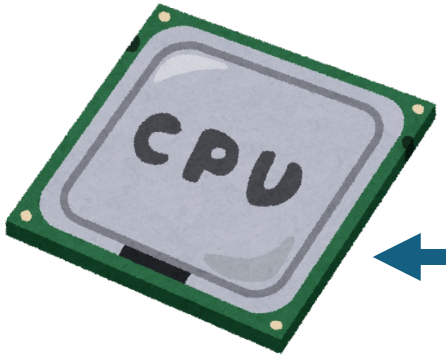


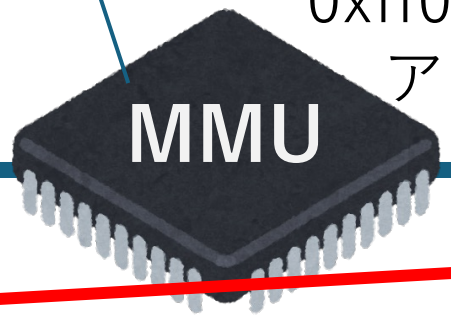
アドレスデコーダという

0xff000000 は
NIC

信号の送出先を振り分ける

物理アドレス
0xff000000 へ
アクセス





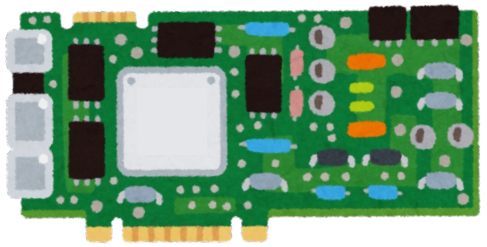
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

メモリの読み書きを通じたデバイス操作

仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	

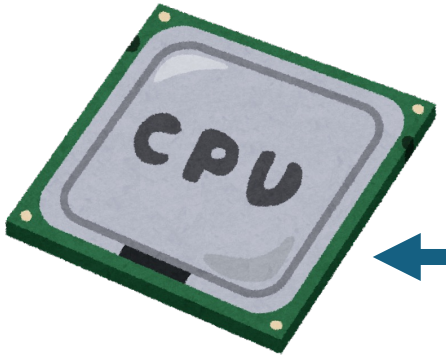


NIC へのアクセス

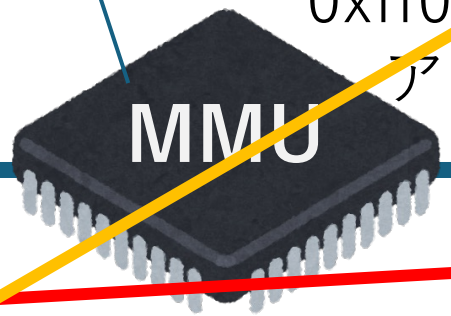
アドレスデコーダという

0xff000000 は
NIC

信号の出力先を振り分ける



物理アドレス
0xff000000 へ
アクセス



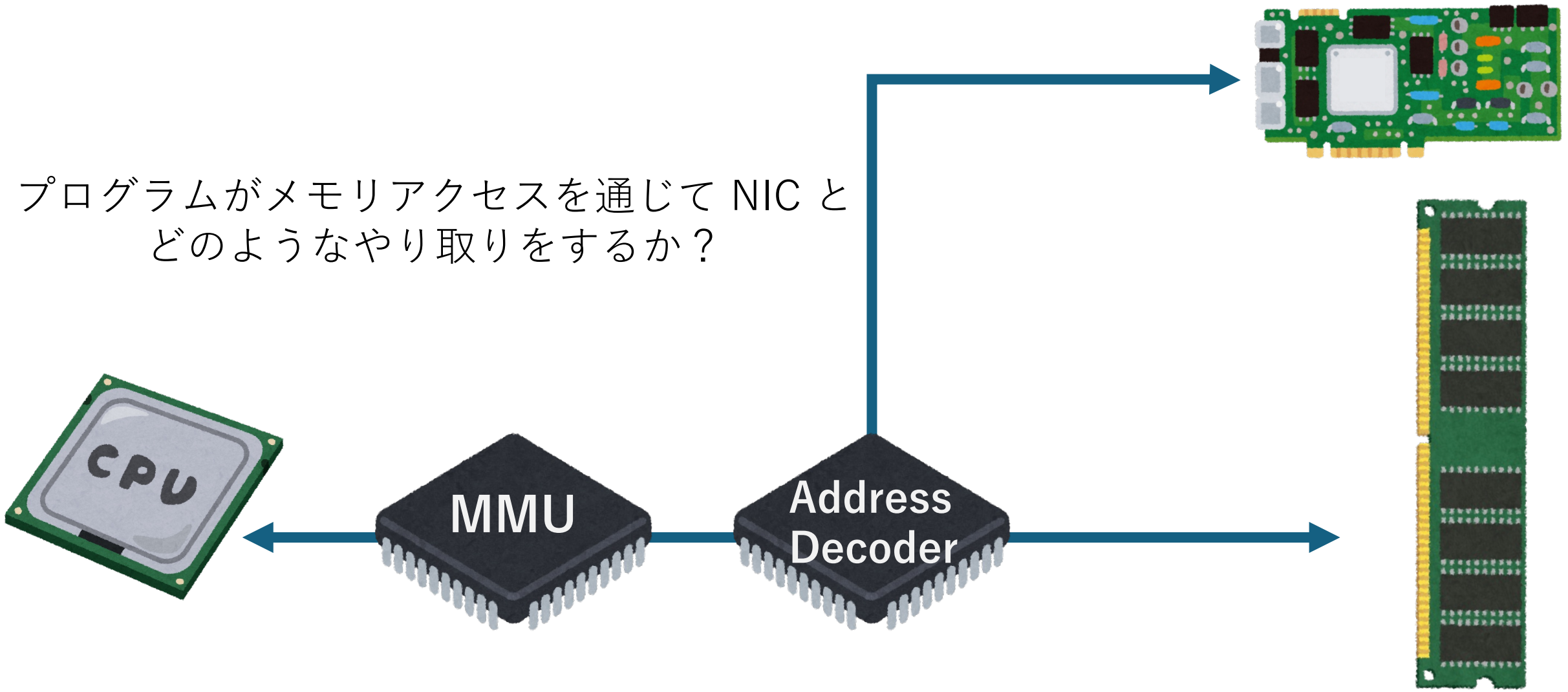
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

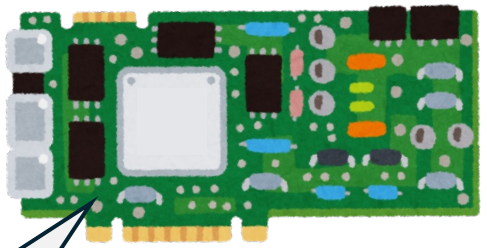
メモリの読み書きを通じたデバイス操作

プログラムがメモリアクセスを通じて NIC とどのようなやり取りをするか？



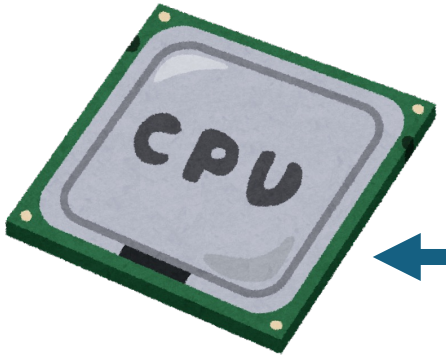
プログラムによる NIC の取り扱い

プログラムは NIC の送受信キューを通して
パケットの送受信を行う



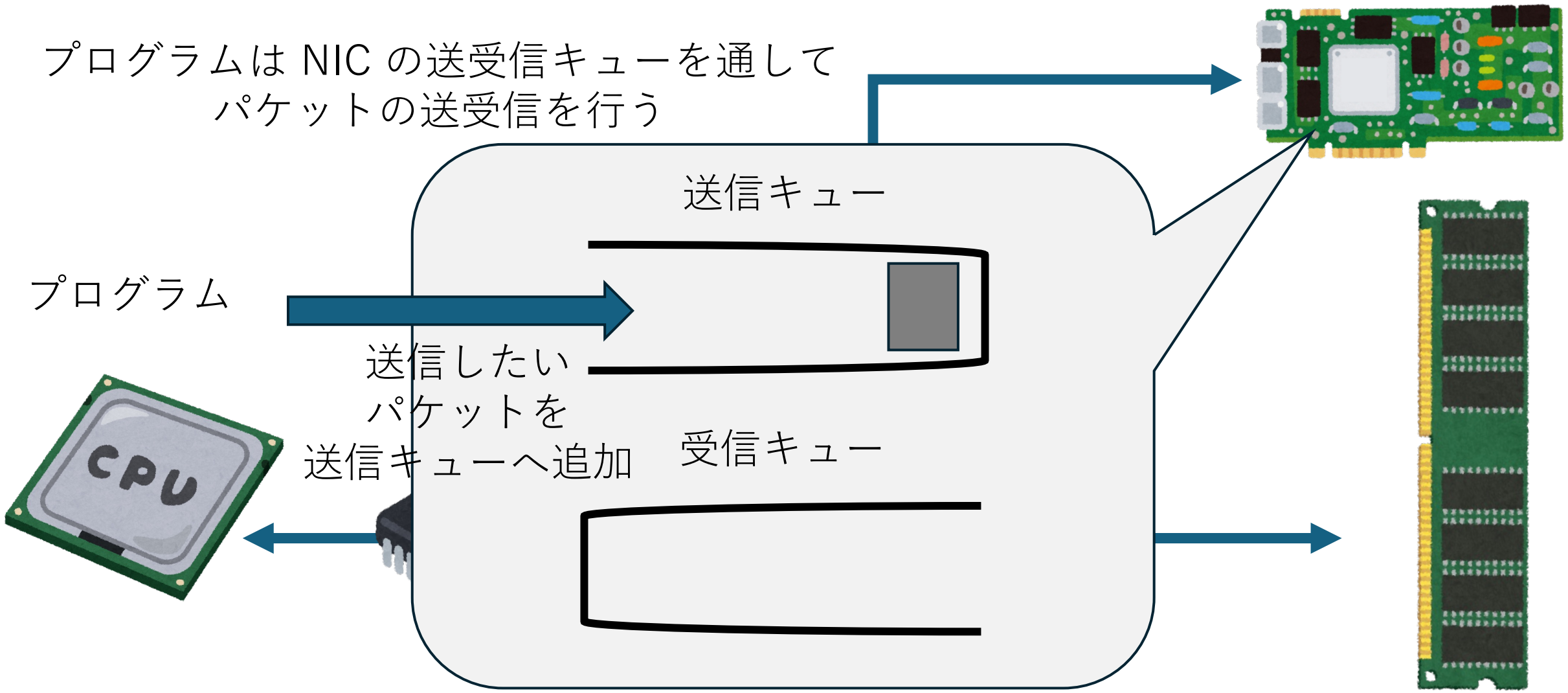
送信キュー

プログラム

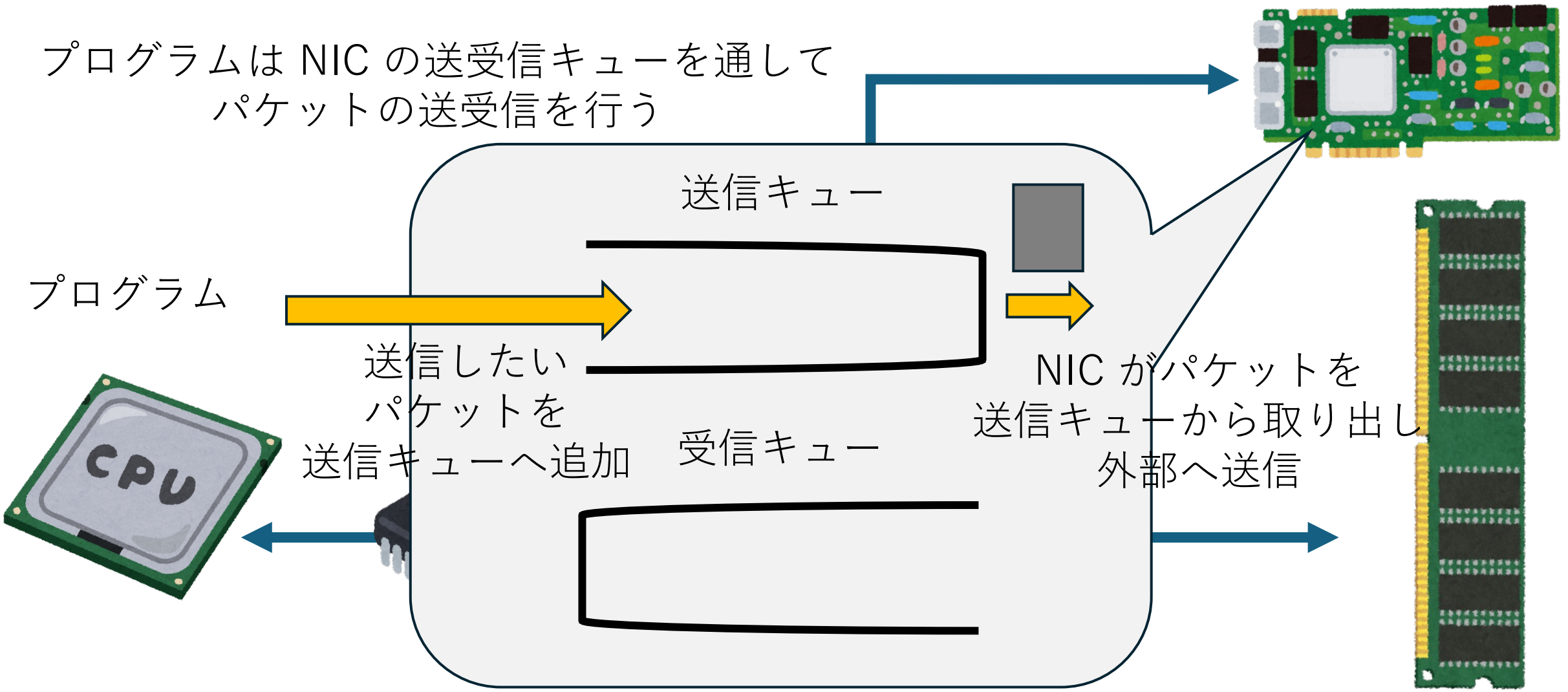


受信キュー

プログラムによる NIC の取り扱い

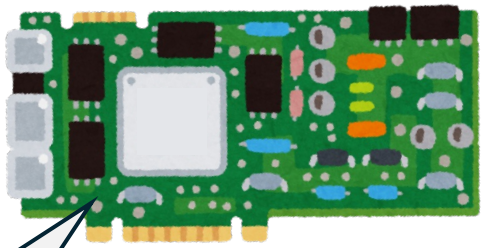


プログラムによる NIC の取り扱い



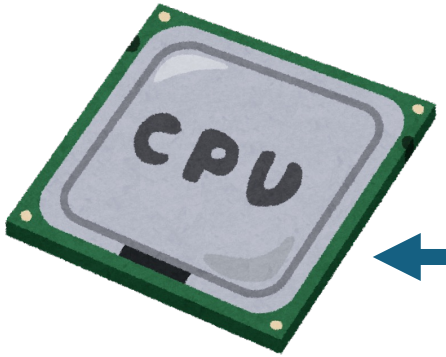
プログラムによる NIC の取り扱い

プログラムは NIC の送受信キューを通して
パケットの送受信を行う



送信キュー

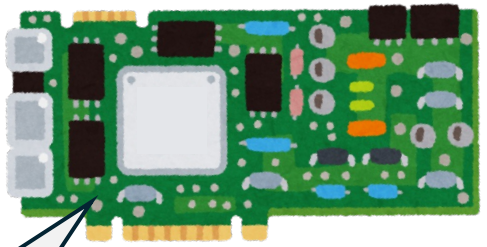
プログラム



受信キュー

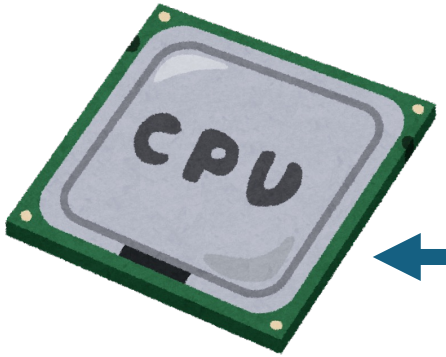
プログラムによる NIC の取り扱い

プログラムは NIC の送受信キューを通して
パケットの送受信を行う



送信キュー

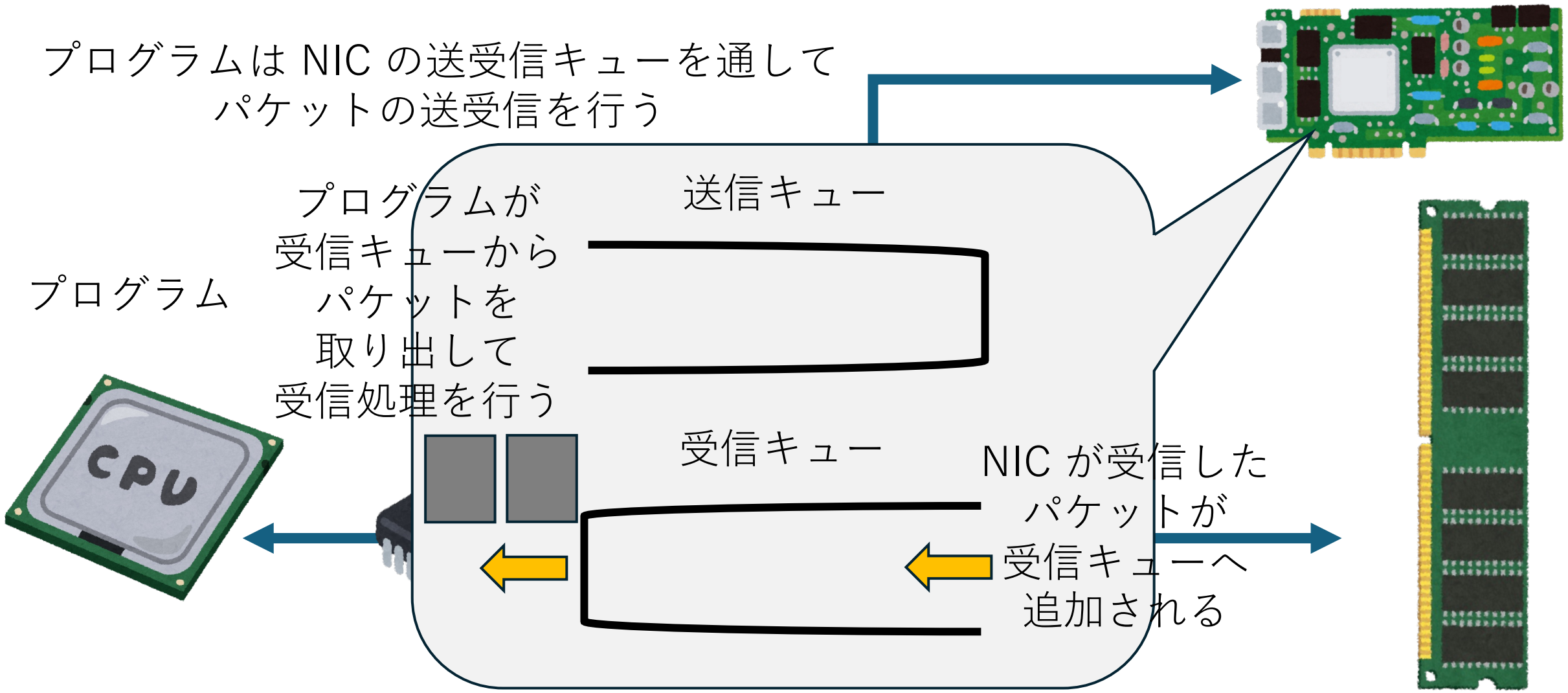
プログラム



受信キュー

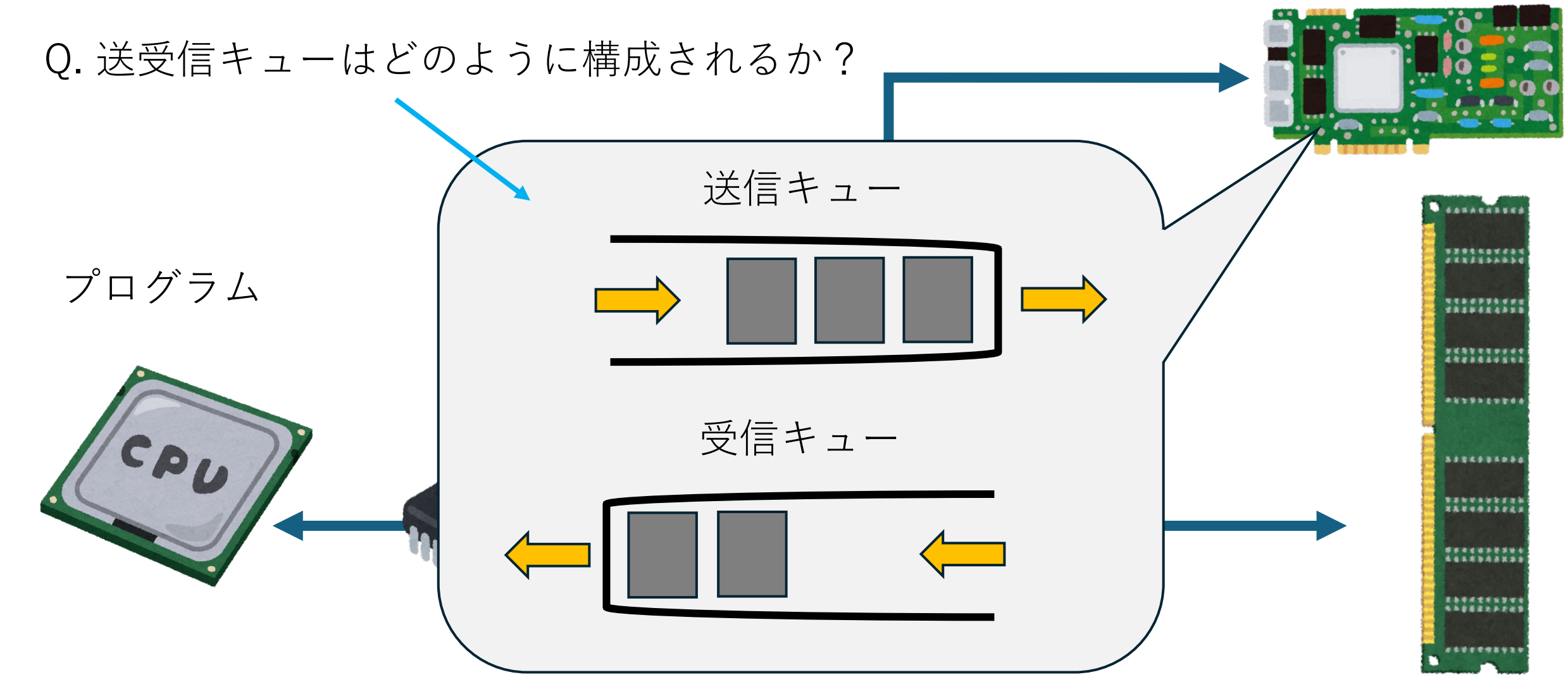
NIC が受信した
パケットが
受信キューへ
追加される

プログラムによる NIC の取り扱い



プログラムによる NIC の取り扱い

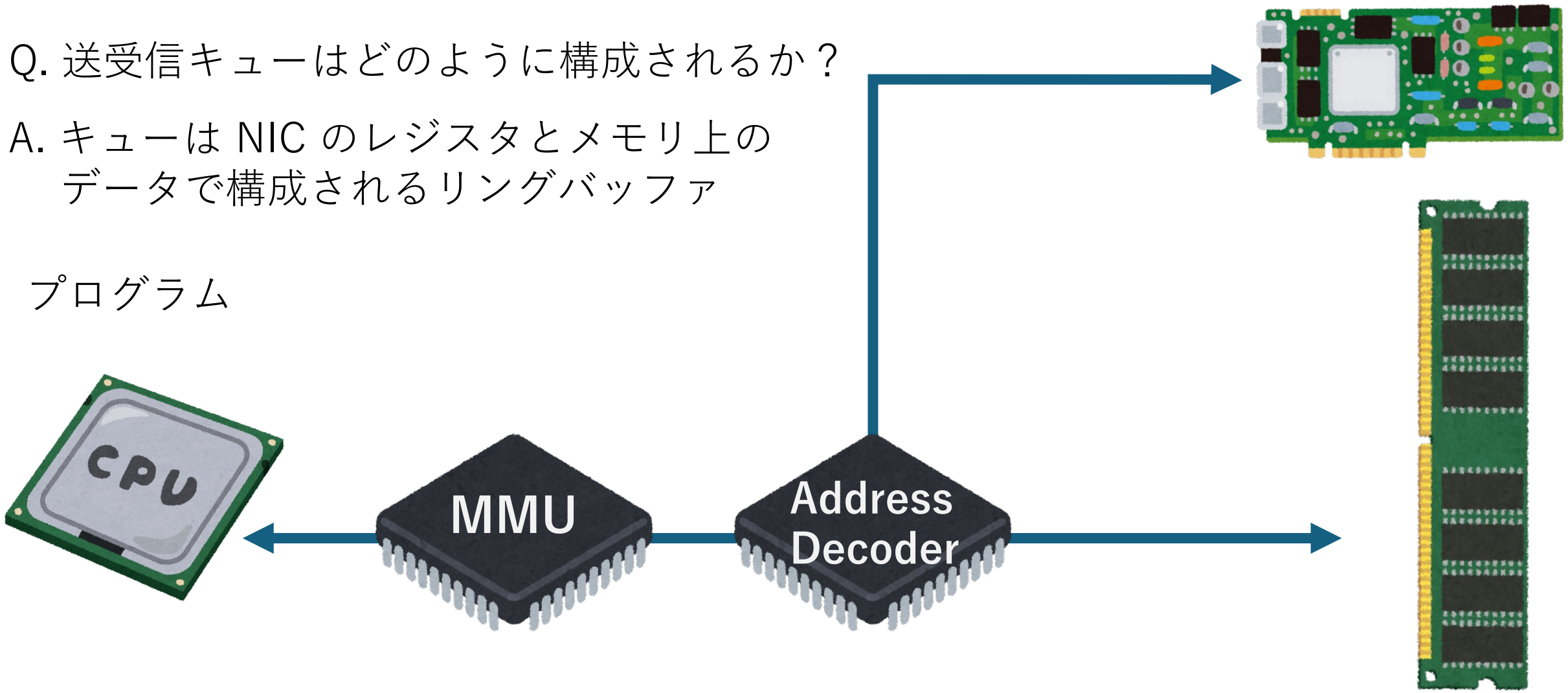
Q. 送受信キューはどのように構成されるか？



プログラムによる NIC の取り扱い

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



プログラムによる NIC の取り扱い

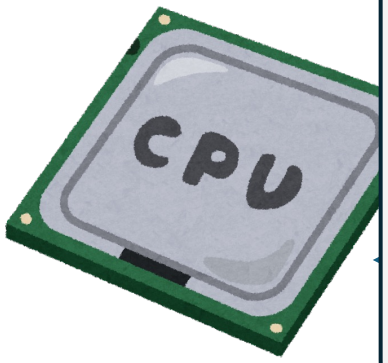
Q. 送受信キューはどのように構成されるか？

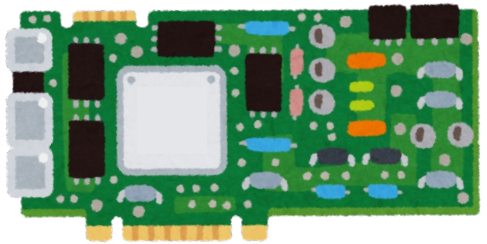
A. キューは NIC のレジスタとメモリ上のデータで構成されるリングバッファ

プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

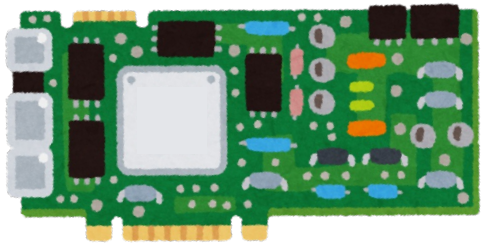




プログラムによる NIC の取り扱い

Q. 送受信キューはどのように構成されるか？

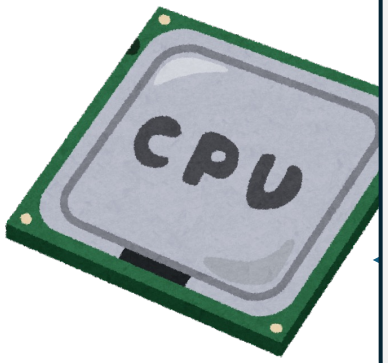
A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



デスクリプタリングとも呼ばれます

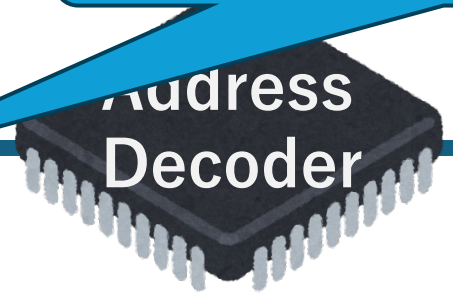
デスクリプタとも呼ばれます

プログラム



簡単な実装

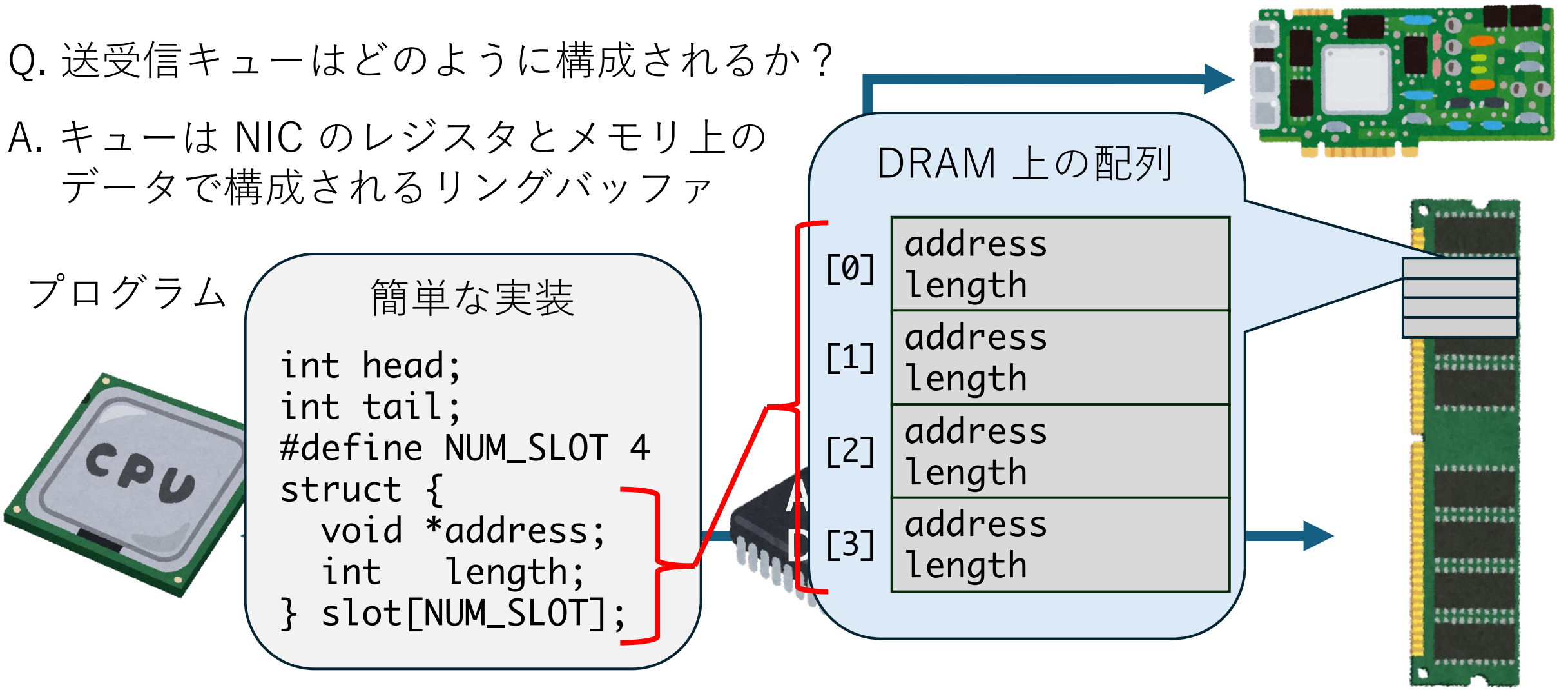
```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```



プログラムによる NIC の取り扱い

Q. 送受信キューはどのように構成されるか？

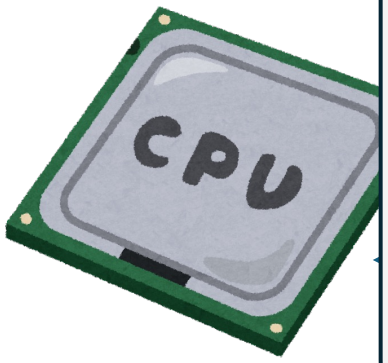
A. キューは NIC のレジスタとメモリ上のデータで構成されるリングバッファ



プログラムによる NIC の

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



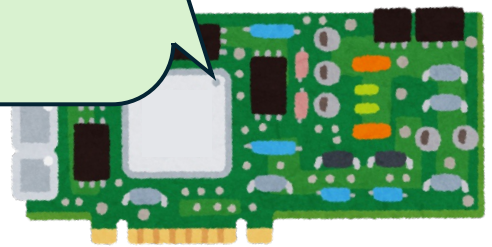
プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

NIC のレジスタ

```
ring_head
ring_tail
ring_address
ring_size
```



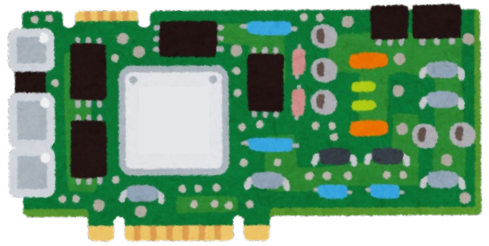
DRAM 上の配列

[0]	address length
[1]	address length
[2]	address length
[3]	address length

プログラムによる NIC の取り扱い

Q. 送受信キューはどのように構成されるか？

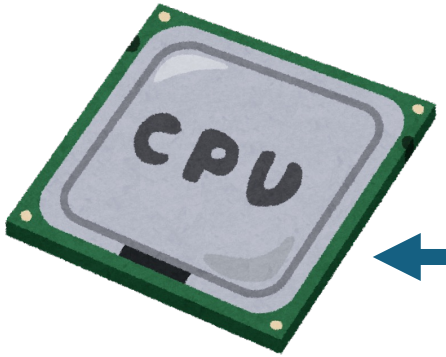
A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

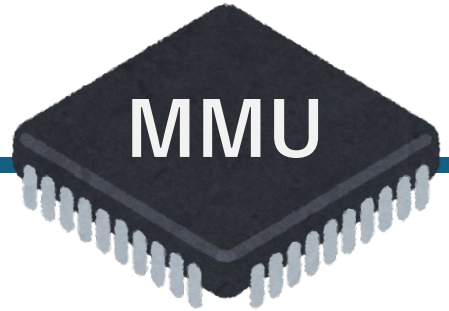


プログラム

**プログラムから
NIC のレジスタと DRAM は
MMU / Address Decoder を
通じてアクセス可能**

アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける



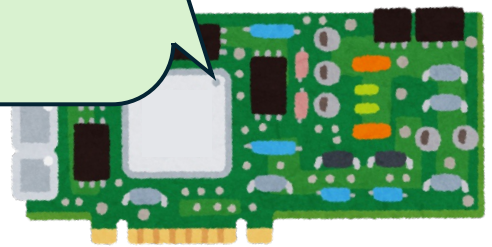


（どのアドレスが NIC のどのレジスタに対応するかは NIC の仕様に依存）

これらレジスタは NIC が扱うことのできる
キュー（リングバッファ）の数だけあります

NIC のレジスタ

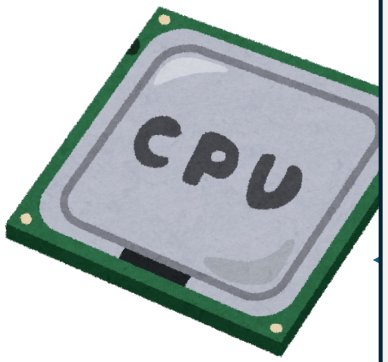
```
ring_head
ring_tail
ring_address
ring_size
```



Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

プログラム



簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

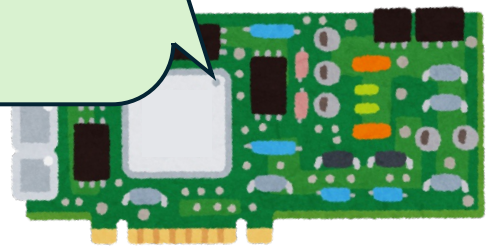
DRAM 上の配列

[0]	address length
[1]	address length
[2]	address length
[3]	address length

これらレジスタは NIC が扱うことのできる
キュー（リングバッファ）の数だけあります

NIC のレジスタ

```
ring_head
ring_tail
ring_address
ring_size
```

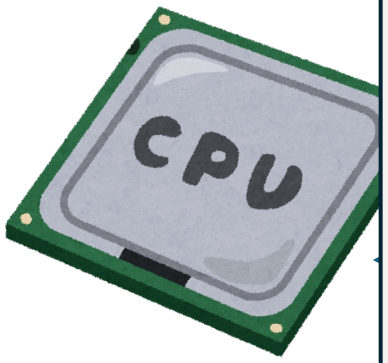


Q. 送受信キューはどのように構成されるか？

A.

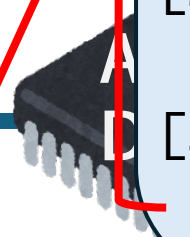
キューは基本的に
送信キューもしくは受信キューです

プログラム



簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```



DRAM 上の配列

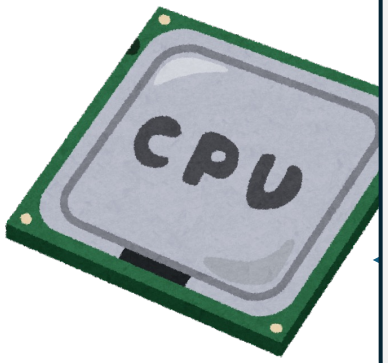
[0]	address length
[1]	address length
[2]	address length
[3]	address length

プログラムによる NIC の

プログラムによる初期設定

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



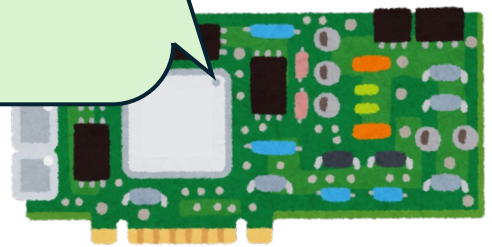
プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

NIC のレジスタ

```
ring_head
ring_tail
ring_address
ring_size
```



DRAM 上の配列

[0]	address length
[1]	address length
[2]	address length
[3]	address length

プログラムによる NIC の

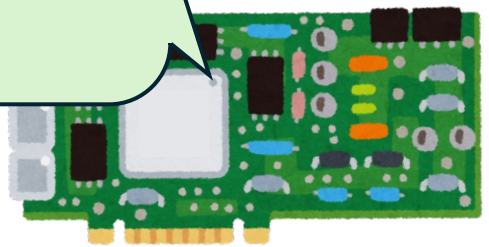
プログラムによる初期設定

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

NIC のレジスタ
ring_head
ring_tail
ring_address
ring_size: 4

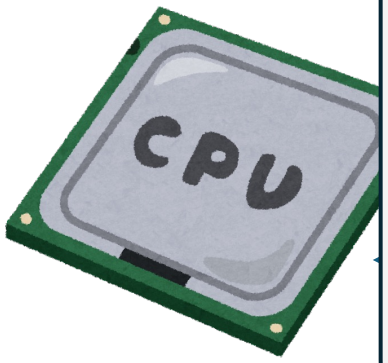
リングサイズを設定



DRAM 上の配列

[0]	address length
[1]	address length
[2]	address length
[3]	address length

プログラム



簡単な実装

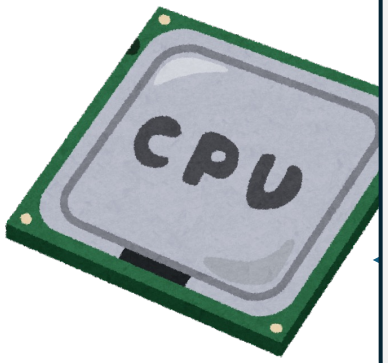
```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

プログラムによる NIC の

プログラムによる初期設定

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



プログラム

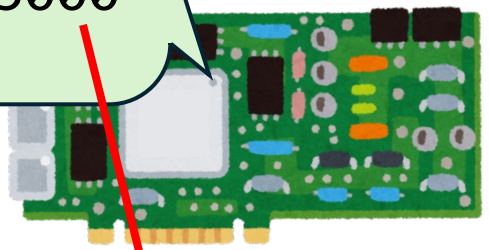
簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

NIC のレジスタ

```
ring_head
ring_tail
ring_address:0x5000
ring_size: 4
```

DRAM 上の
配列の先頭の
物理アドレスを
登録



DRAM 上の配列

[0]	address length
[1]	address length
[2]	address length
[3]	address length

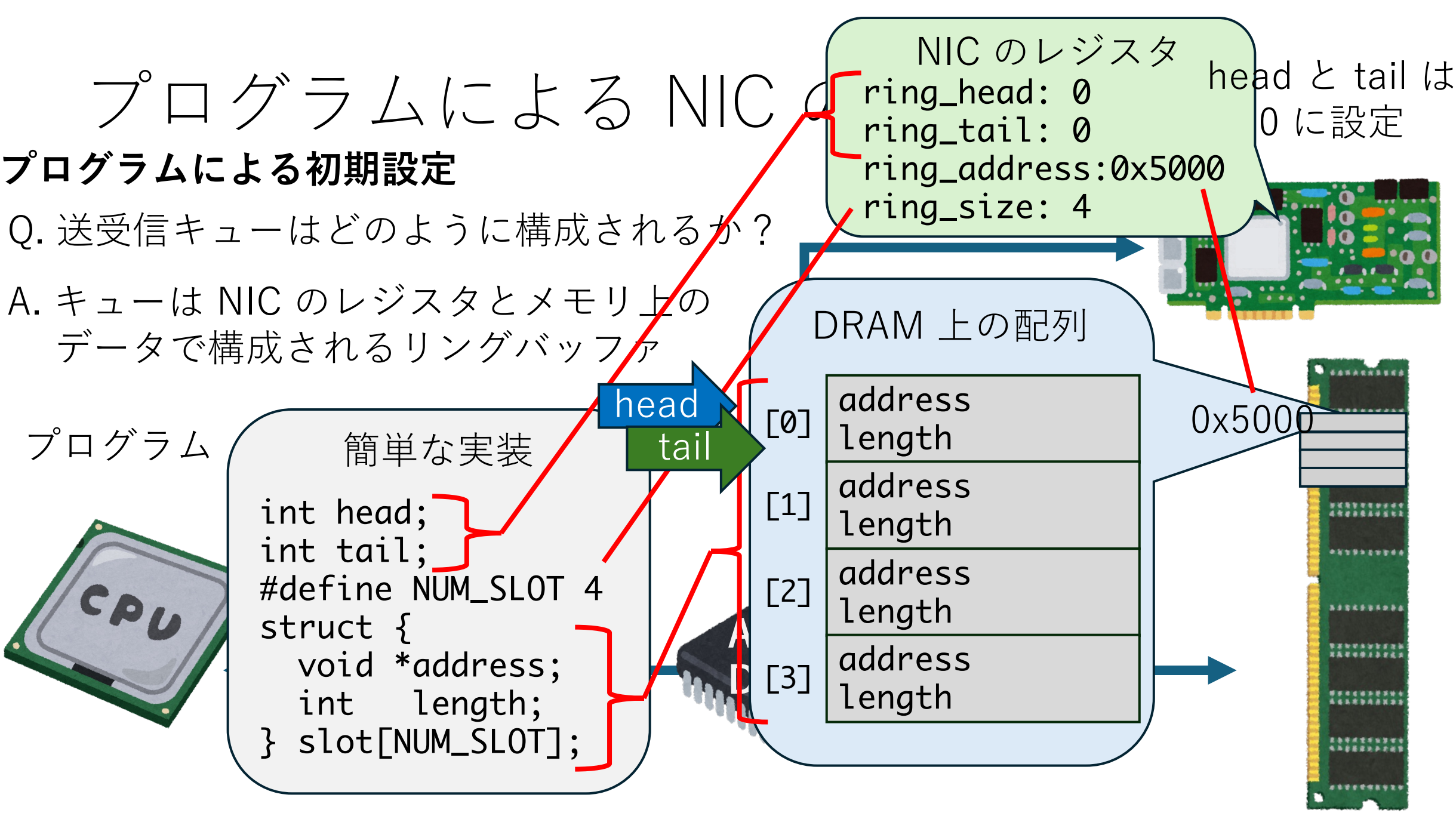
0x5000

プログラムによる NIC の

プログラムによる初期設定

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

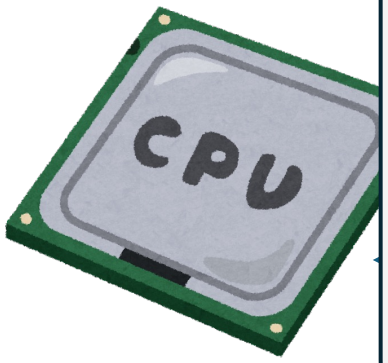


プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



プログラム

簡単な実装

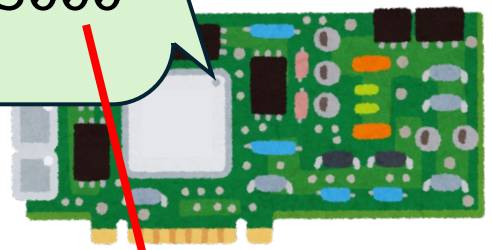
```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

head

tail

NIC のレジスタ

```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address length
[1]	address length
[2]	address length
[3]	address length

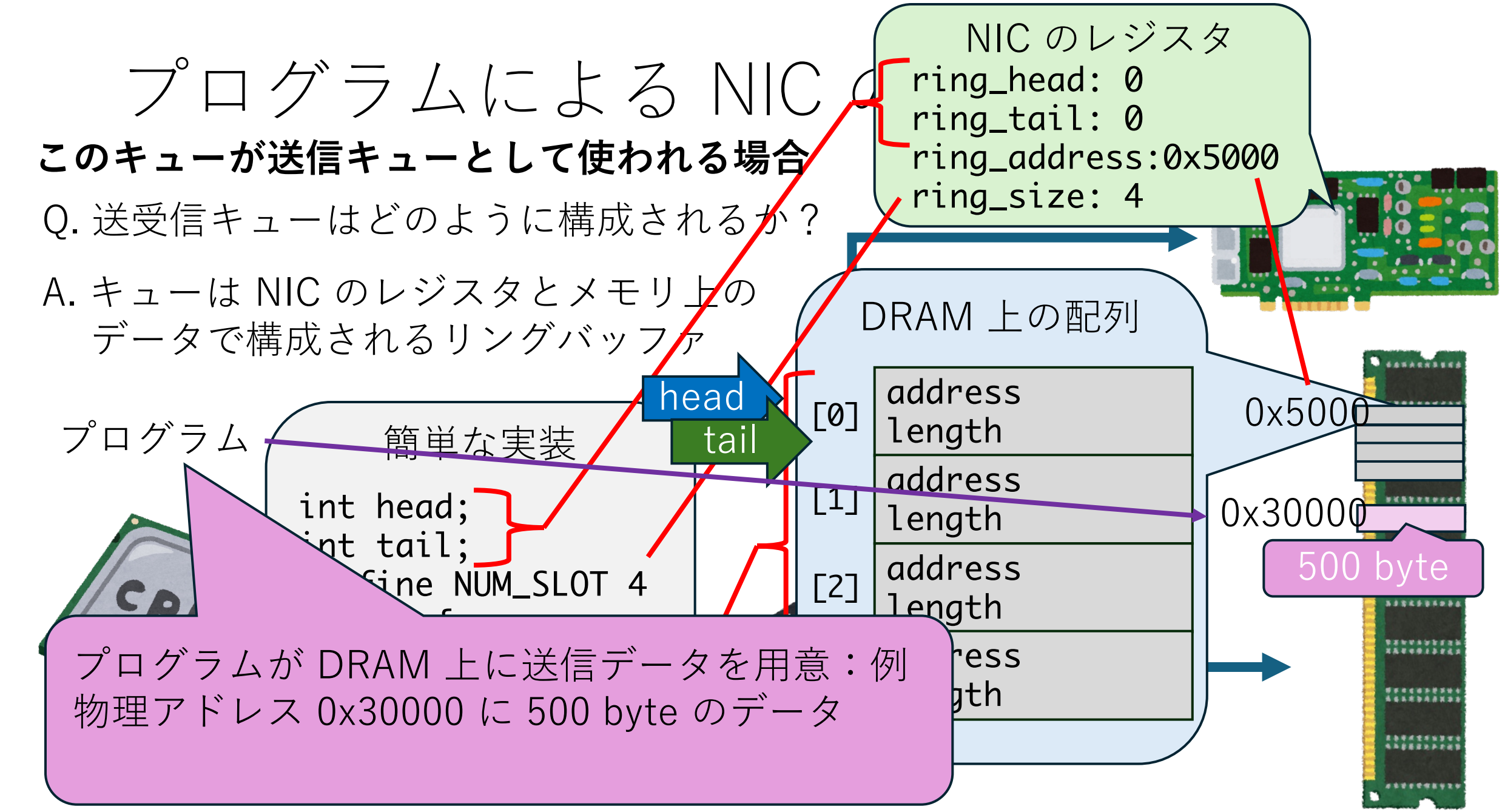
0x5000

プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

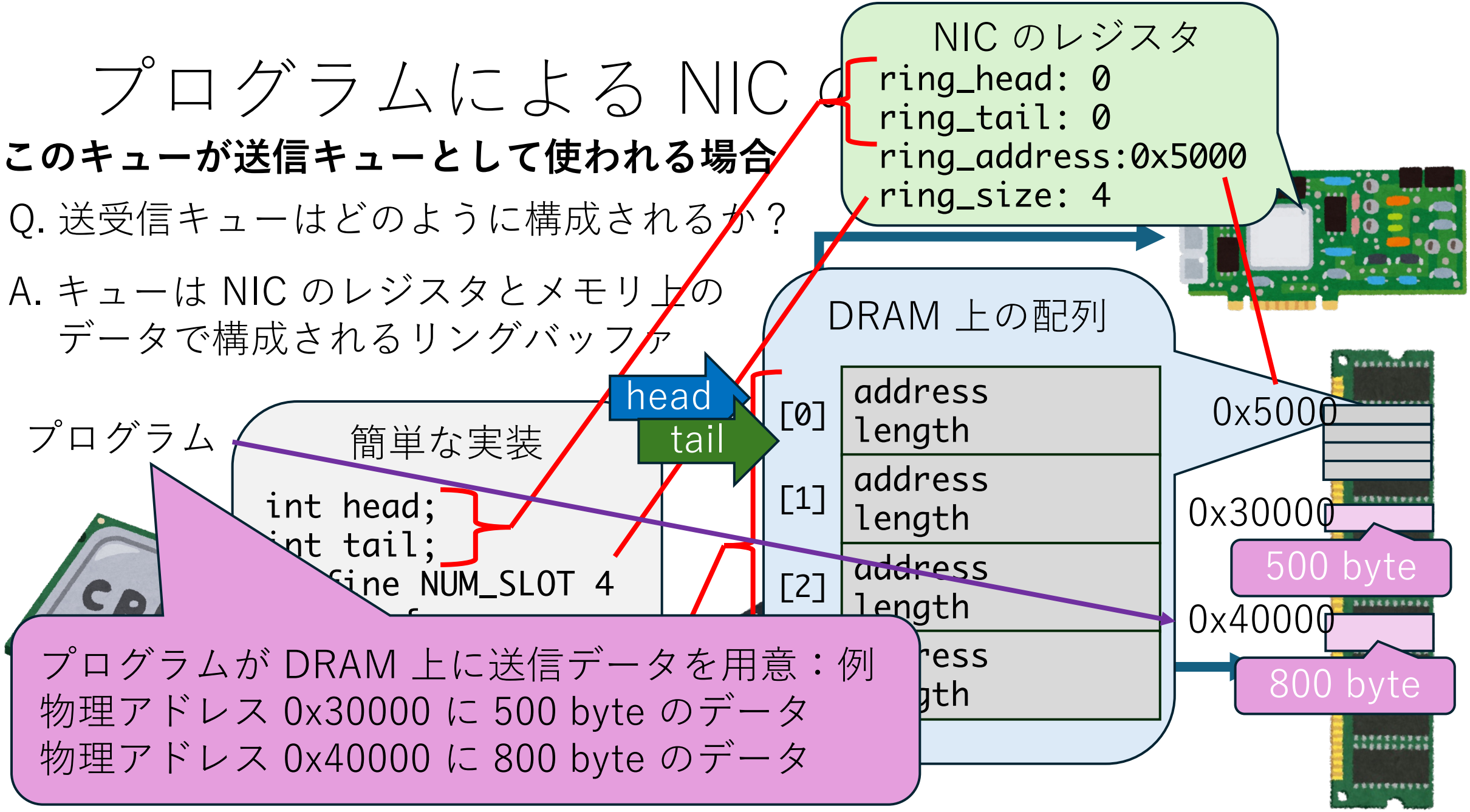


プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

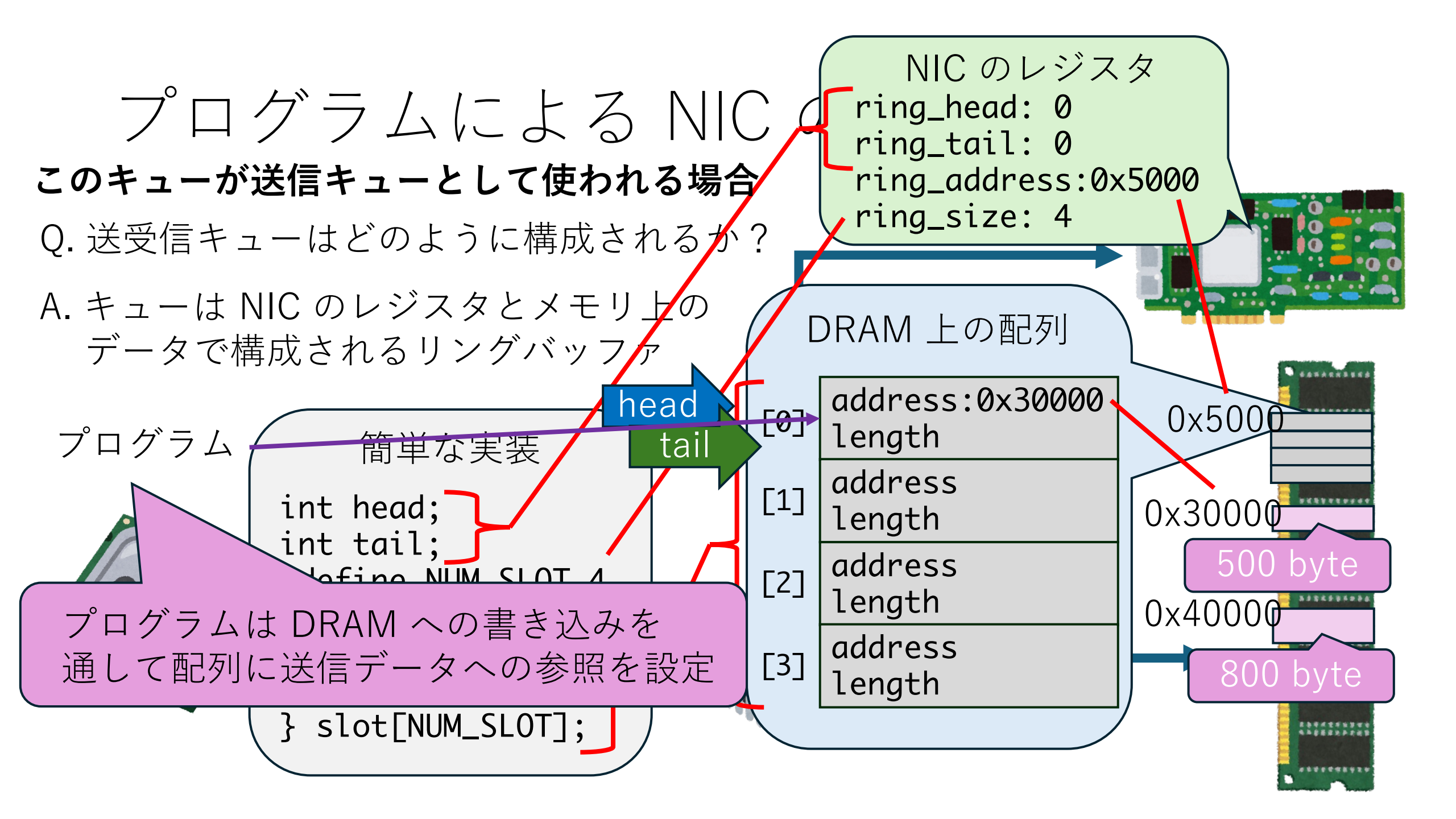


プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

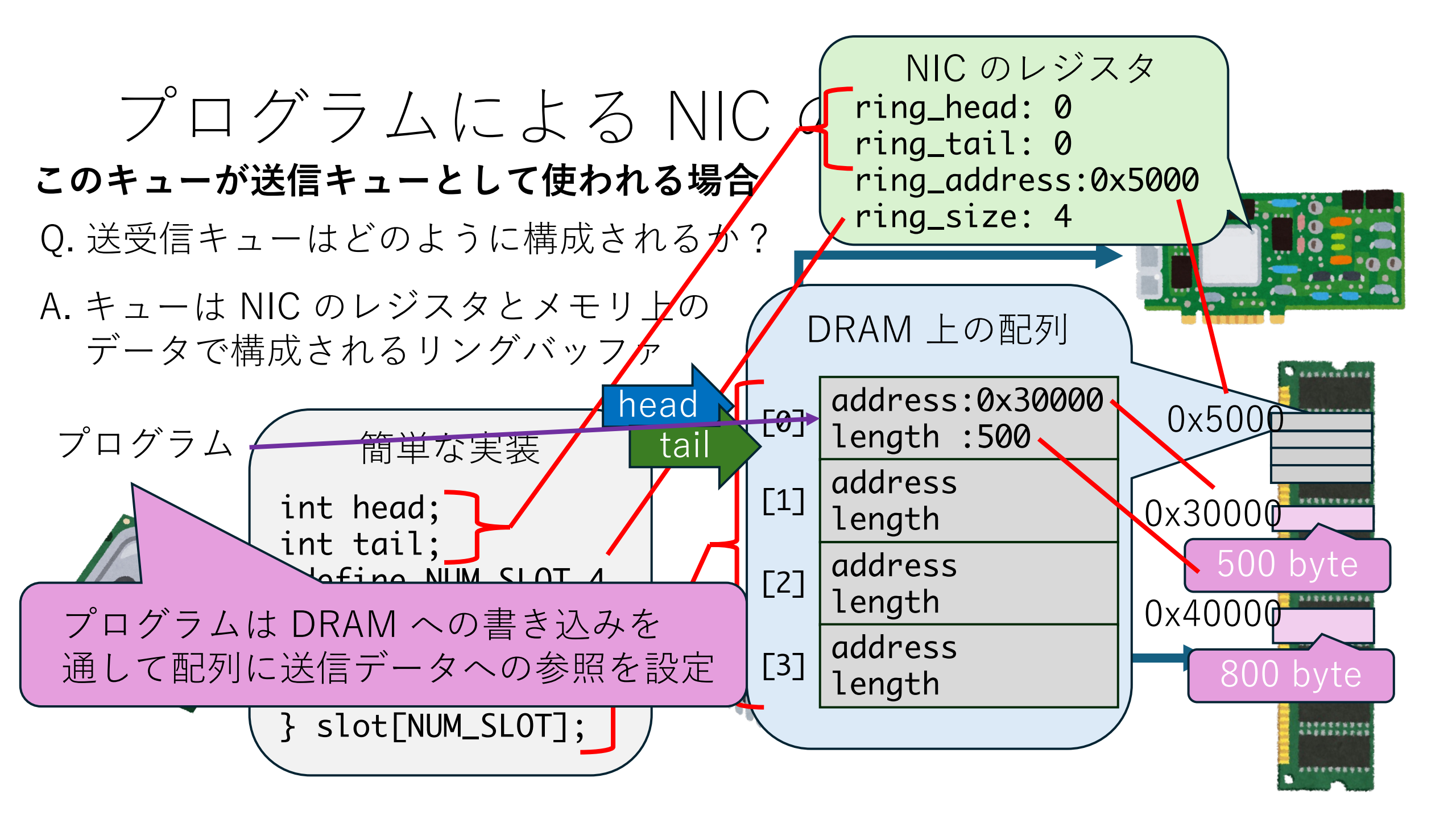


プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

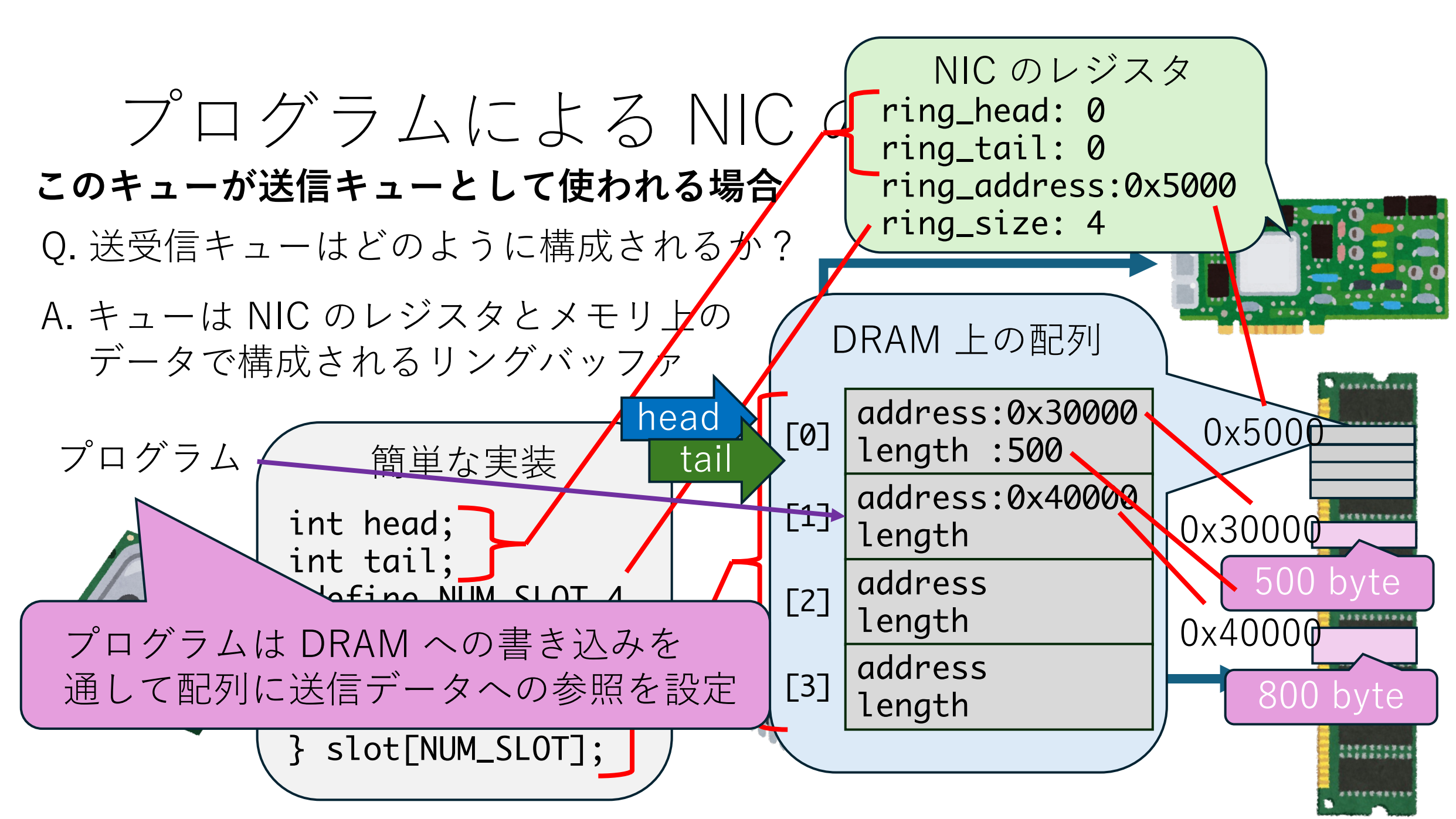


プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

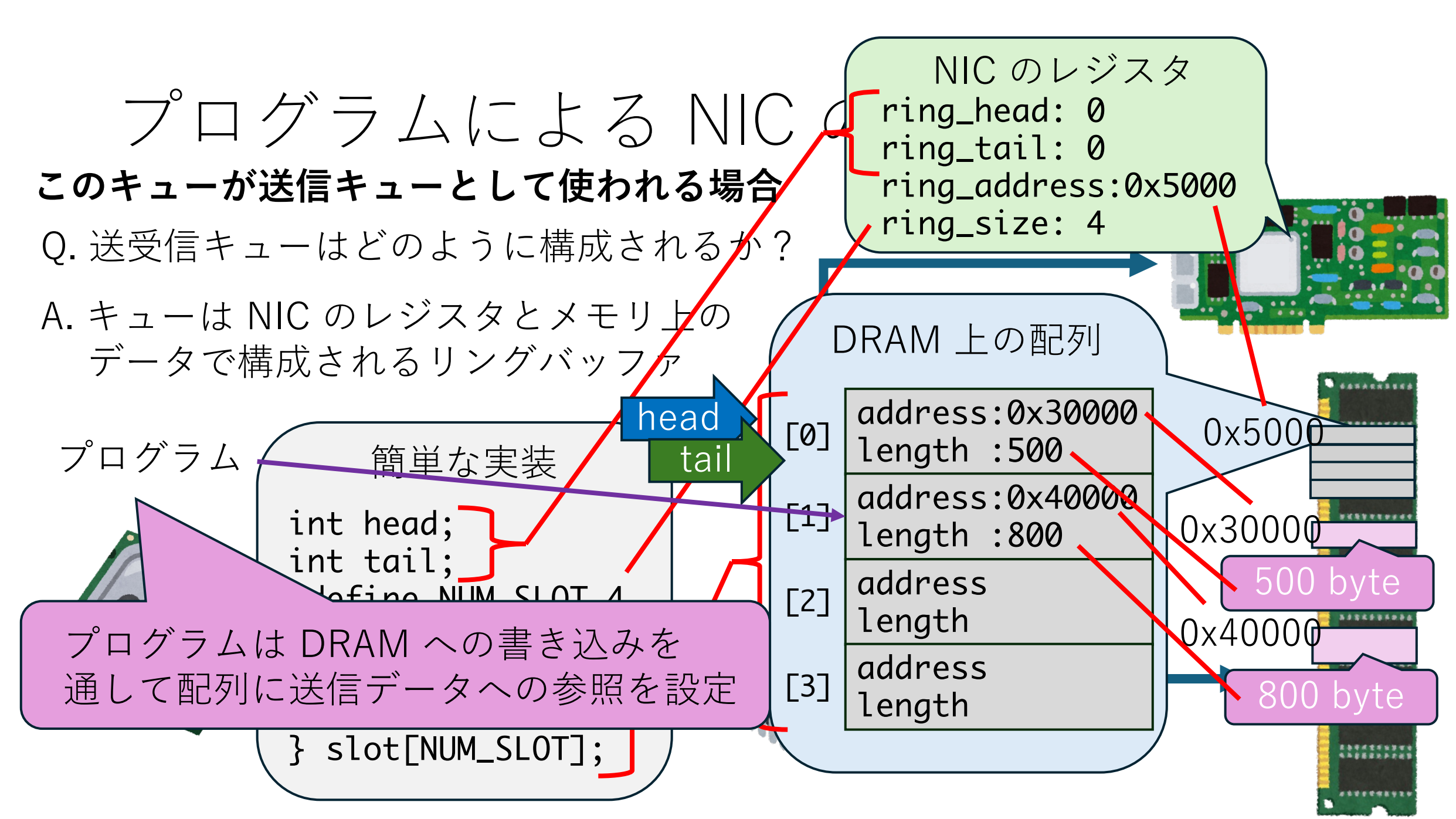


プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

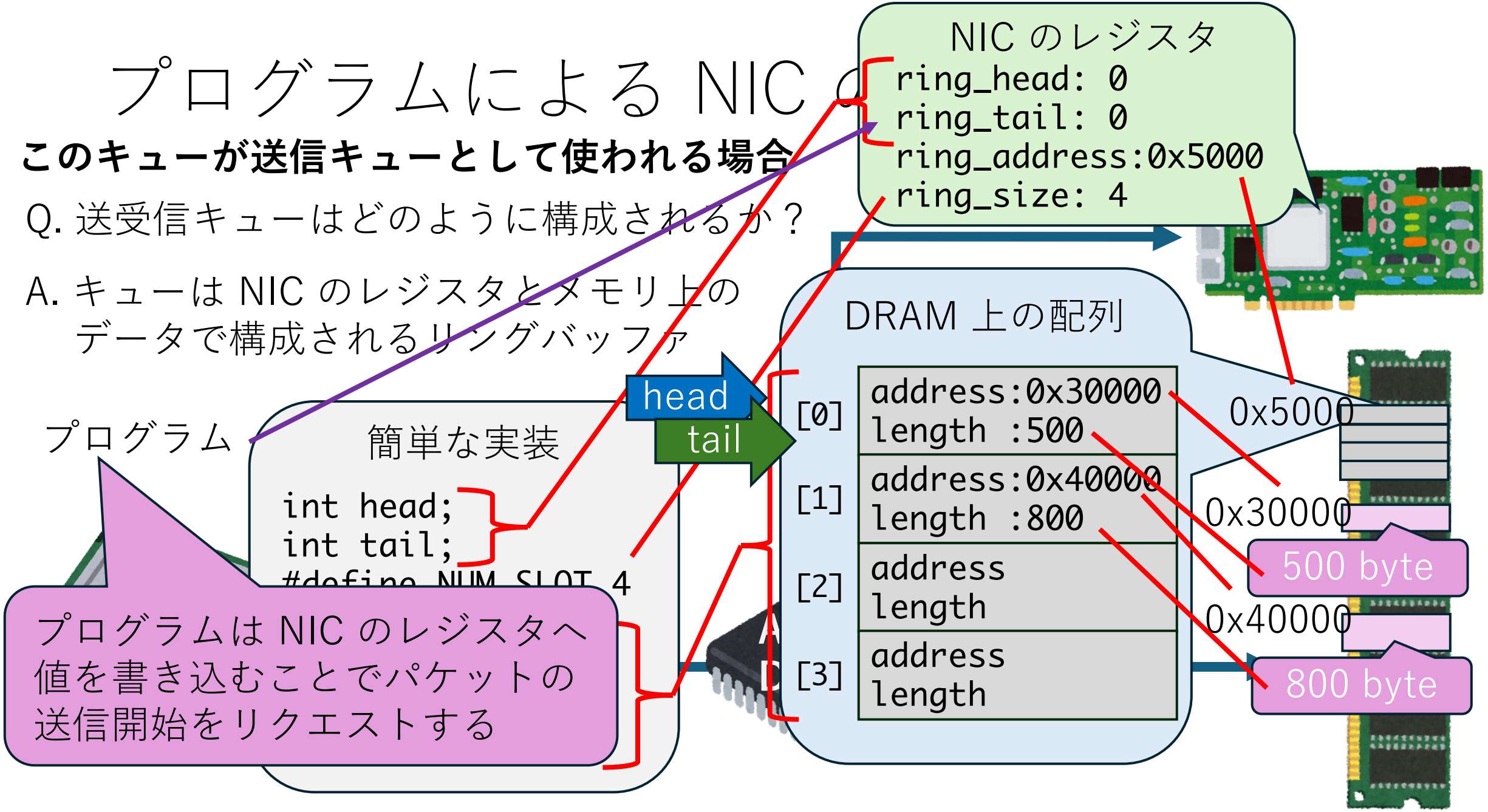


プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

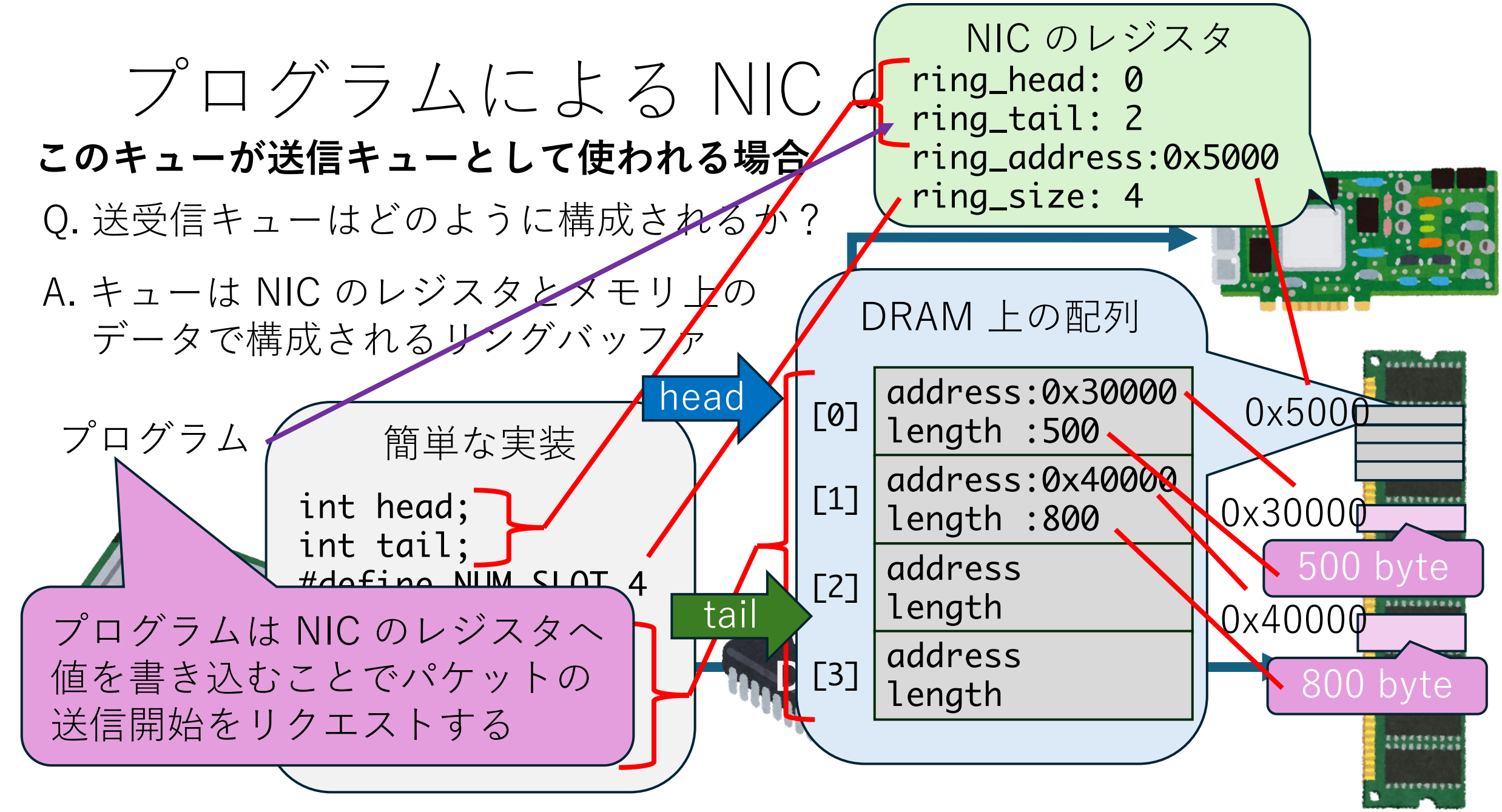


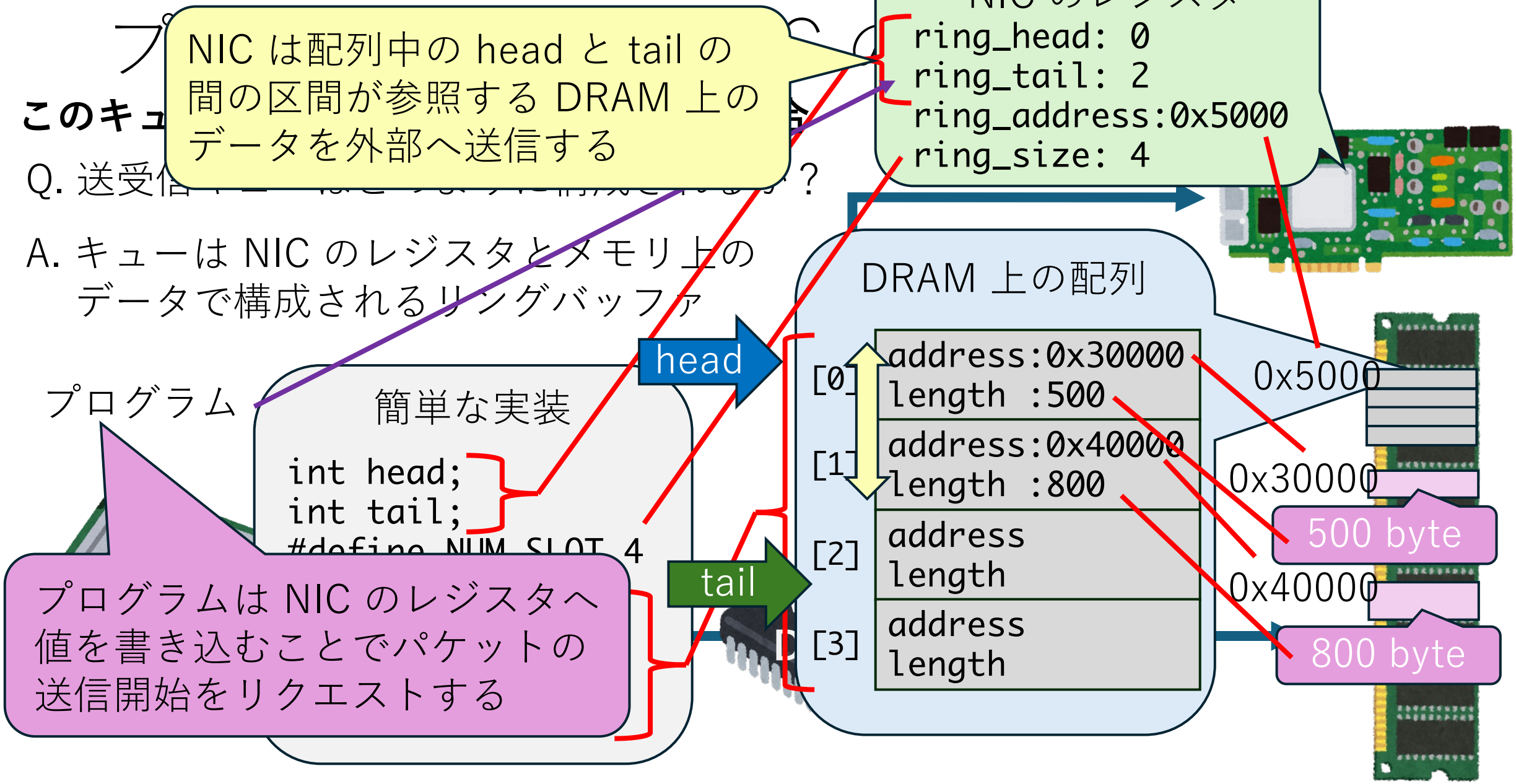
プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ




NIC は配列中の head と tail の
間の区間が参照する DRAM 上の
データを外部へ送信する
NIC のレジスタ
ring_head: 0
ring_tail: 2
ring_address:0x5000
ring_size: 4
Q. 送受信キューはどのように構成されるか？
A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ
DRAM 上の配列
head
tail
プログラム
簡単な実装
int head;
int tail;
#define NUM_SLOT 4
[0]
[1]
[2]
[3]
address:0x30000
length :500
address:0x40000
length :800
address
length
address
length
0x5000
0x30000
0x40000
500 byte
800 byte
プログラムは NIC のレジスタへ
値を書き込むことでパケットの
送信開始をリクエストする

プ

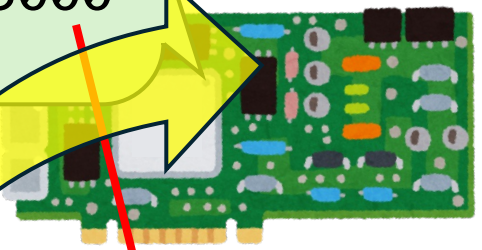
このキュ

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上のデータで構成されるリングバッファ

NIC は配列中の head と tail の間の区間が参照する DRAM 上のデータを外部へ送信する

NIC のレジスタ
```
ring_head: 0
ring_tail: 2
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address:0x30000 length :500
[1]	address:0x40000 length :800
[2]	address length
[3]	address length

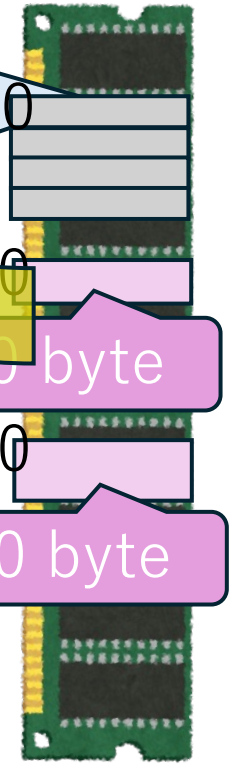
0x5000

0x30000

500 byte

0x40000

800 byte



head

tail

プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
```

プログラムは NIC のレジスタへ値を書き込むことでパケットの送信開始をリクエストする

プ

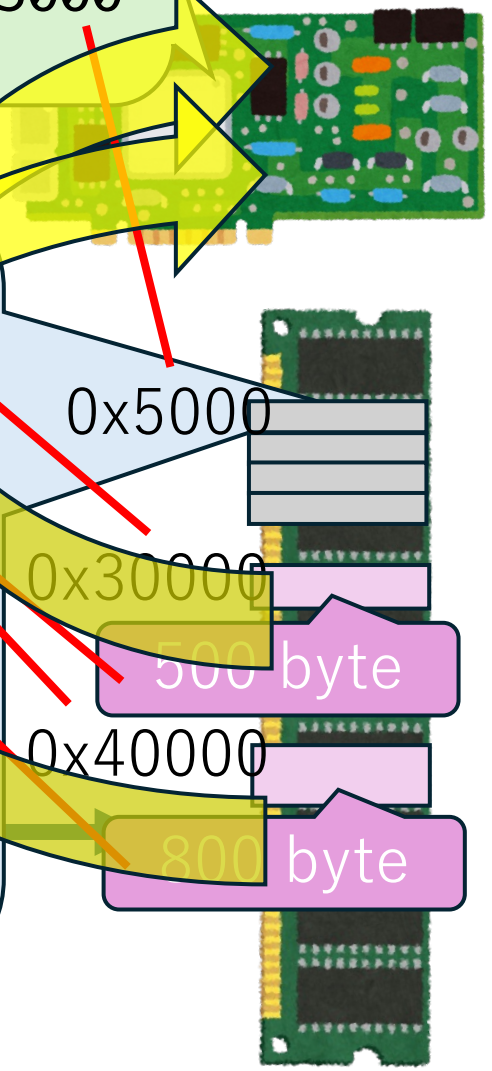
このキュ

Q. 送受信キュー はどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

NIC は配列中の head と tail の
間の区間が参照する DRAM 上の
データを外部へ送信する

NIC のレジスタ
```
ring_head: 0
ring_tail: 2
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address:0x30000 length :500
[1]	address:0x40000 length :800
[2]	address length
[3]	address length

0x5000

0x30000

500 byte

0x40000

800 byte

head

tail

プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
```

プログラムは NIC のレジスタへ
値を書き込むことでパケットの
送信開始をリクエストする

プロ

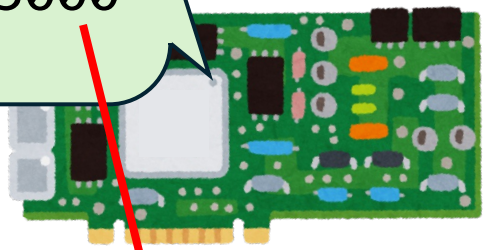
このキューが送

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

NIC は送信完了後に
レジスタの値を更新する

NIC のレジスタ
```
ring_head: 0
ring_tail: 2
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address:0x30000 length :500
[1]	address:0x40000 length :800
[2]	address length
[3]	address length

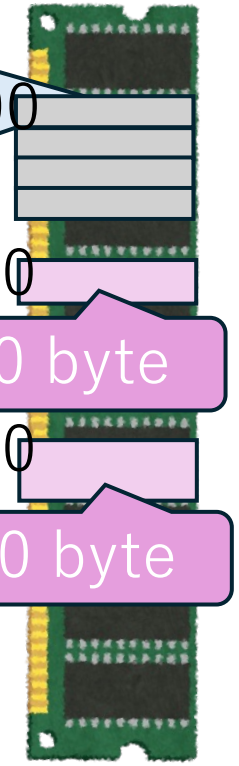
0x5000

0x30000

500 byte

0x40000

800 byte



head

tail

プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
```

プログラムは NIC のレジスタへ
値を書き込むことでパケットの
送信開始をリクエストする

プロ

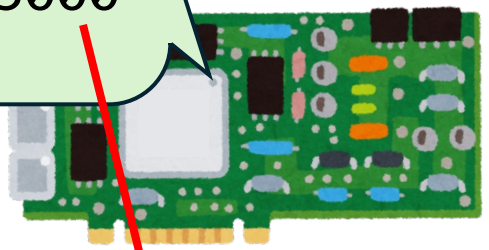
このキューが送

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

NIC は送信完了後に
レジスタの値を更新する

NIC のレジスタ
```
ring_head: 2
ring_tail: 2
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address:0x30000 length :500
[1]	address:0x40000 length :800
[2]	address length
[3]	address length

0x5000

0x30000

500 byte

0x40000

800 byte

プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
```

head

tail

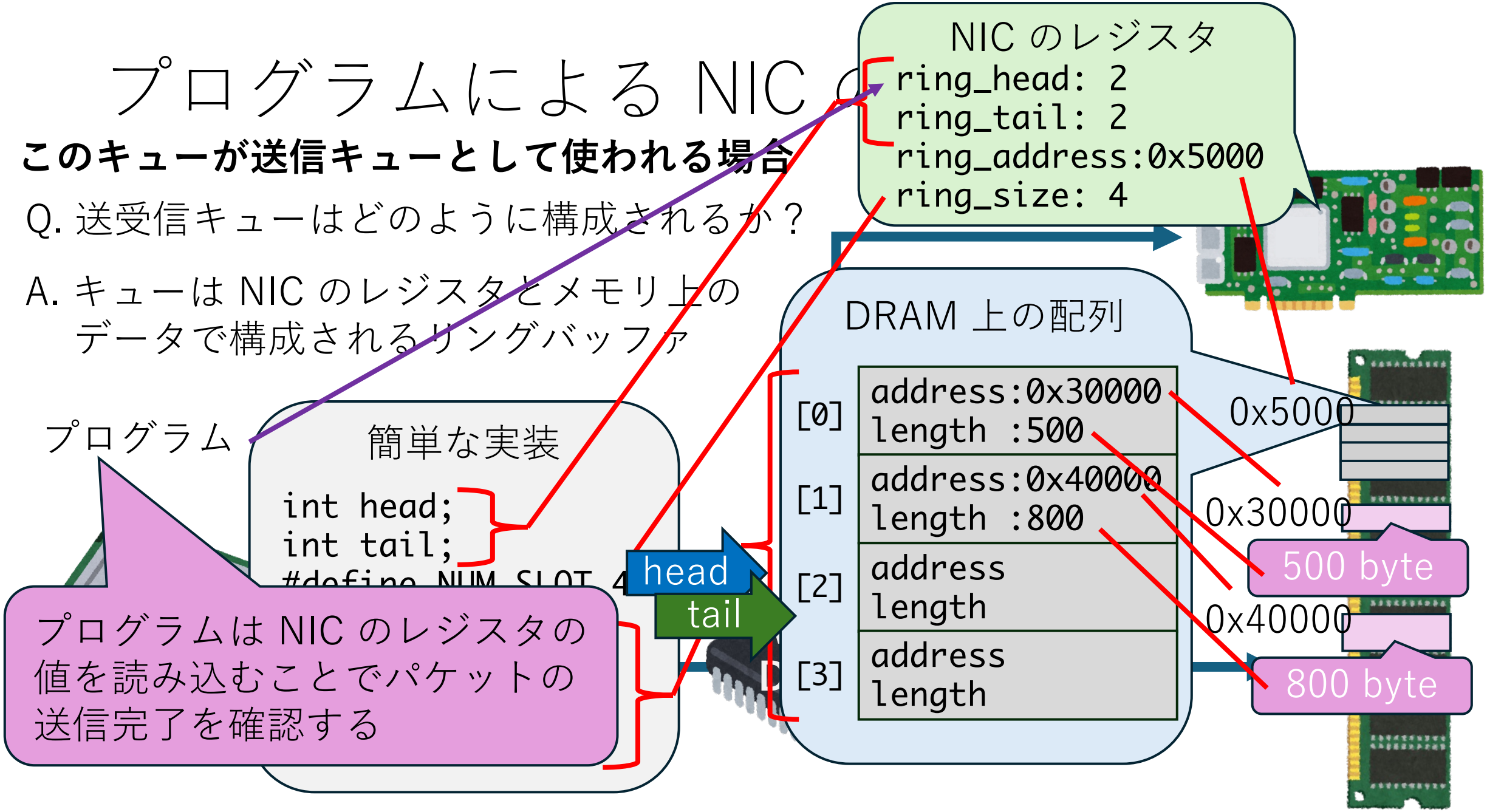
プログラムは NIC のレジスタへ
値を書き込むことでパケットの
送信開始をリクエストする

プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



プログラムによる NIC の

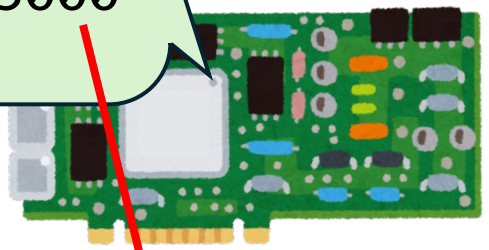
このキ

Q. 送受

A. キュ
デー

プログラムは送信が完了した
データが配置されている DRAM を
別のデータを配置するために
利用して良いと判断できる

NIC のレジスタ
ring_head: 2
ring_tail: 2
ring_address:0x5000
ring_size: 4



DRAM 上の配列

[0]	address:0x30000 length :500
[1]	address:0x40000 length :800
[2]	address length
[3]	address length

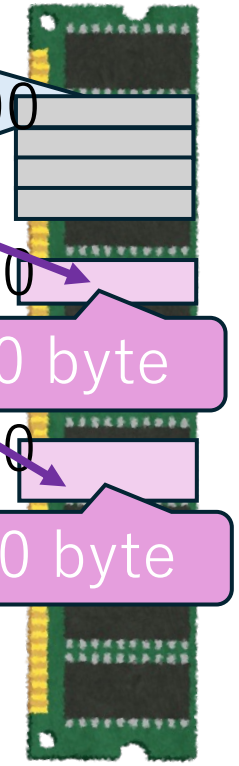
0x5000

0x30000

500 byte

0x40000

800 byte



プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
```

head

tail

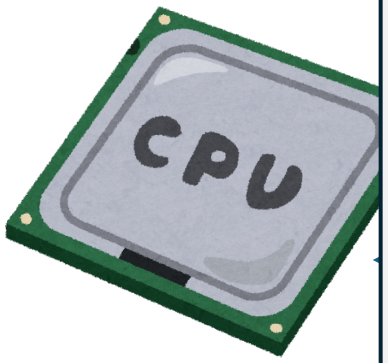
プログラムは NIC のレジスタの
値を読み込むことでパケットの
送信完了を確認する

プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



プログラム

簡単な実装

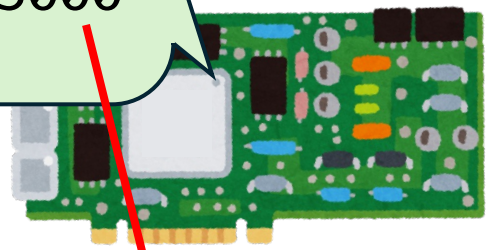
```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

head

tail

NIC のレジスタ

```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address length
[1]	address length
[2]	address length
[3]	address length

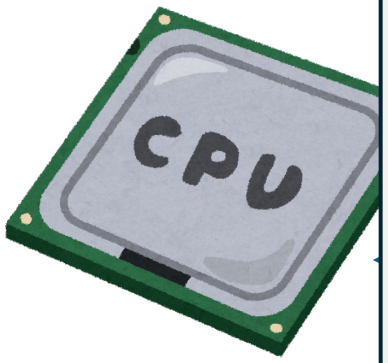
0x5000

プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



プログラム

簡単な実装

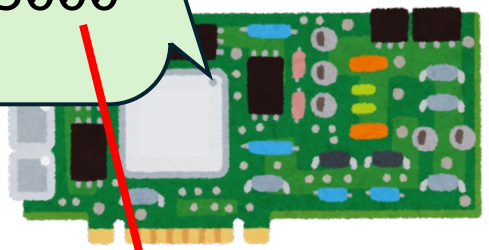
```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

head

tail

NIC のレジスタ

```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address length
[1]	address length
[2]	address length
[3]	address length

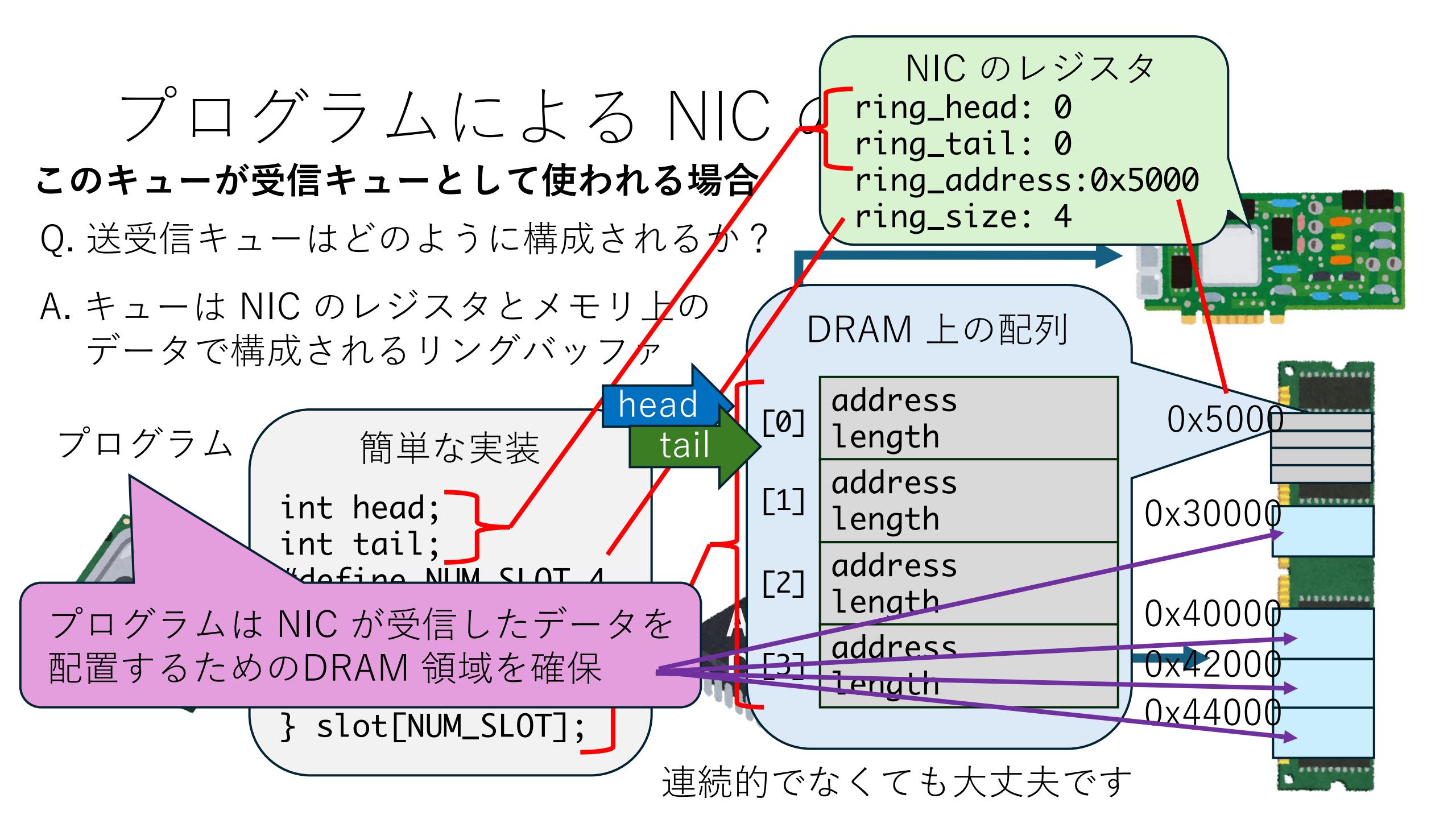
0x5000

プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

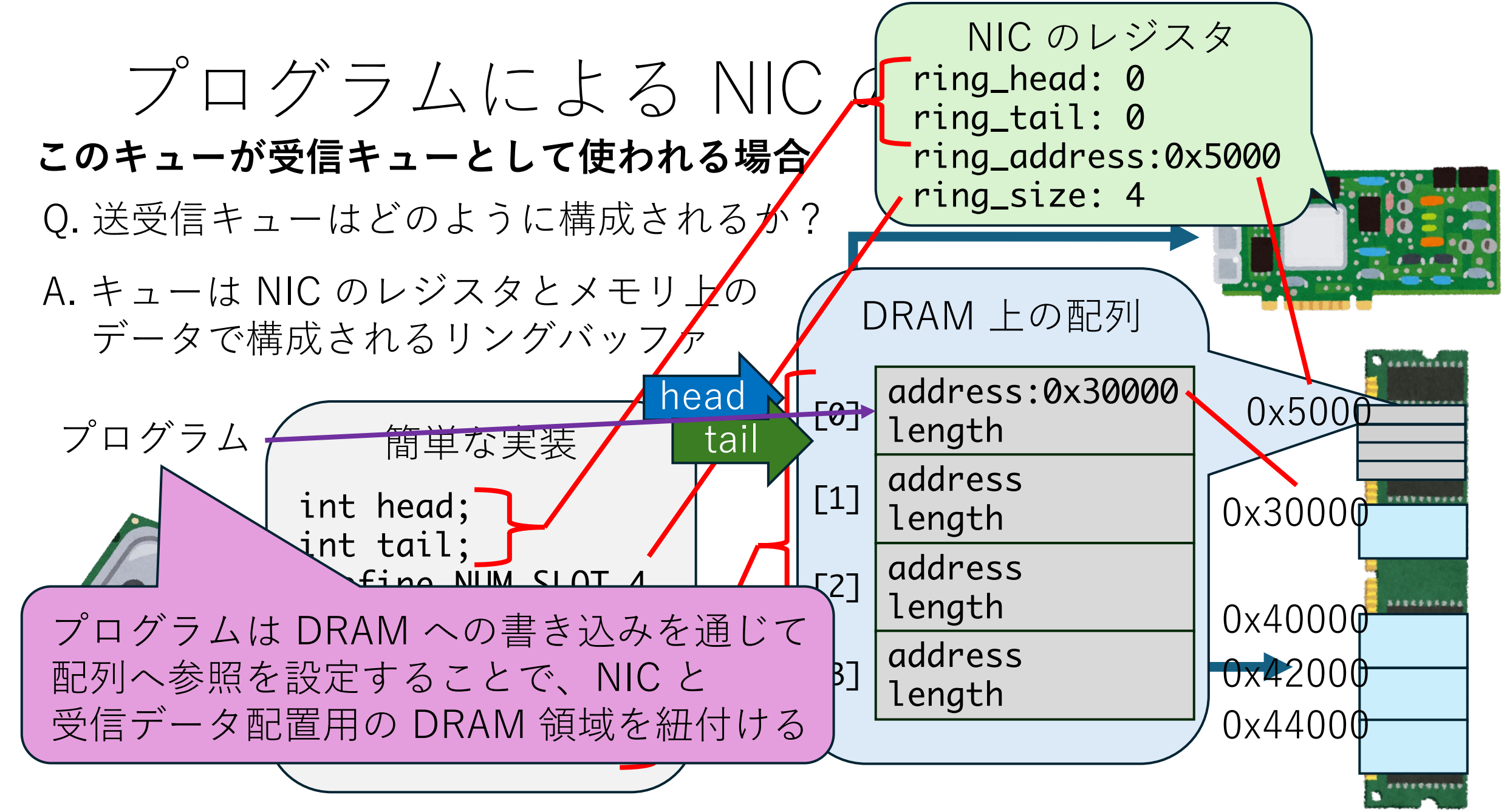


プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

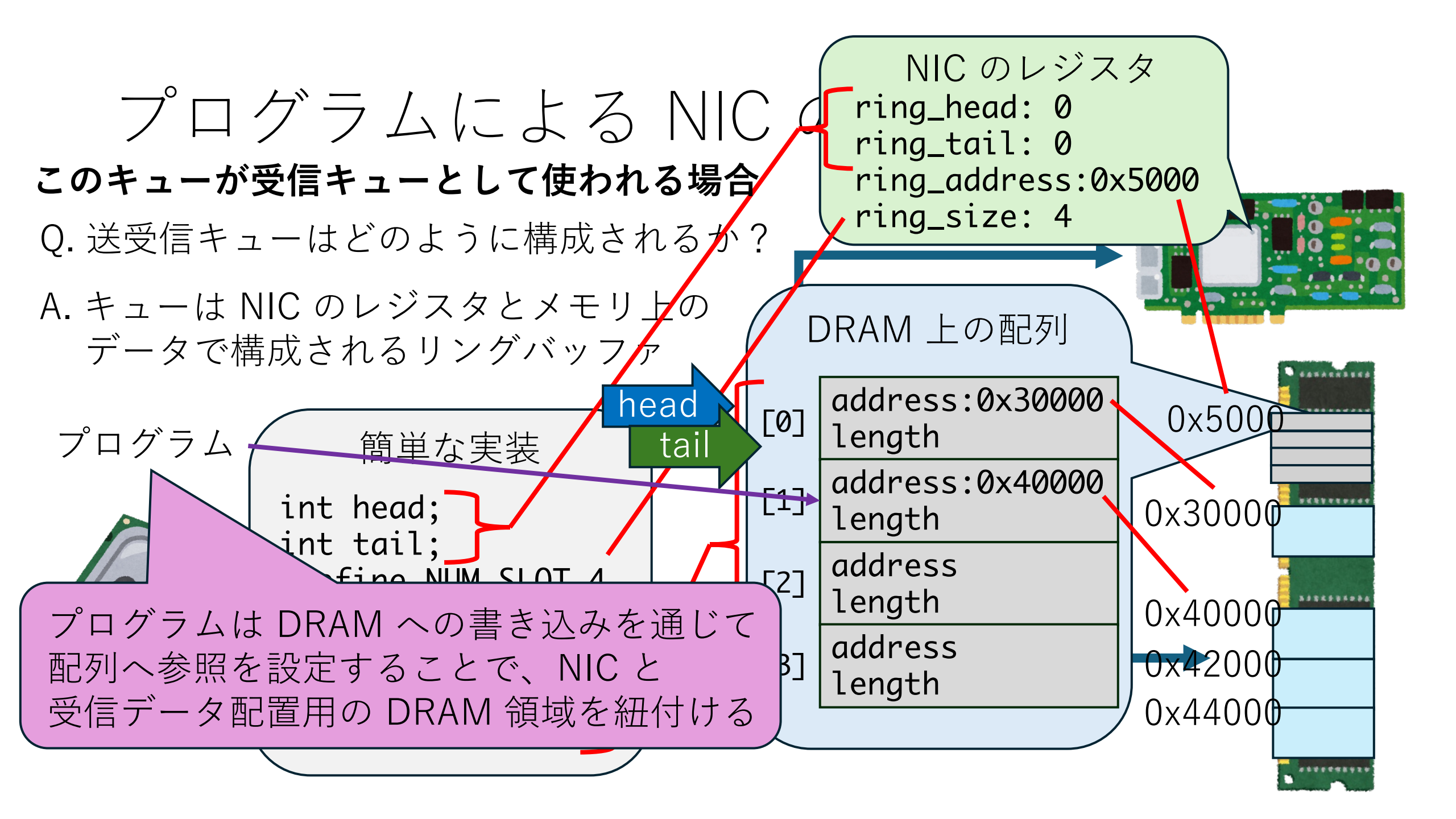


プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

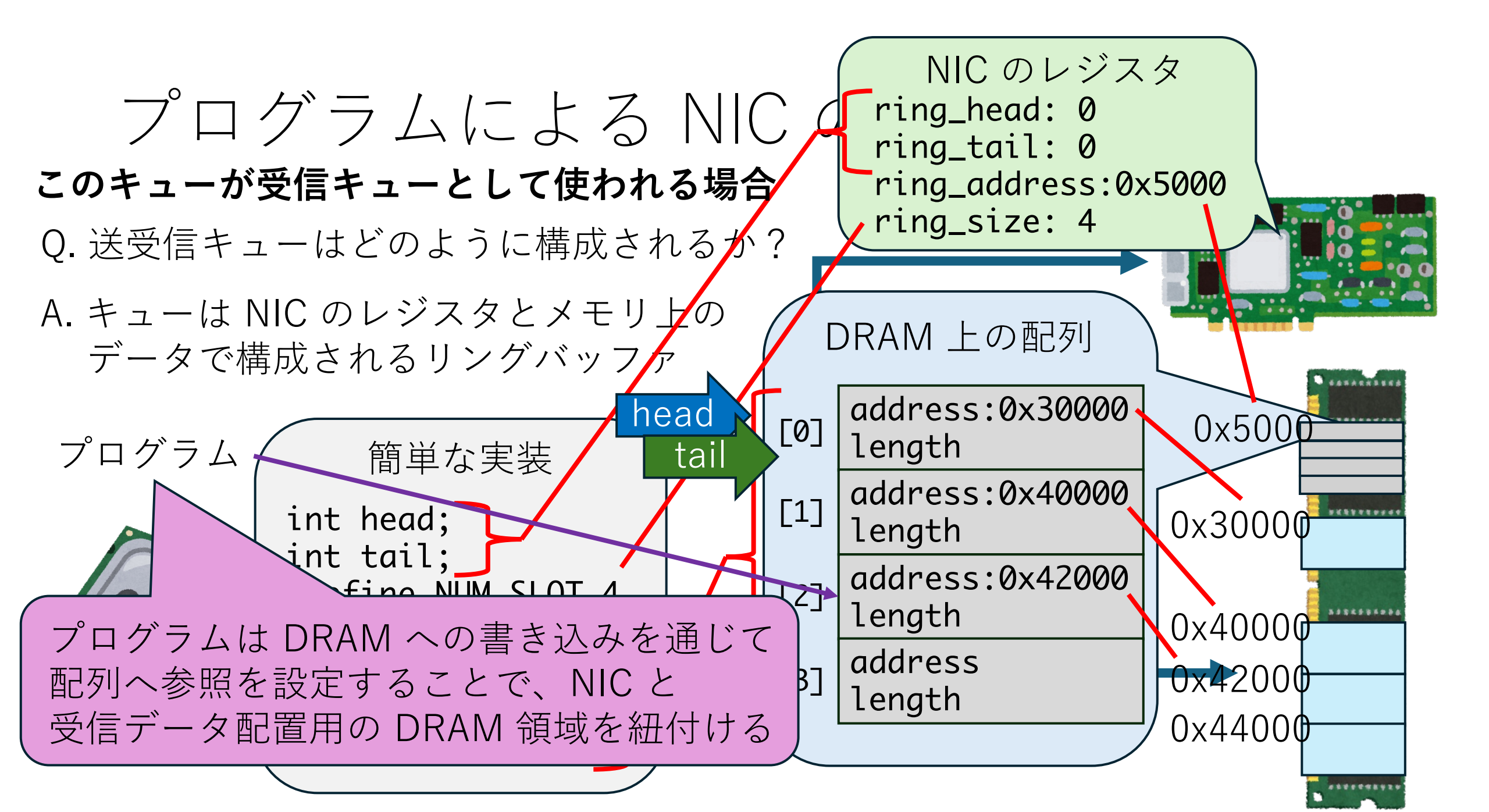


プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

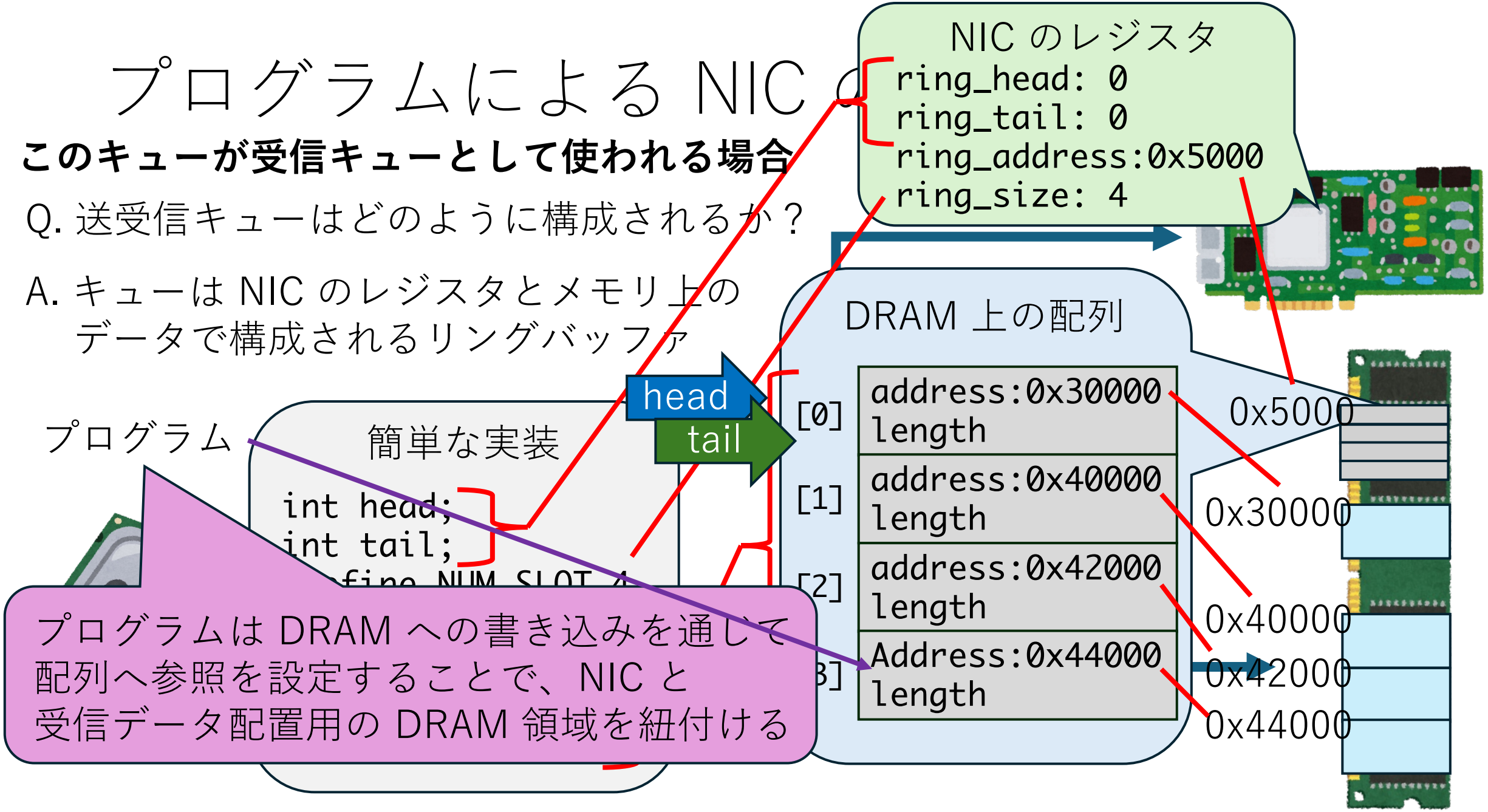


プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

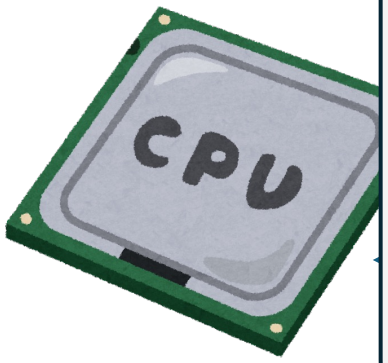


プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



プログラム

簡単な実装

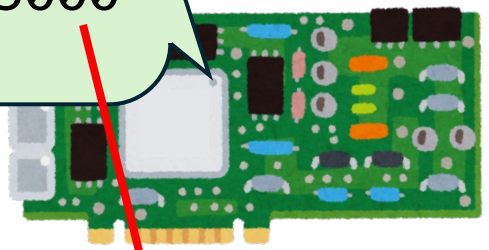
```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

head

tail

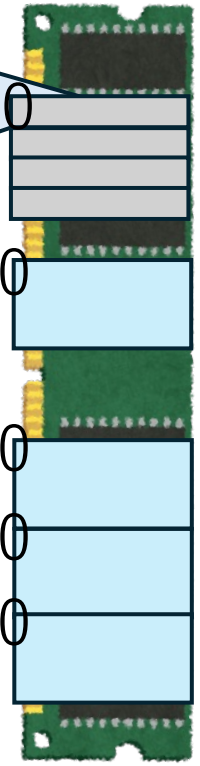
NIC のレジスタ

```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address:0x30000 length
[1]	address:0x40000 length
[2]	address:0x42000 length
[3]	Address:0x44000 length



0x5000

0x30000

0x40000

0x42000

0x44000

プログ

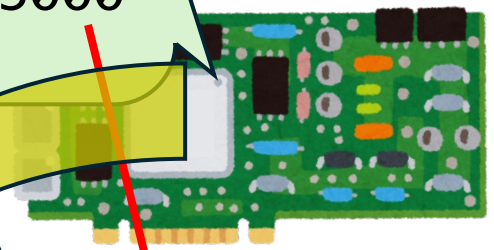
このキューが受信

Q. 送受信キューはどのように構成さ？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

NIC が受信したパケットを
配列の head で参照される
DRAM 領域へ書き込む

NIC のレジスタ
```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



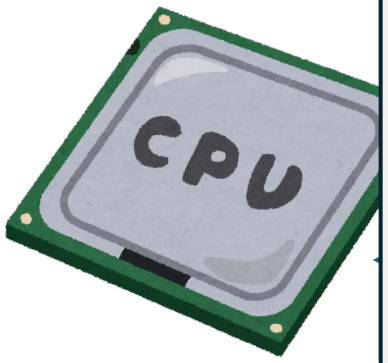
DRAM 上の配列

[0]	address:0x30000 length
[1]	address:0x40000 length
[2]	address:0x42000 length
[3]	Address:0x44000 length

0x5000
0x30000
100 byte
0x40000
0x42000
0x44000

head
tail

プログラム



簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

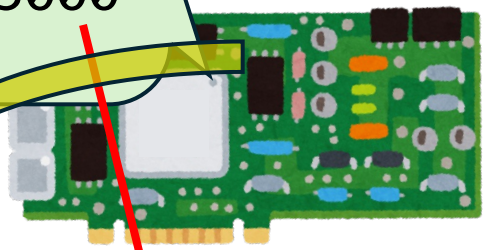
プロ C
このキュー 合

Q. 送受信キューはどのように構 るか？

A. キューは NIC のレジスタとメモリ上
データで構成されるリングバッファ

NIC は配列の長さを表す
フィールド(length) に
受信したデータの長さを設定

NIC のレジスタ
```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

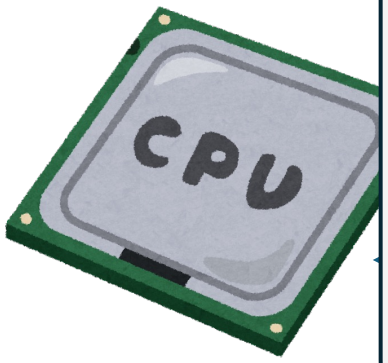
[0]	address:0x30000 length :100
[1]	address:0x40000 length
[2]	address:0x42000 length
[3]	Address:0x44000 length

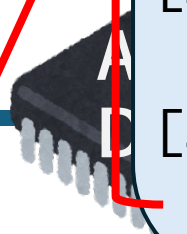
head
tail

プログラム

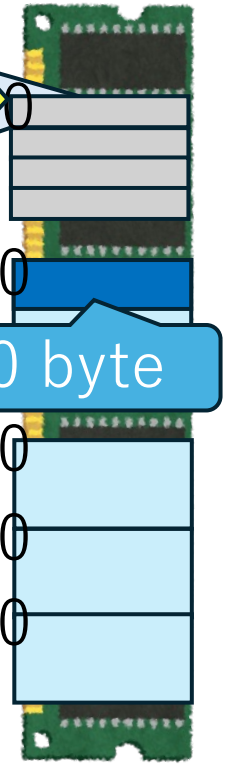
簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```





0x5000
0x30000
100 byte
0x40000
0x42000
0x44000



プログラムとNICの

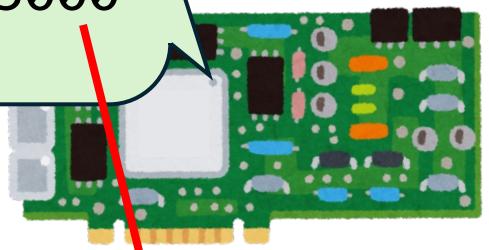
このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

NIC は head の値を進める

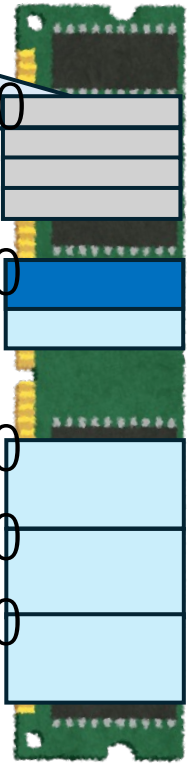
NIC のレジスタ
```
ring_head: 1
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



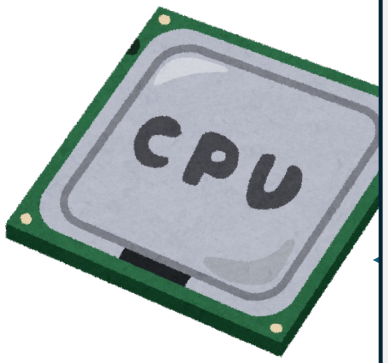
DRAM 上の配列

[0]	address:0x30000 length :100
[1]	address:0x40000 length
[2]	address:0x42000 length
[3]	Address:0x44000 length

0x5000
0x30000
0x40000
0x42000
0x44000



プログラム



簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

tail

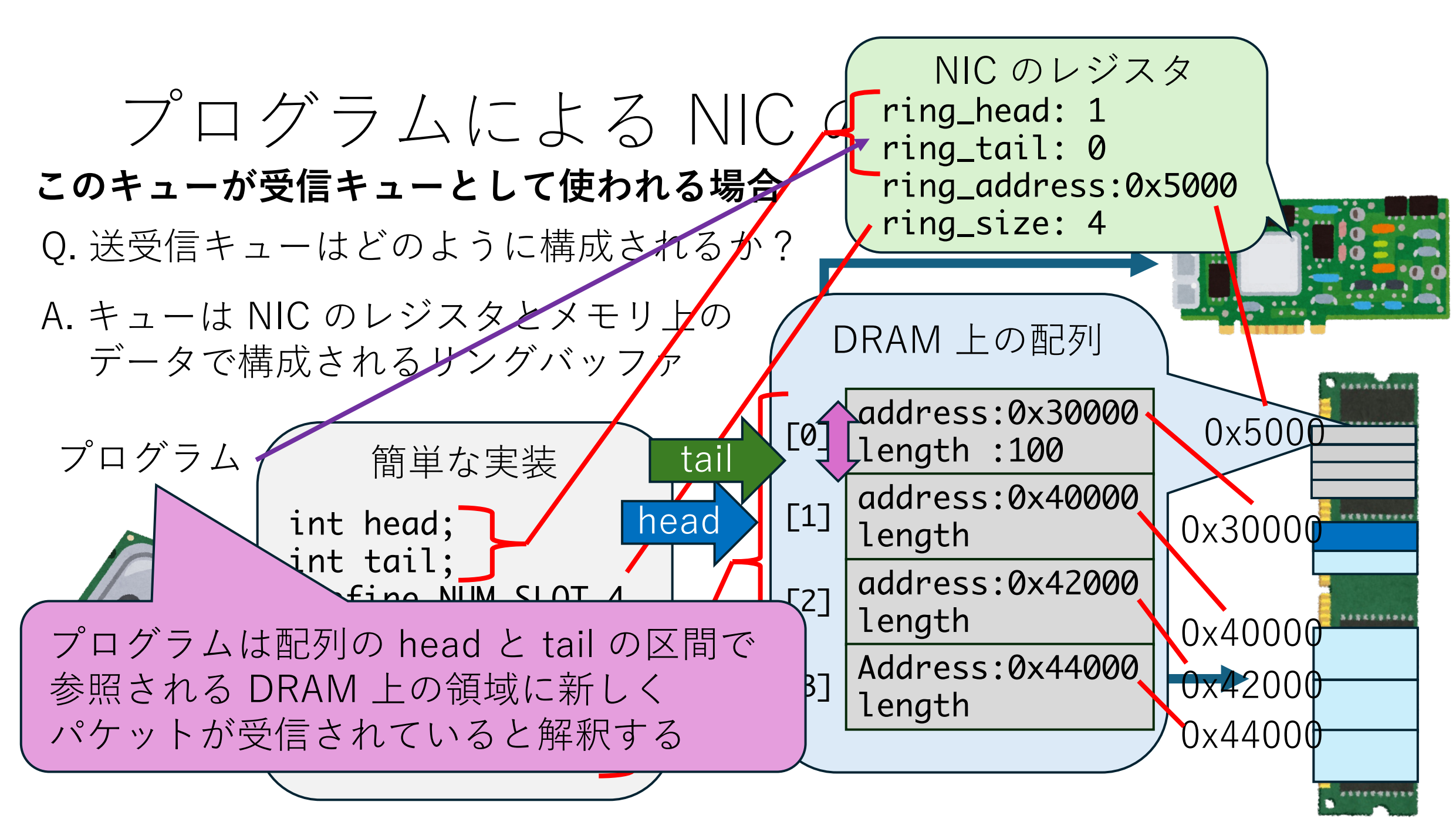
head

プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

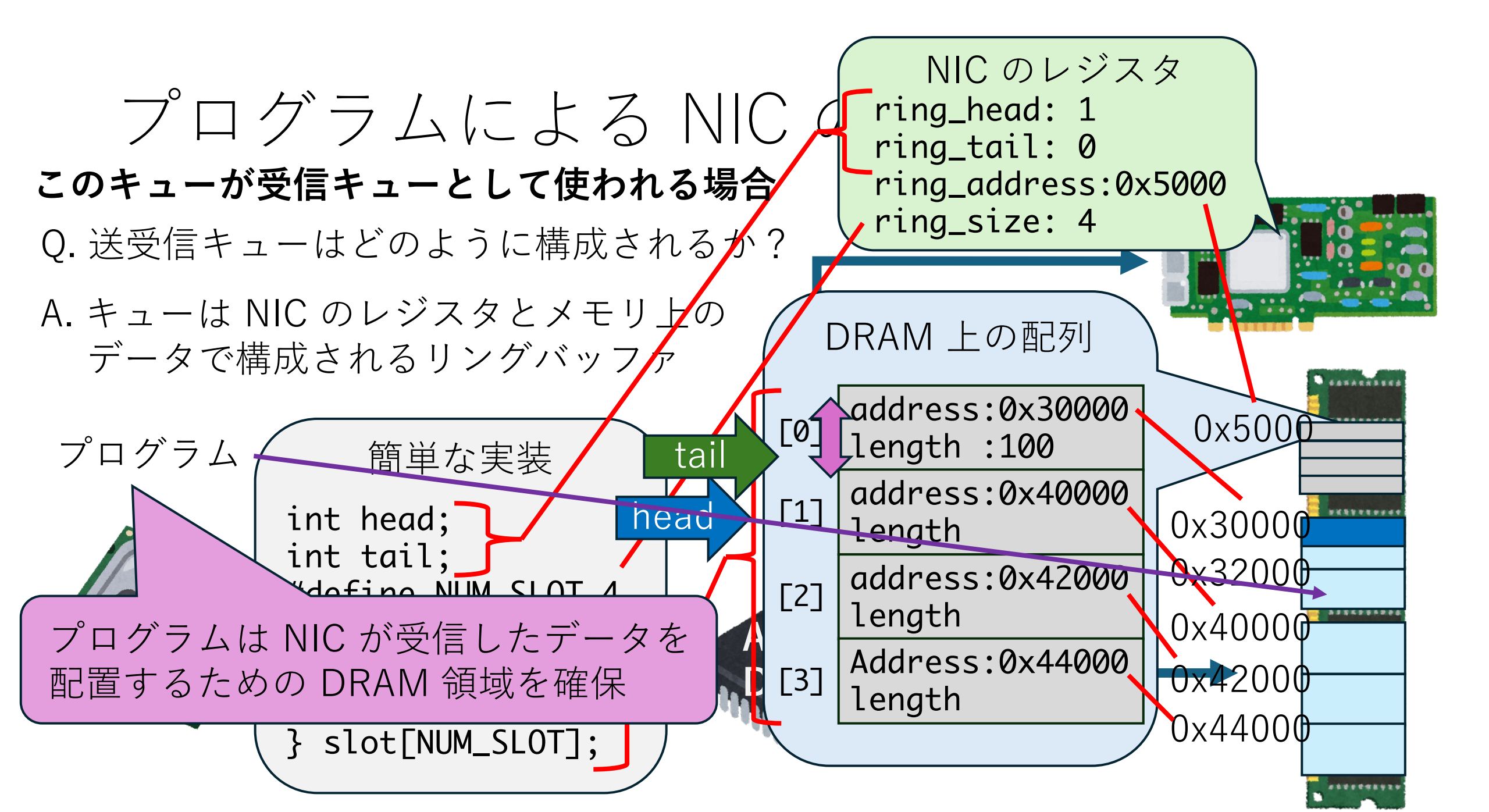


プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

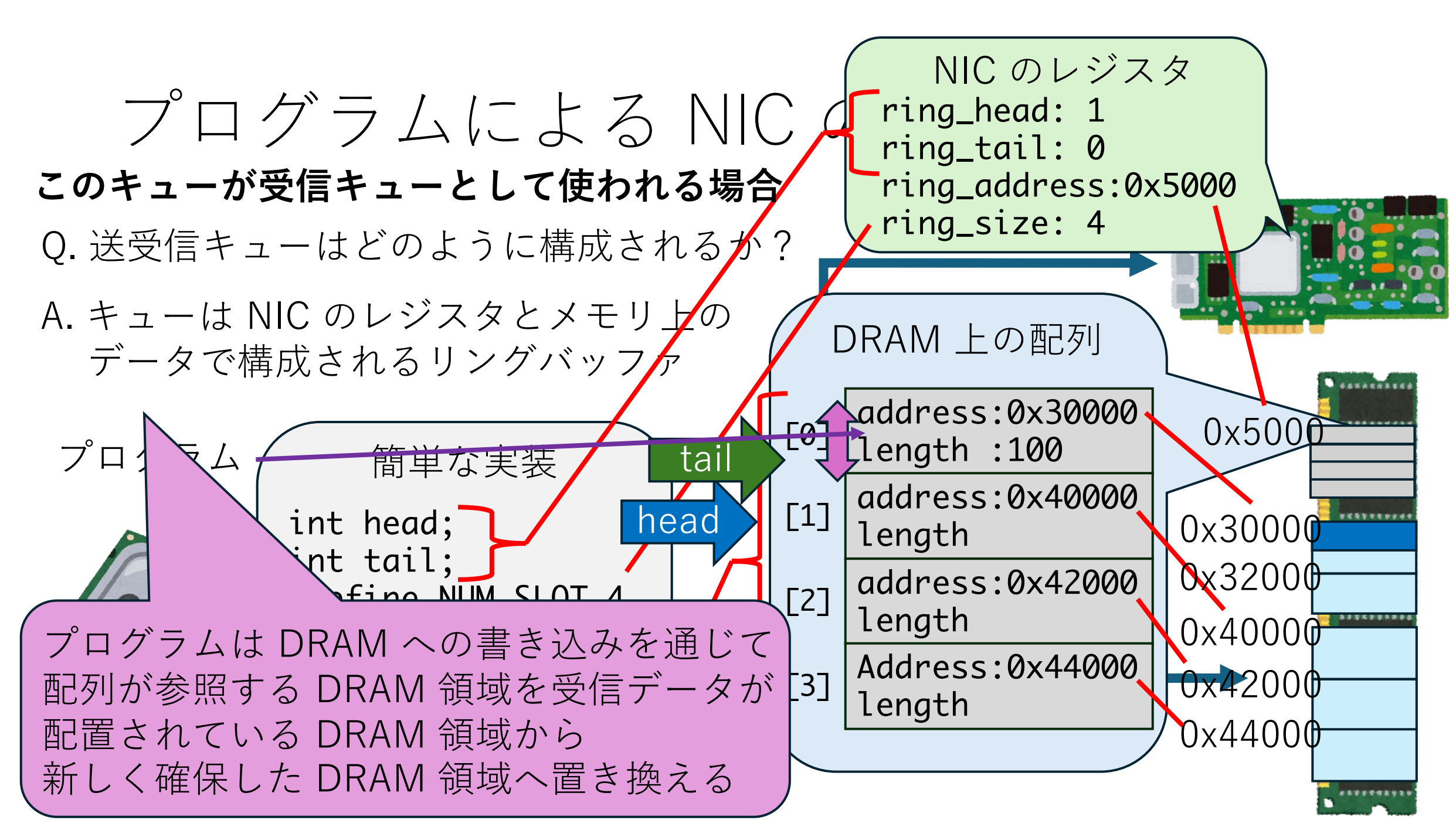


プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

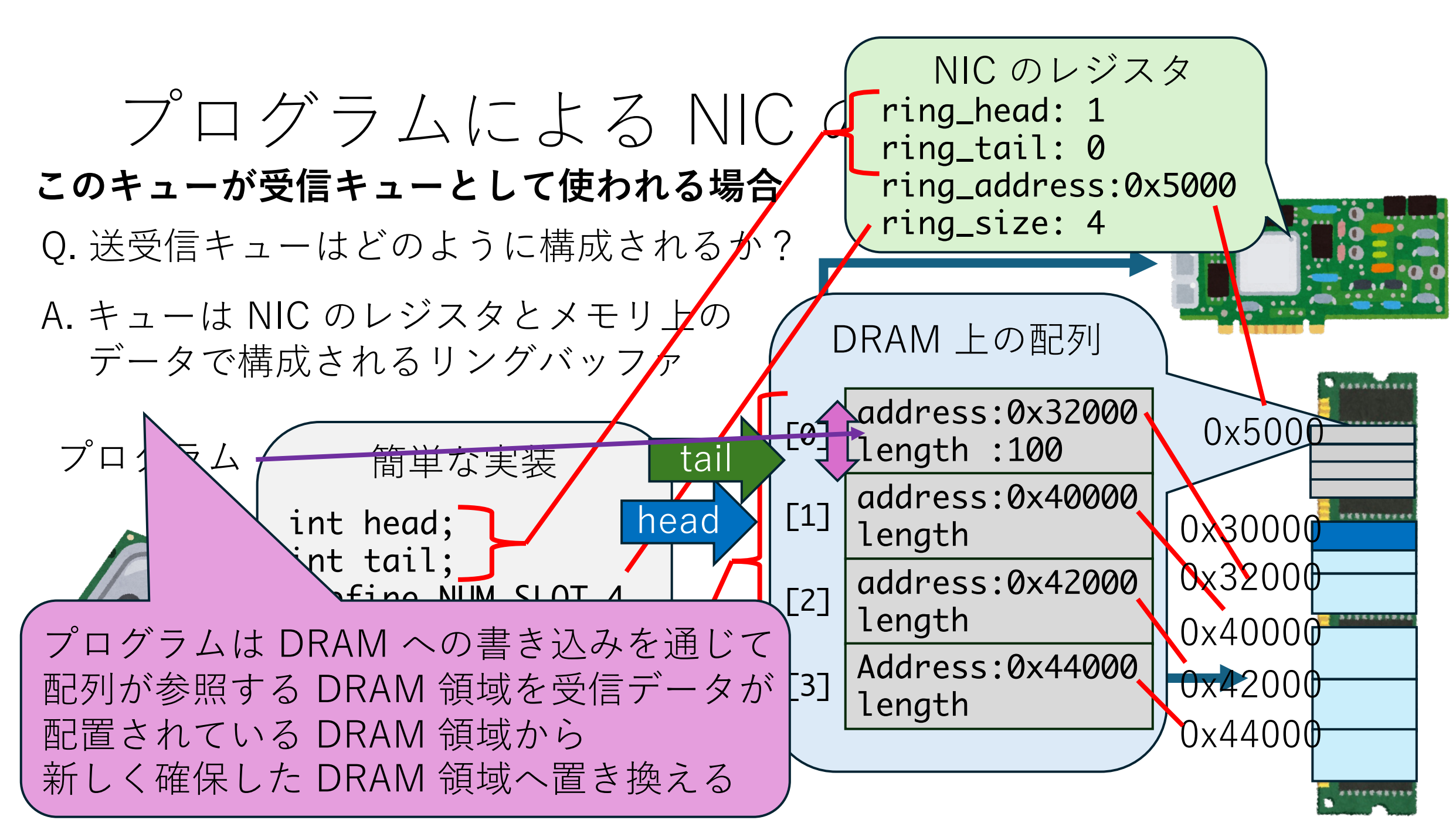


プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



プログラムによる NIC の

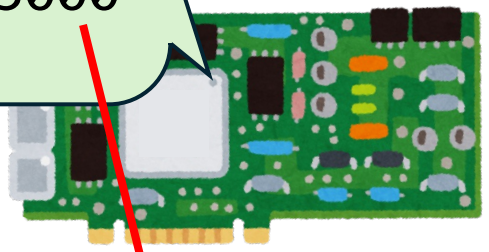
この

Q. 送

A. キ
データで構成されるリングバッファ

これで物理アドレス 0x30000 からの
DRAM 領域は NIC の紐付けが解除され
新たに物理アドレス 0x32000 からの領域が
NIC と紐付けられる

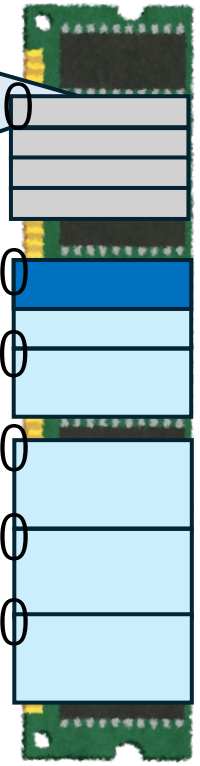
NIC のレジスタ
```
ring_head: 1
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address:0x32000 length :100
[1]	address:0x40000 length
[2]	address:0x42000 length
[3]	Address:0x44000 length

0x5000



0x30000
0x32000
0x40000
0x42000
0x44000

プログラム

簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
```

tail

head

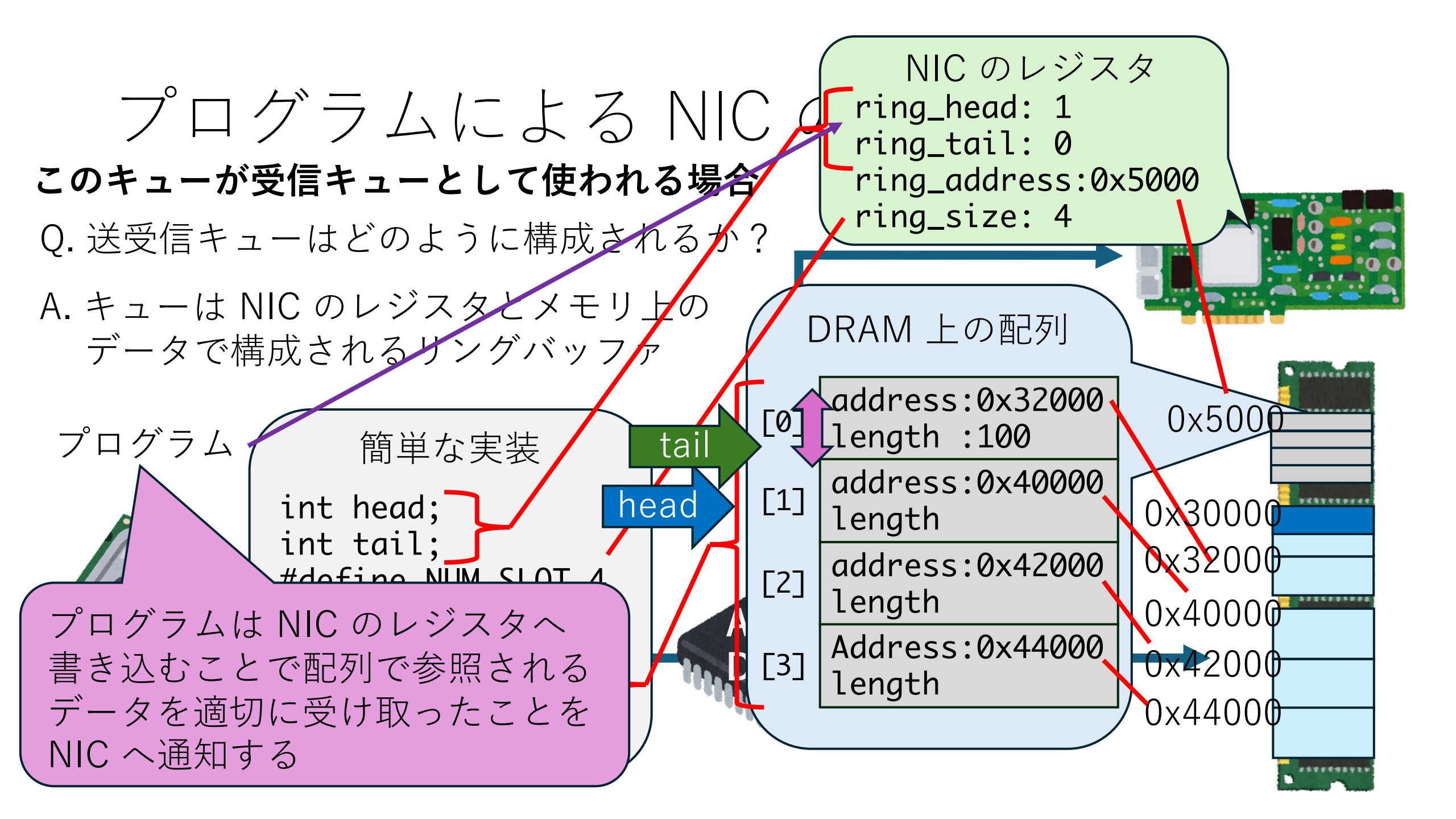
プログラムは DRAM への書き込みを通じて
配列が参照する DRAM 領域を受信データが
配置されている DRAM 領域から
新しく確保した DRAM 領域へ置き換える

プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

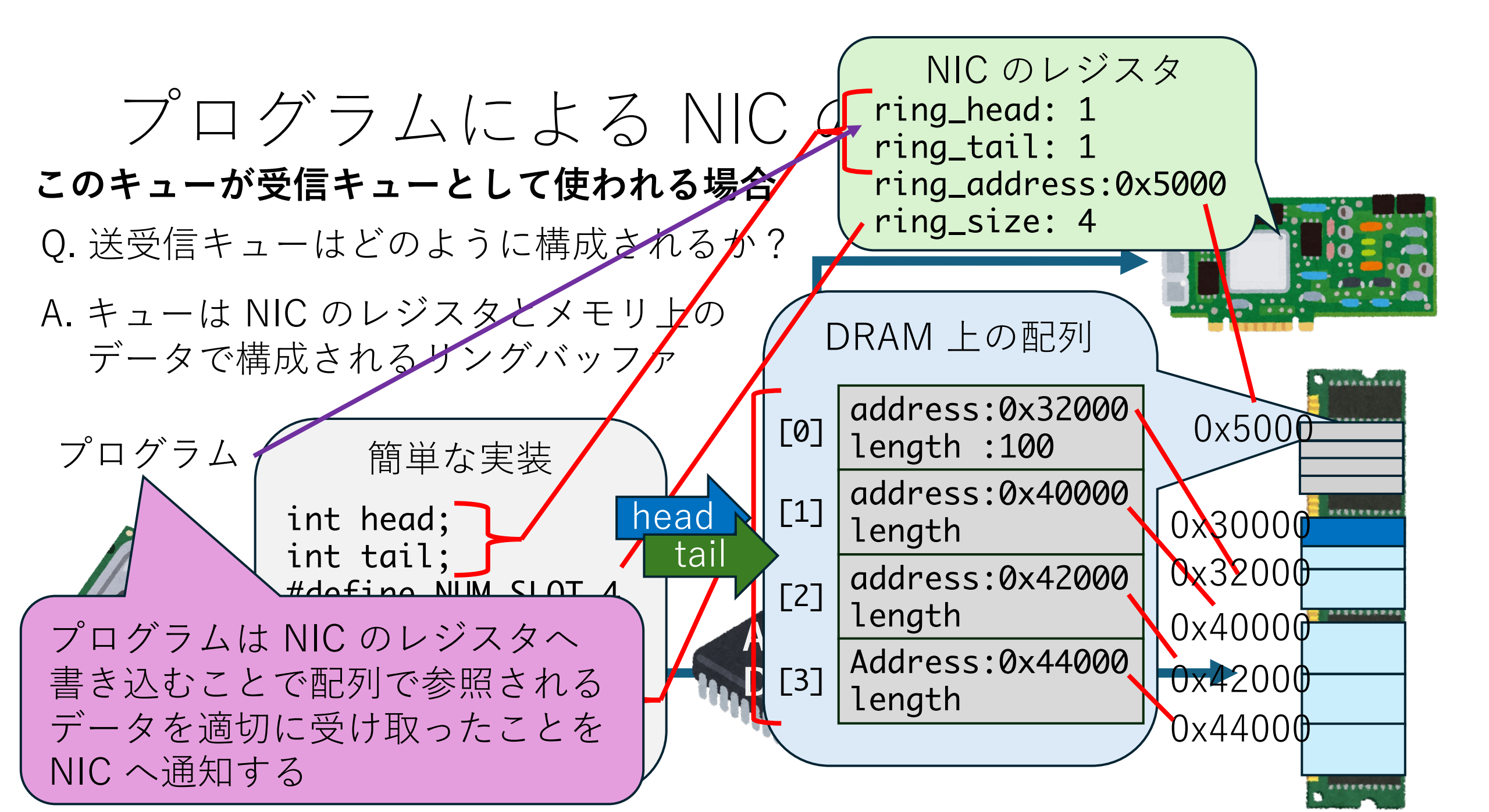


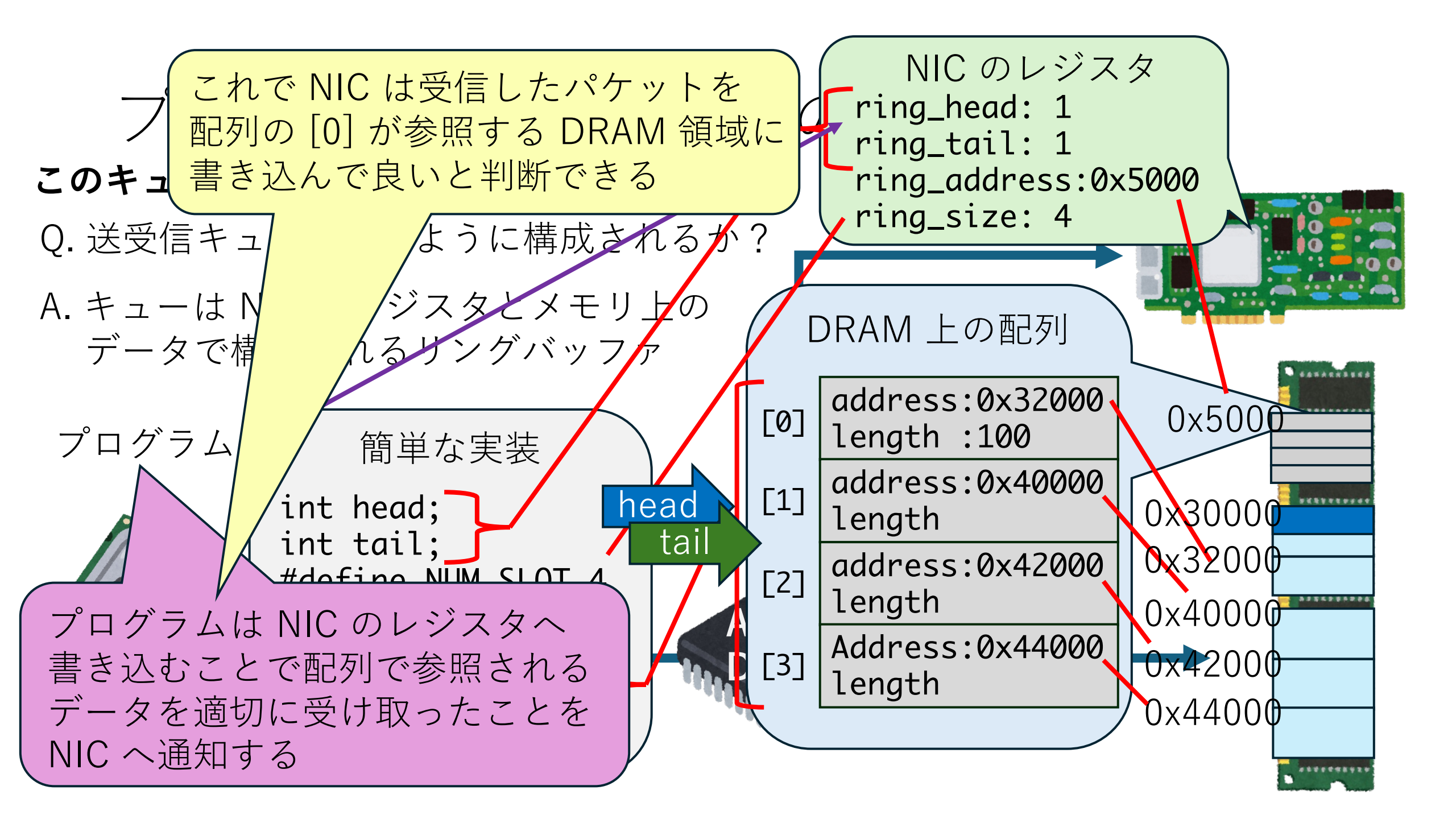
プログラムによる NIC の

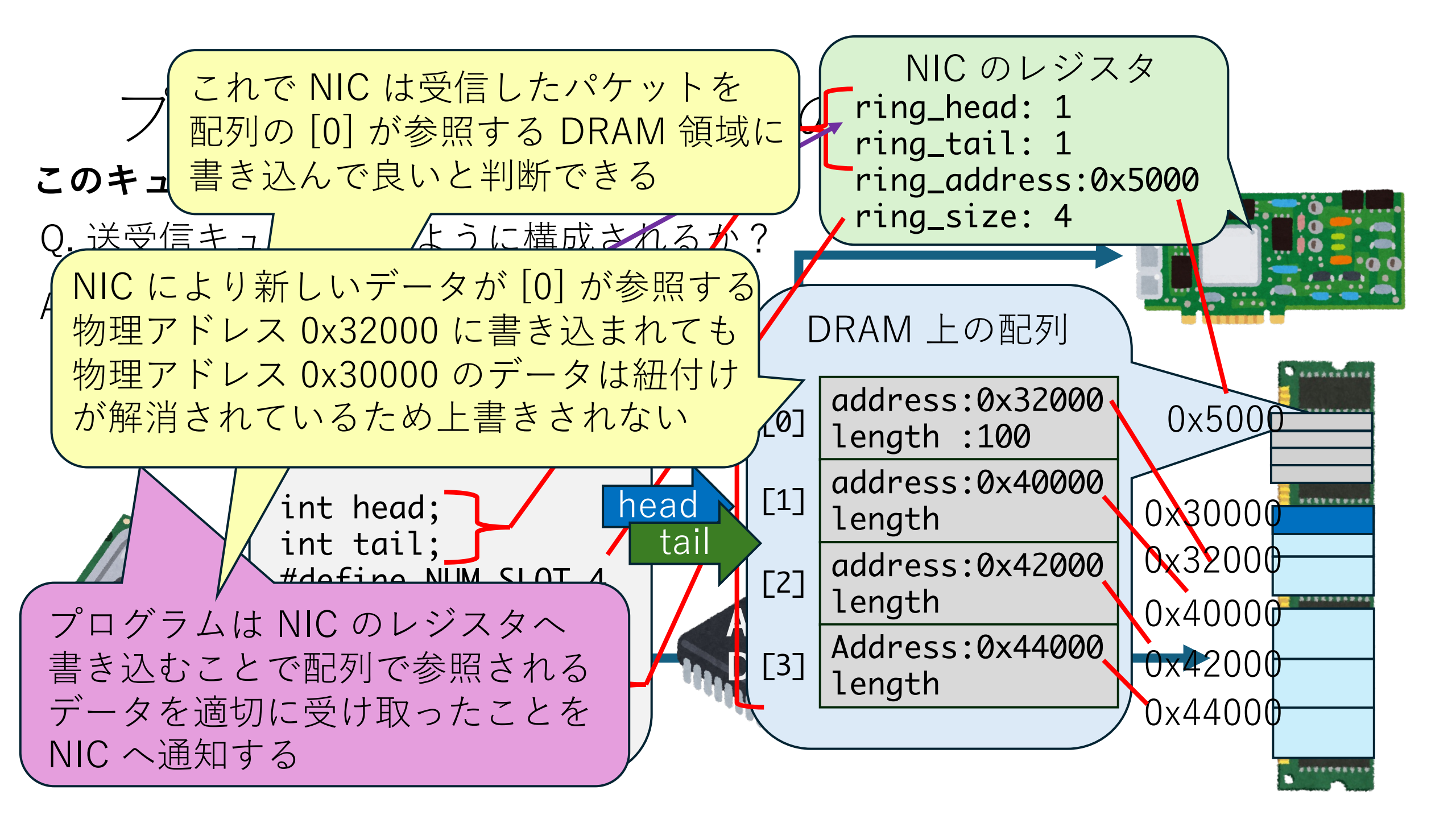
このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ




これで NIC は受信したパケットを
配列の [0] が参照する DRAM 領域に
書き込んで良いと判断できる
NIC のレジスタ
ring_head: 1
ring_tail: 1
ring_address:0x5000
ring_size: 4
このキュ
Q. 送受信キュ
ように構成されるか？
A. キューは N
ジスタとメモリ上の
データで構
れるリングバッファ
プログラム
簡単な実装
int head;
int tail;
#define NUM_SLOT 4
head
tail
DRAM 上の配列
[0]
address:0x32000
length :100
[1]
address:0x40000
length
[2]
address:0x42000
length
[3]
Address:0x44000
length
0x5000
0x30000
0x32000
0x40000
0x42000
0x44000
プログラムは NIC のレジスタへ
書き込むことで配列で参照される
データを適切に受け取ったことを
NIC へ通知する

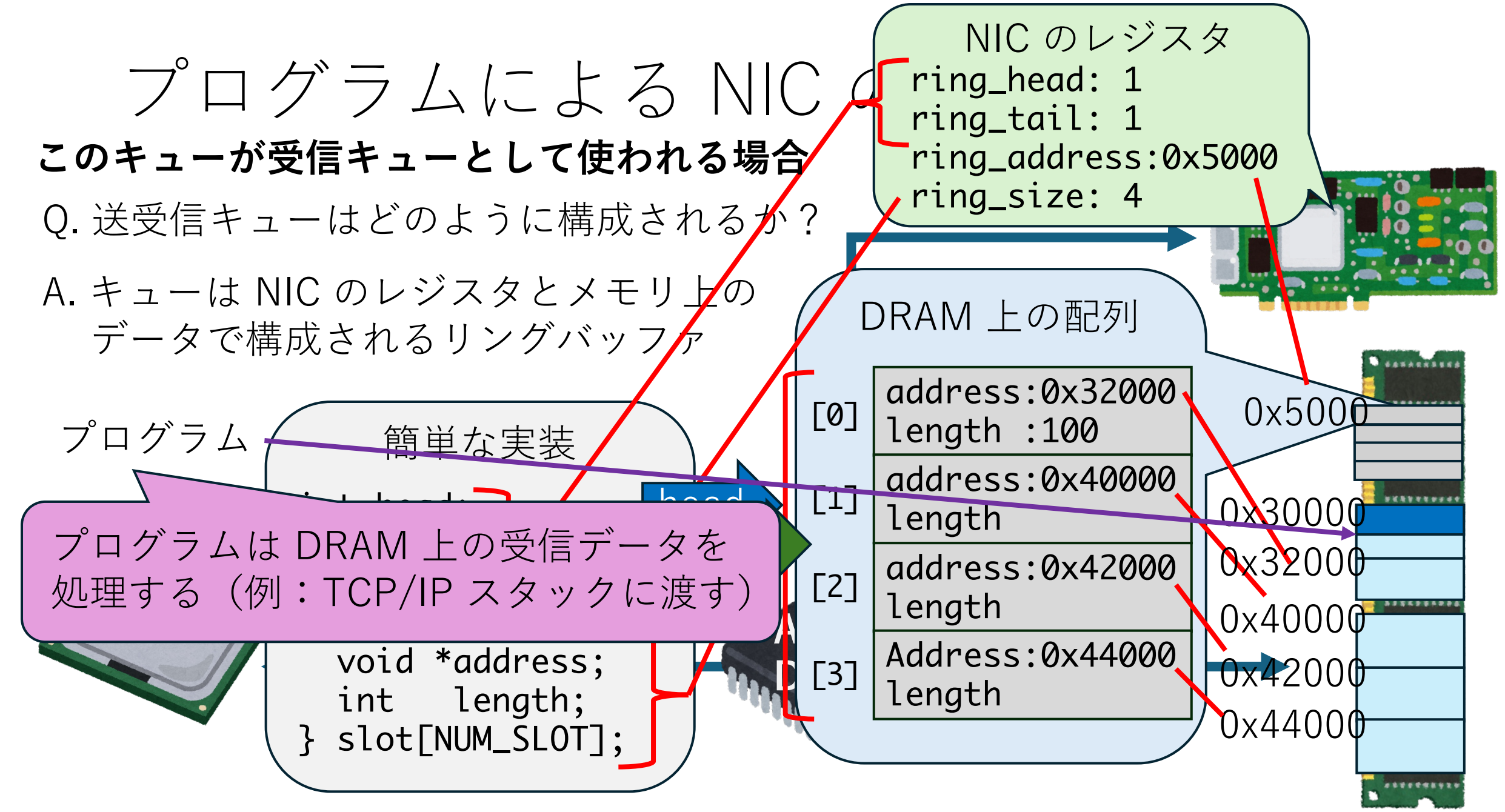

これで NIC は受信したパケットを
配列の [0] が参照する DRAM 領域に
書き込んで良いと判断できる
NIC のレジスタ
ring_head: 1
ring_tail: 1
ring_address:0x5000
ring_size: 4
NIC により新しいデータが [0] が参照する
物理アドレス 0x32000 に書き込まれても
物理アドレス 0x30000 のデータは紐付け
が解消されているため上書きされない
DRAM 上の配列
[0] address:0x32000 length :100
[1] address:0x40000 length
[2] address:0x42000 length
[3] Address:0x44000 length
0x5000
0x30000
0x32000
0x40000
0x42000
0x44000
int head;
int tail;
head
tail
プログラムは NIC のレジスタへ
書き込むことで配列で参照される
データを適切に受け取ったことを
NIC へ通知する

プログラムによる NIC の

このキューが受信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

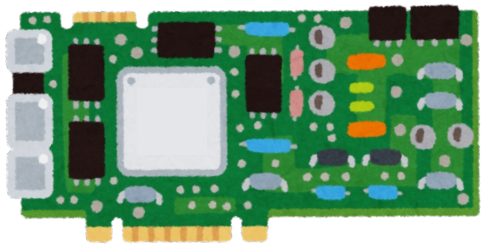
Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

メモリの読み書きを通じたデバイス操作

仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	

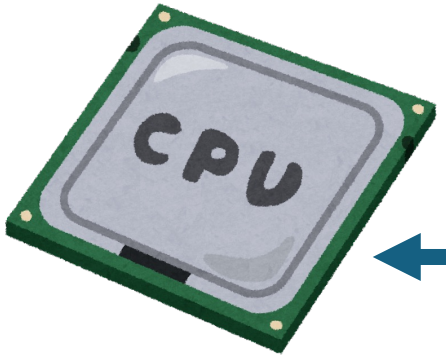


NIC へのアクセス

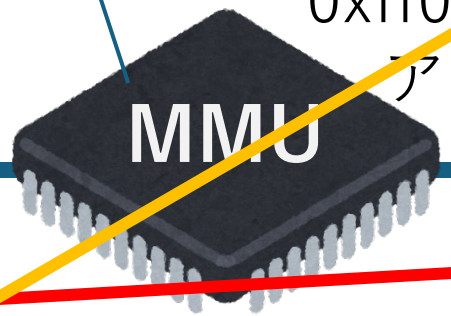
アドレスデコーダという

0xff000000 は
NIC

信号の出力先を振り分ける



物理アドレス
0xff000000 へ
アクセス



ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

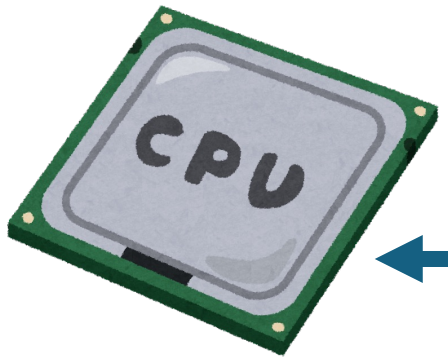
メモリの読み書きを通じたデバイス操作

仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	

基本的にカーネルは一般的なプロセスの
ページテーブルにデバイスアクセス用の
物理アドレスを記載しない
もしくは、記載しても非特権モード時の
アクセスを許可しない設定を適用

NIC
信号の送出先を振り分ける

物理アドレス
0xff000000 へ
アクセス



MMU

Address
Decoder

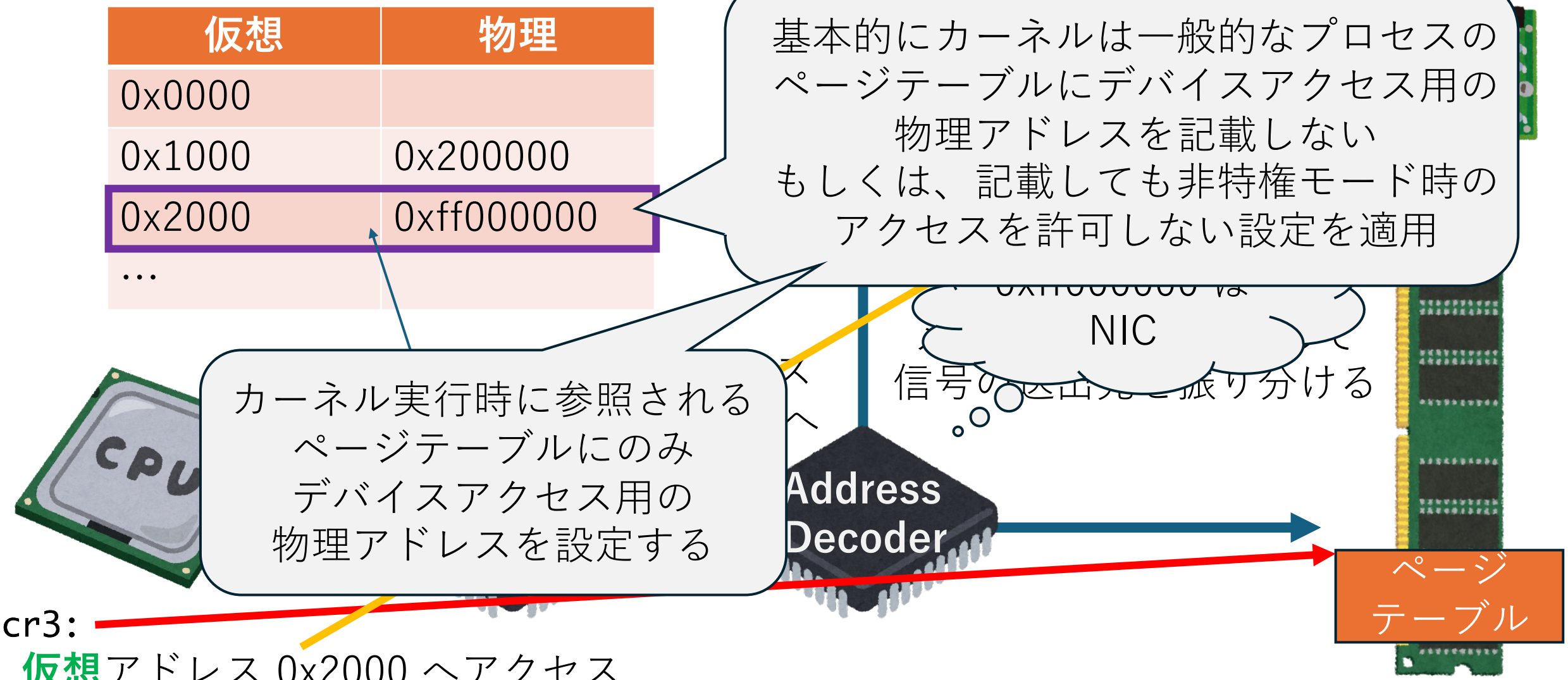
cr3:

仮想アドレス 0x2000 へアクセス

メモリの読み書きを通じたデバイス操作

仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	

基本的にカーネルは一般的なプロセスの
ページテーブルにデバイスアクセス用の
物理アドレスを記載しない
もしくは、記載しても非特権モード時の
アクセスを許可しない設定を適用



カーネル実行時に参照される
ページテーブルにのみ
デバイスアクセス用の
物理アドレスを設定する

cr3:

仮想アドレス 0x2000 へアクセス

メモリの読み書きを通じたデバイス操作

仮想	物理
0x0000	
0x1000	0x200000
0x2000	0xff000000
…	

基本的にカーネルは一般的なプロセスの
ページテーブルにデバイスアクセス用の
物理アドレスを記載しない
もしくは、記載しても非特権モード時の
アクセスを許可しない設定を適用

NIC

カーネル実行時に参照される
ページテーブルにのみ
デバイスアクセス用の
物理アドレスを設定する

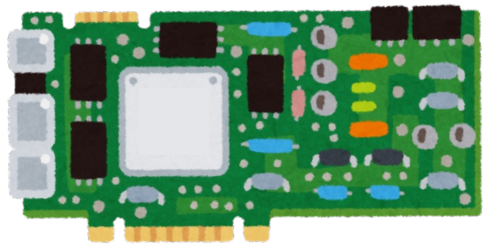
一般的なプロセスは
システムコールを通じて
カーネルにリクエストすることで
代わりにデバイスに
アクセスしてもらう

cr3:

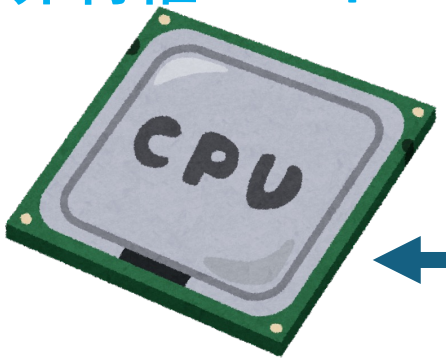
仮想アドレス 0x2000 へアクセス

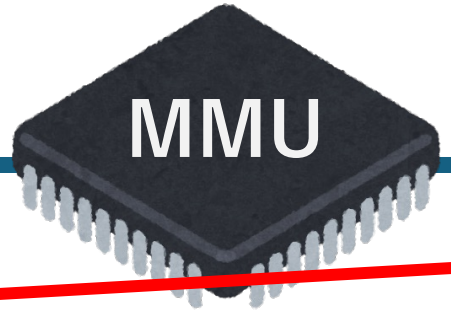
メモリの読み書きを通じたデバイス操作

仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



非特権モード



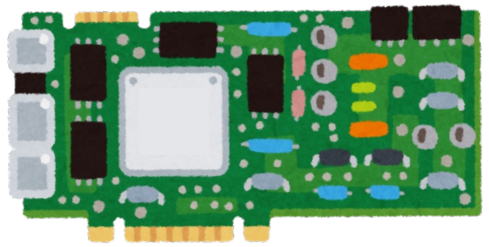


ページ
テーブル

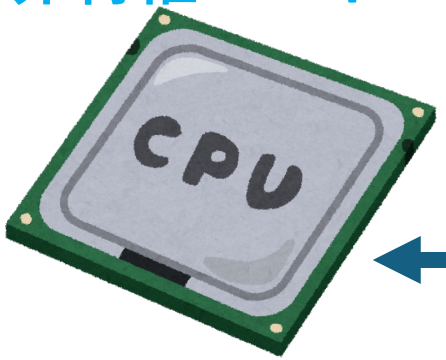
cr3:

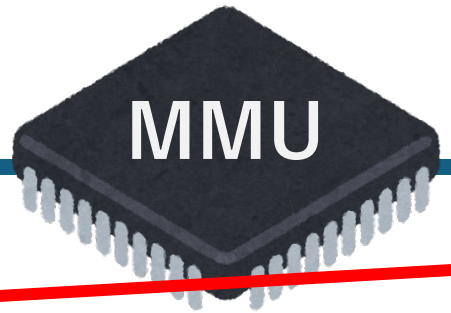
メモリの読み書きを通じたデバイス操作

仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



非特権モード





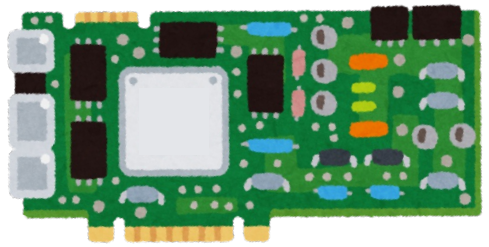
ページ
テーブル

cr3:

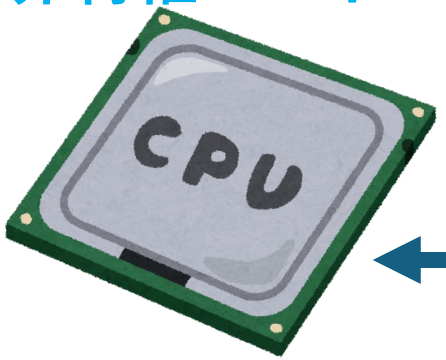
仮想アドレス 0x2000 へアクセス

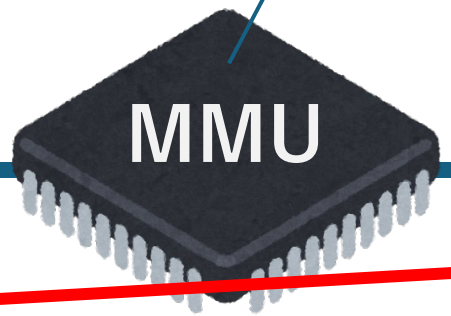
メモリの読み書きを通じたデバイス操作

仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



非特権モード





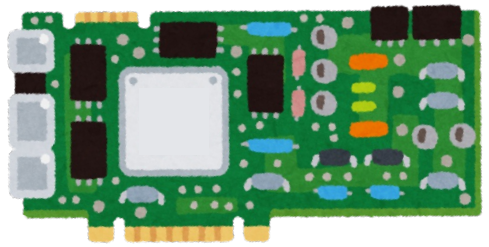
ページ
テーブル

cr3:

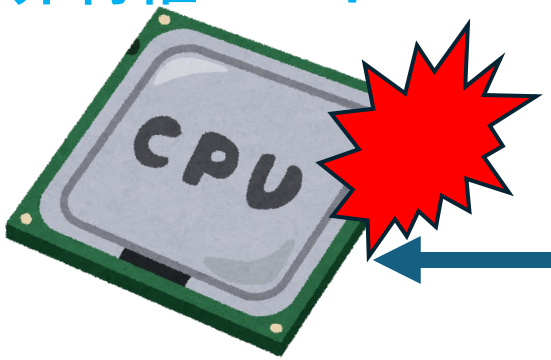
仮想アドレス 0x2000 へアクセス

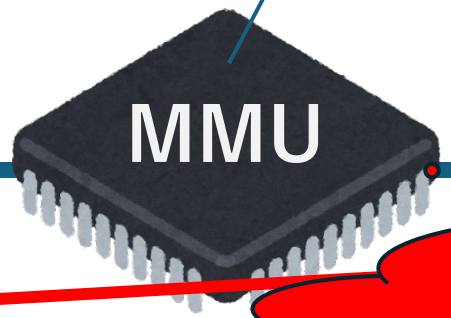
メモリの読み書きを通じたデバイス操作

仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



非特権モード





Segmentation fault を起動

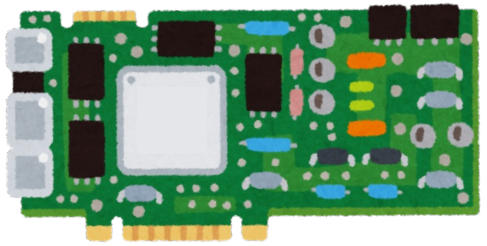
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

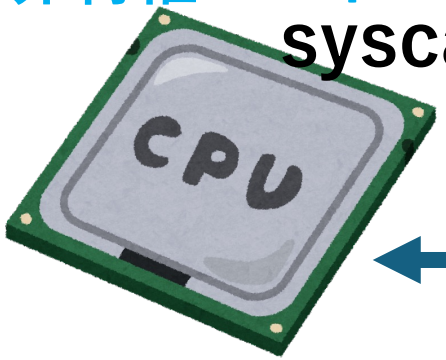
メモリの読み書きを通じたデバイス操作

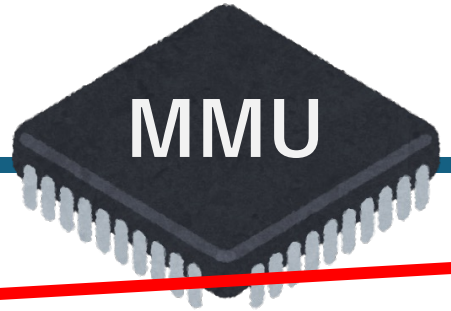
仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



非特権モード

syscall 命令



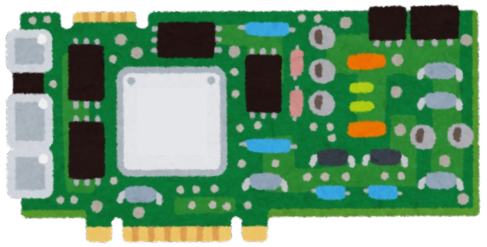


ページ
テーブル

cr3:

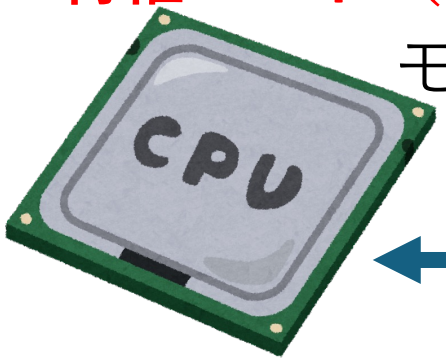
メモリの読み書きを通じたデバイス操作

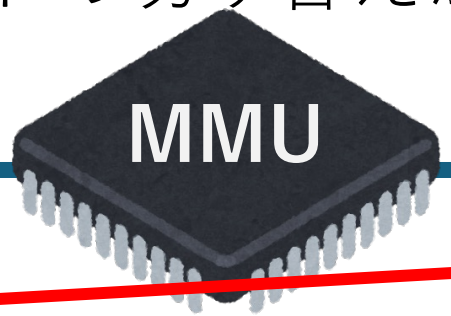
仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



特権モード（カーネル）

モードの切り替えが完了



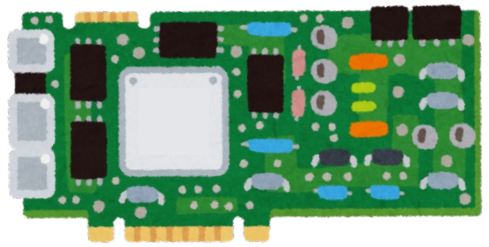


ページ
テーブル

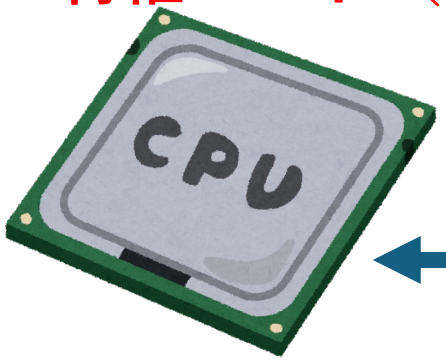
cr3:

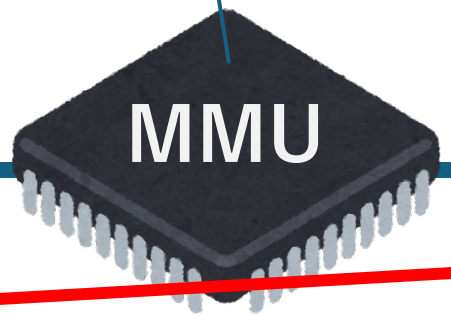
メモリの読み書きを通じたデバイス操作

仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



特権モード（カーネル）





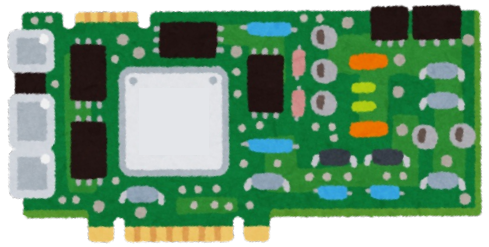
ページ
テーブル

cr3:

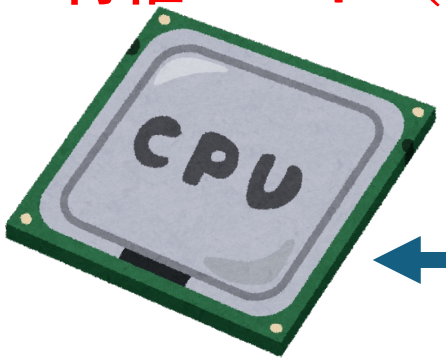
仮想アドレス 0x2000 へアクセス

メモリの読み書きを通じたデバイス操作

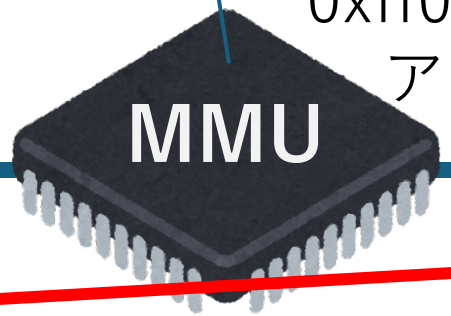
仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



特権モード（カーネル）



物理アドレス
0xff000000 へ
アクセス



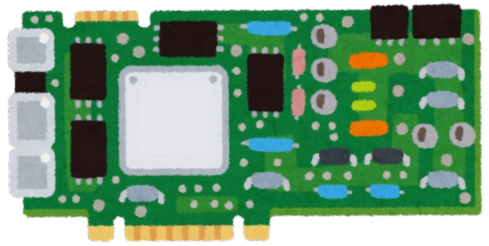
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

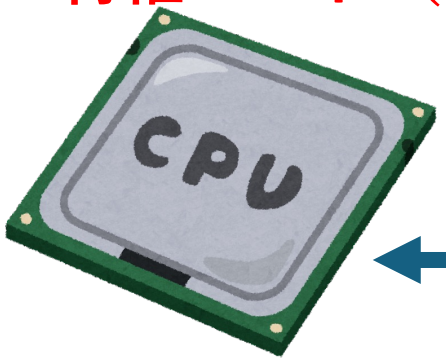
メモリの読み書きを通じたデバイス操作

仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		

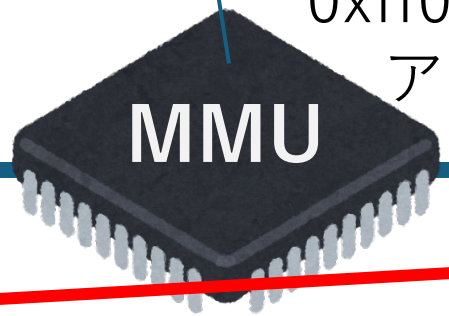


特権モードが
適用されているので
アクセス可能

特権モード（カーネル）



物理アドレス
0xff000000 へ
アクセス



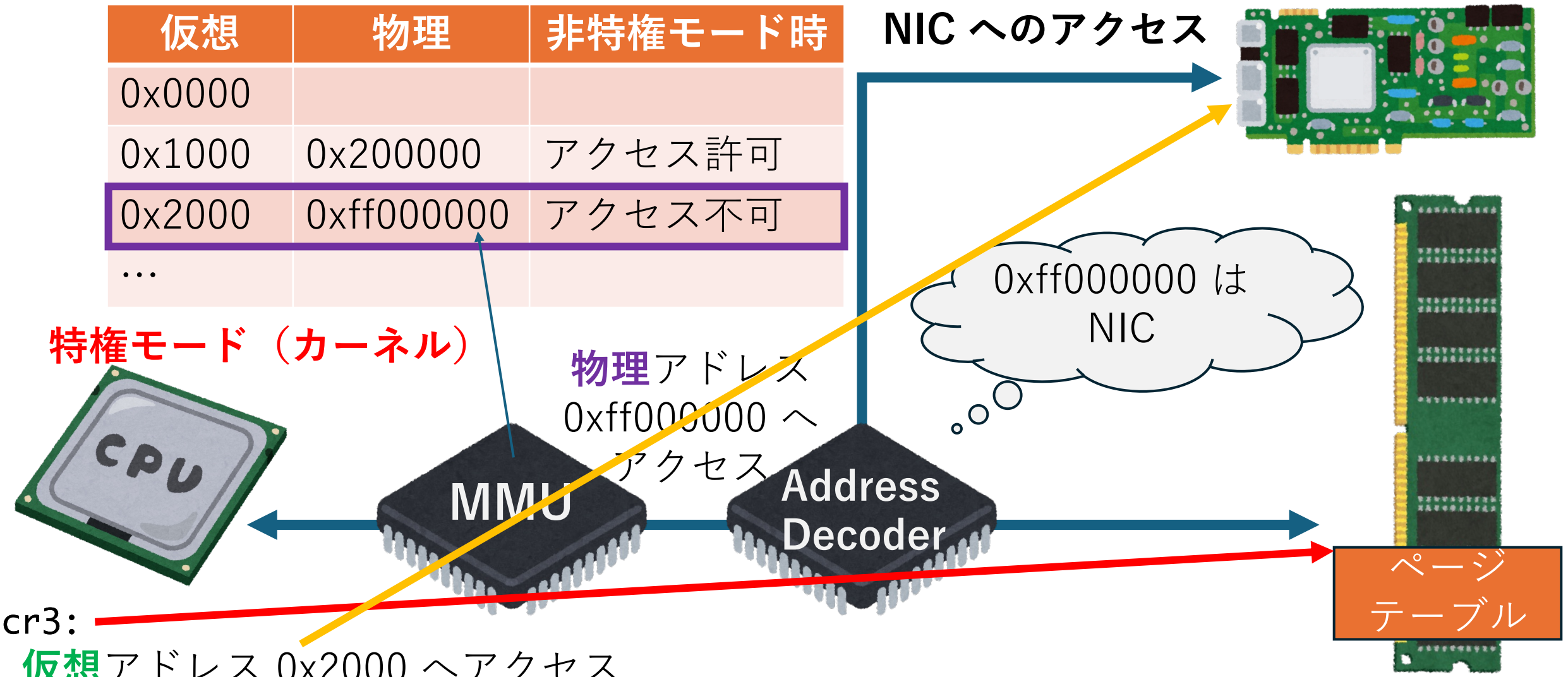
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

メモリの読み書きを通じたデバイス操作

仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

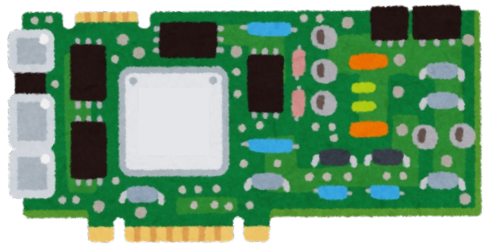
Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

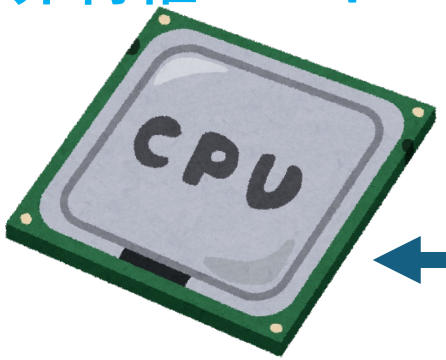
Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

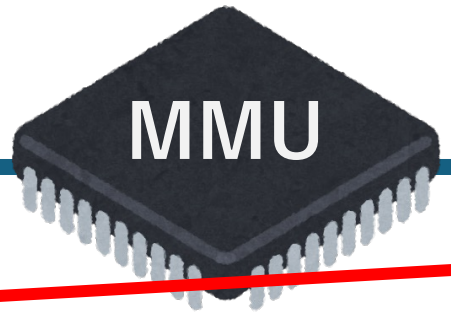
メモリの読み書きを通じたデバイス操作

仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス不可
…		



非特権モード





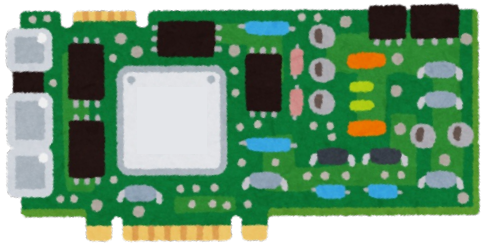
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

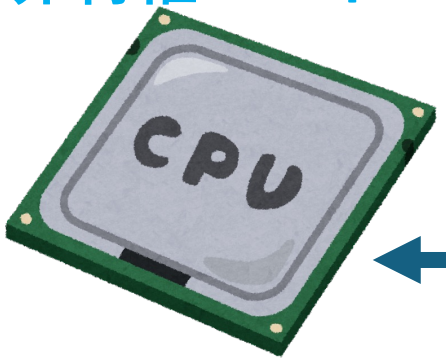
メモリの読み書きを通じたデバイス操作

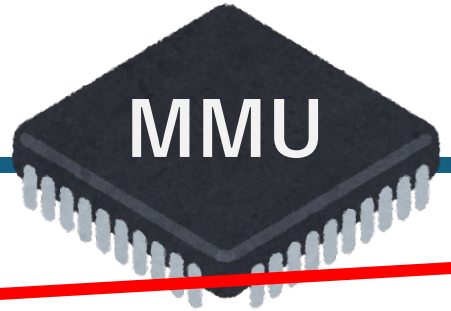
仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス**許可**
…		

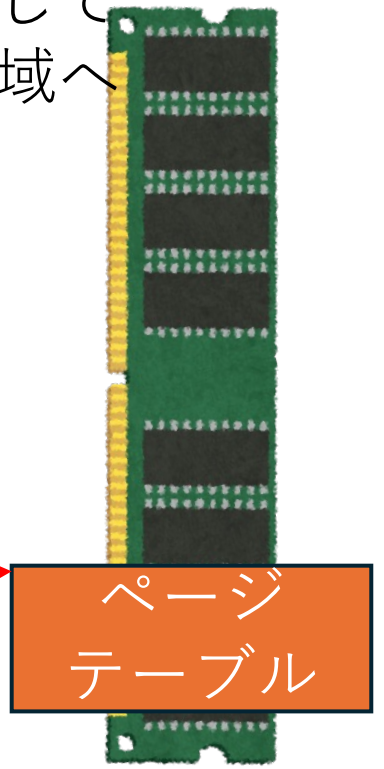


ページテーブル設定を通して
デバイス操作用メモリ領域へ
非特権モードで
アクセス可能にする

非特権モード





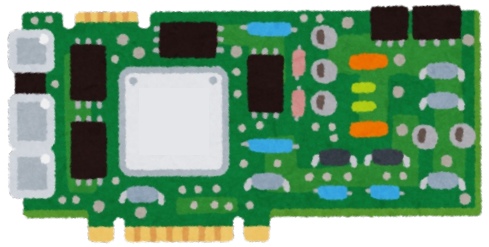


cr3:

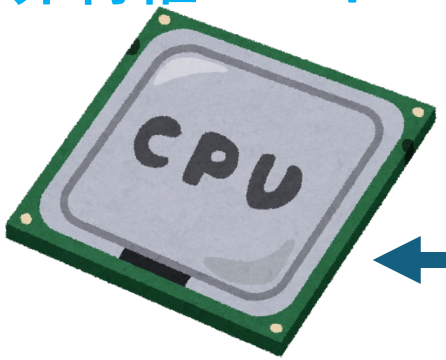
仮想アドレス 0x2000 へアクセス

メモリの読み書きを通じたデバイス操作

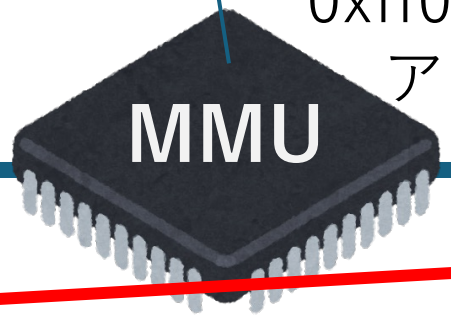
仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス**許可**
…		



非特権モード



物理アドレス
0xff000000 へ
アクセス



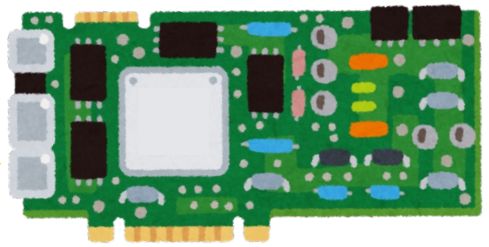
ページ
テーブル

cr3:

仮想アドレス 0x2000 へアクセス

メモリの読み書きを通じたデバイス操作

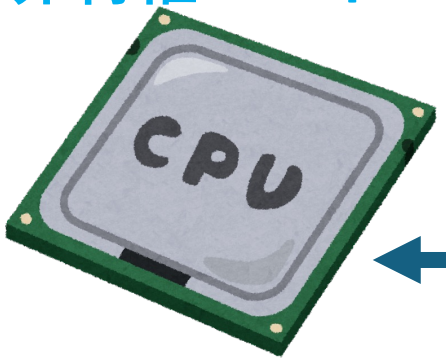
仮想	物理	非特権モード時
0x0000		
0x1000	0x200000	アクセス許可
0x2000	0xff000000	アクセス**許可**
…		



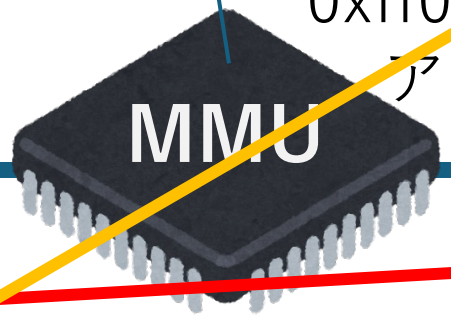
NIC へのアクセス

0xff000000 は NIC

非特権モード



物理アドレス 0xff000000 へアクセス



ページテーブル

cr3:

仮想アドレス 0x2000 へアクセス

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. デバイスアクセスのための処理と
I/O デバイスの I/O 対象データ

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

カーネルバイパスとは？

Q. これは何？
A. ユーザー空間で
動作するプログラム

Q. 何がカーネルを中継しなくなる？
A. 実装次第

Q. デバイスへの
アクセスとは
具体的に何？
A. 基本的には
メモリの読み書き

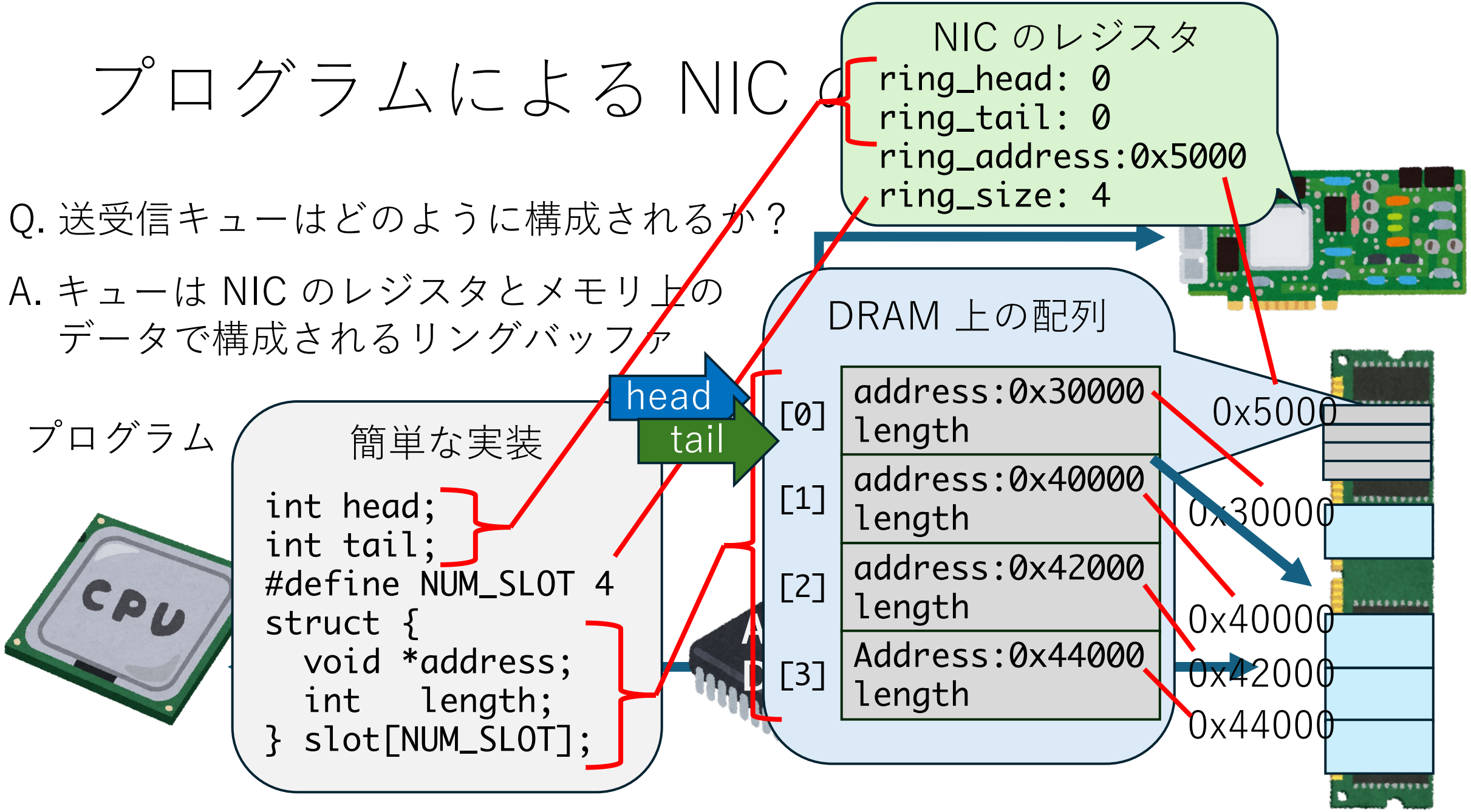
Q. 何故通常はカーネルを経由する？
A. ユーザー空間プログラムは通常では
デバイス操作用のメモリ領域への
アクセスが許可されていないから

Q. どうやってバイパスする？
A. ユーザー空間プログラムへデバイス操作用
メモリ領域へのアクセスを許可する

プログラムによる NIC の

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ



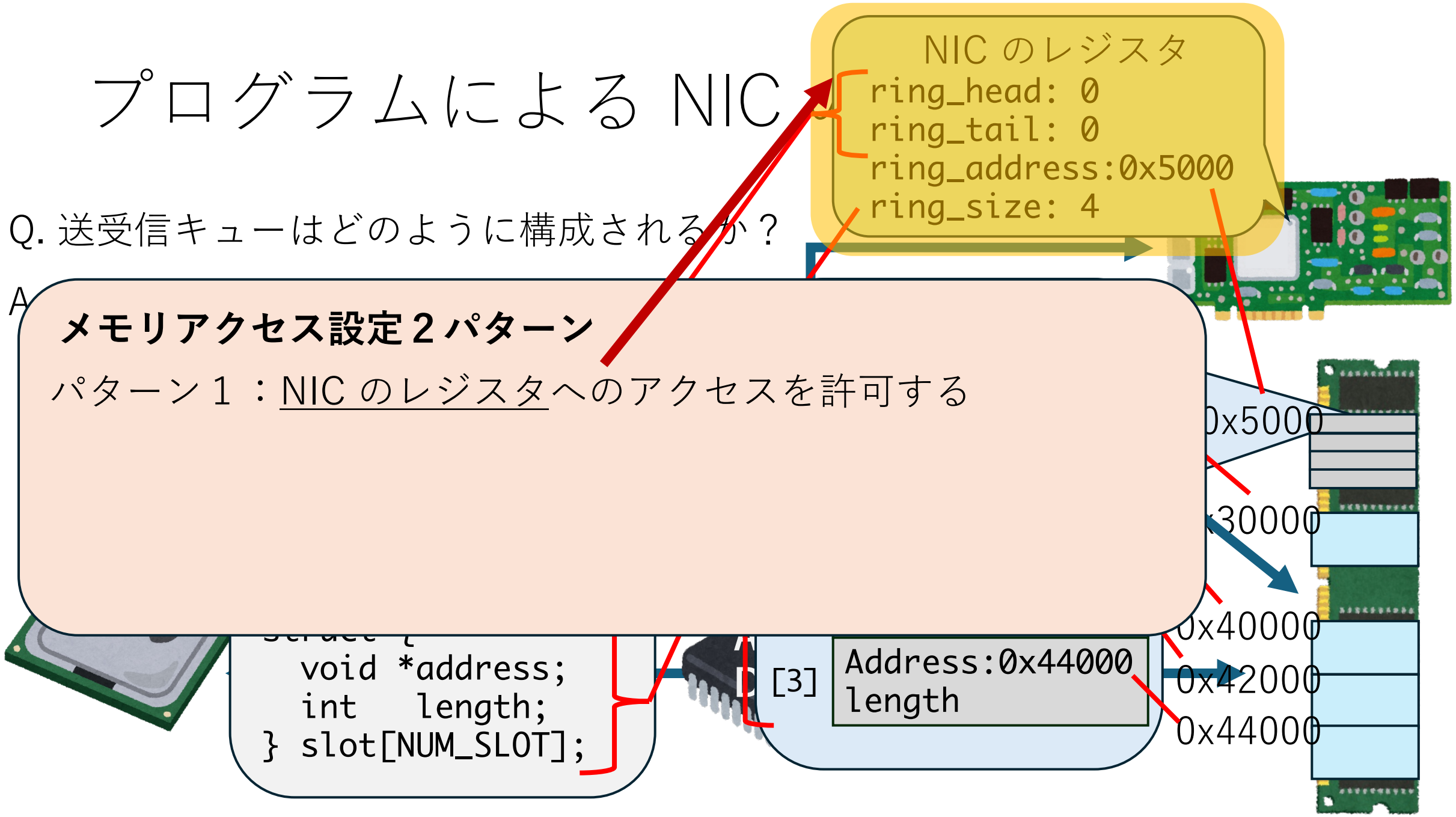
プログラムによる NIC

Q. 送受信キューはどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する



プログラムによる NIC

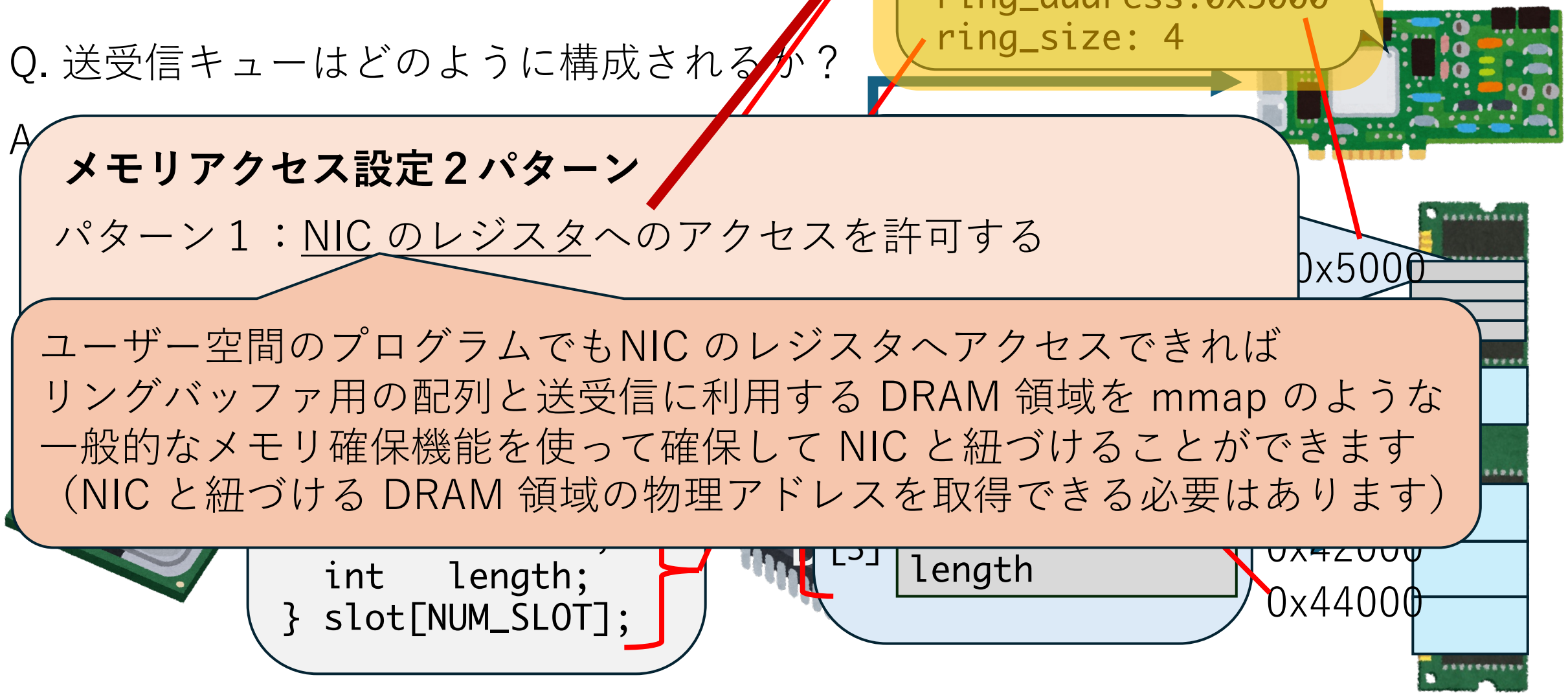
Q. 送受信キューはどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する

ユーザー空間のプログラムでもNIC のレジスタへアクセスできれば
リングバッファ用の配列と送受信に利用する DRAM 領域を mmap のような
一般的なメモリ確保機能を使って確保して NIC と紐づけることができます
（NIC と紐づける DRAM 領域の物理アドレスを取得できる必要はあります）



プログラムによる NIC

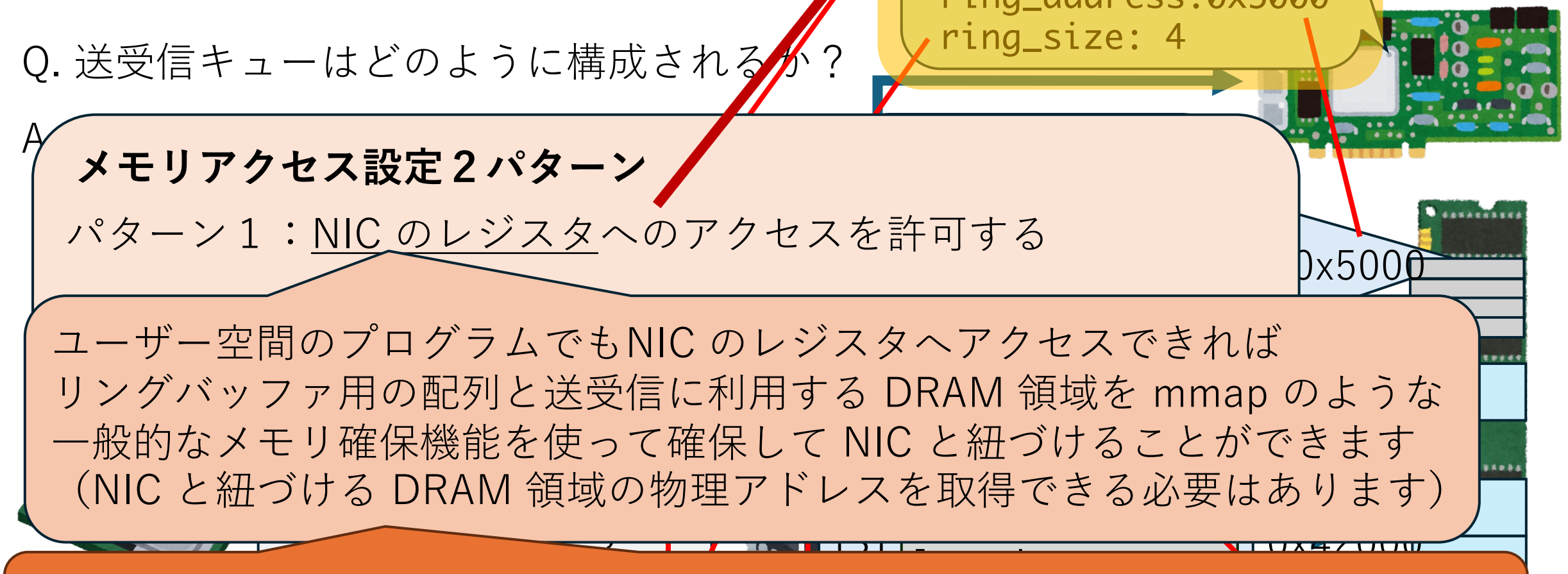
Q. 送受信キューはどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する

NIC のレジスタ
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4



ユーザー空間のプログラムでもNIC のレジスタへアクセスできれば
リングバッファ用の配列と送受信に利用する DRAM 領域を mmap のような一般的なメモリ確保機能を使って確保して NIC と紐づけることができます
（NIC と紐づける DRAM 領域の物理アドレスを取得できる必要はあります）

この場合はユーザー空間で各 NIC のハードウェア仕様に対応した操作が必要となるため、ユーザー空間でデバイスドライバを動作させる必要があります

プログラムによる NIC の

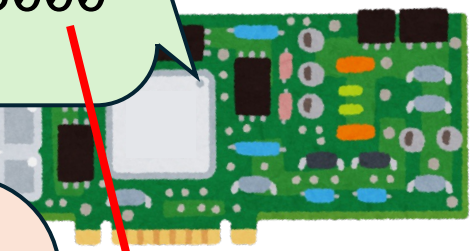
Q. 送受信キューはどのように構成されるか？

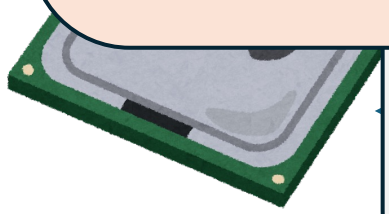
A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する
パターン２：NIC に紐付けした DRAM 領域だけアクセス許可し
アプリが NIC の送受信の起動のために利用可能な
専用システムコールをカーネルが提供する

NIC のレジスタ
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4





```
  void *address;
  int   length;
} slot[NUM_SLOT];
```

[3] Address:0x44000
length

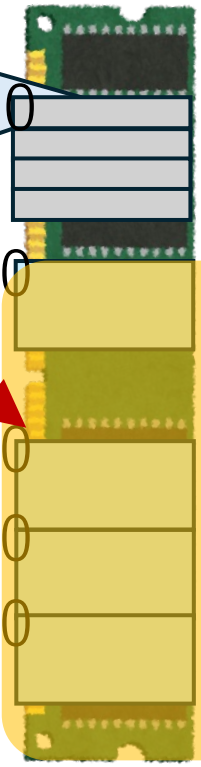
0x5000

0x30000

0x40000

0x42000

0x44000



プログラムによる NIC の

Q. 送受信キューはどのように構成されるか？

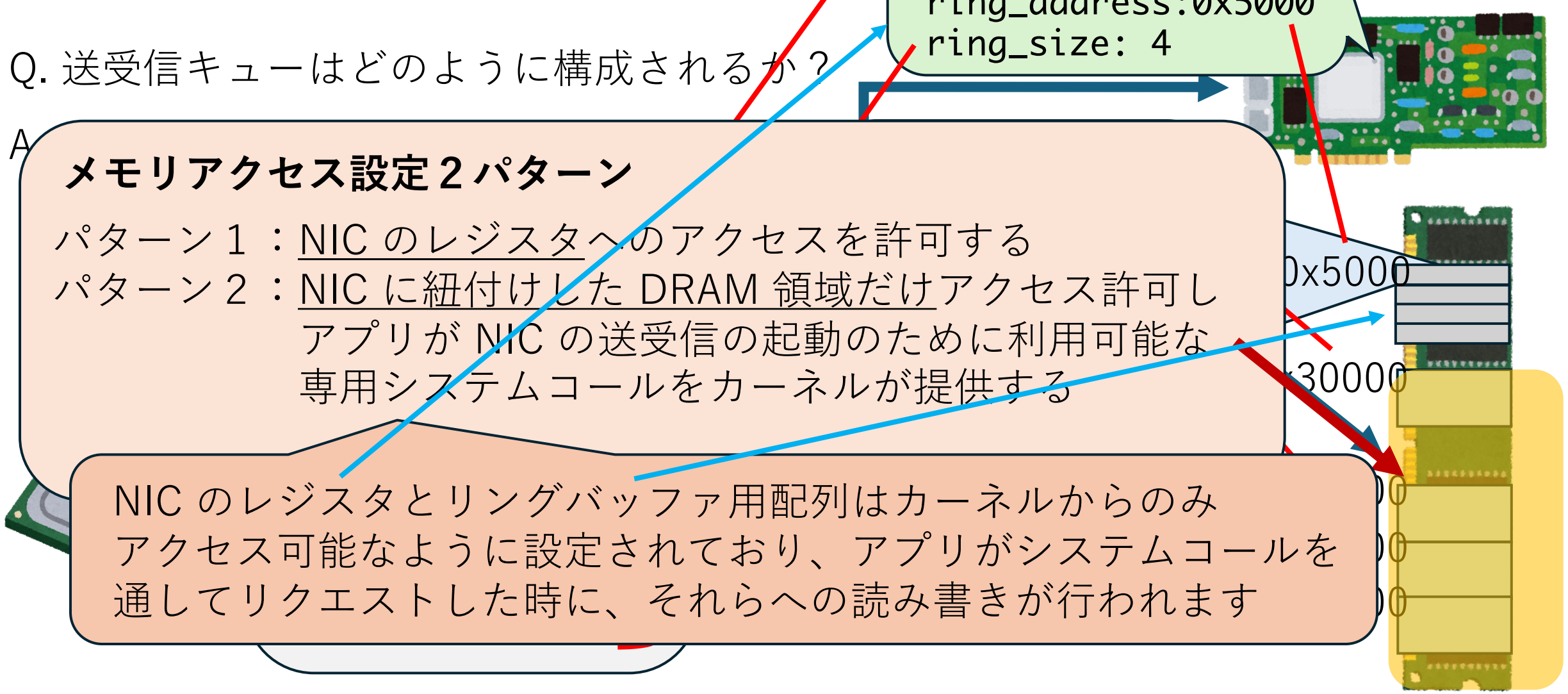
A

NIC のレジスタ
```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する
パターン２：NIC に紐付けした DRAM 領域だけアクセス許可し
アプリが NIC の送受信の起動のために利用可能な
専用システムコールをカーネルが提供する

NIC のレジスタとリングバッファ用配列はカーネルからのみ
アクセス可能なように設定されており、アプリがシステムコールを
通してリクエストした時に、それらへの読み書きが行われます



プログラムによる NIC の

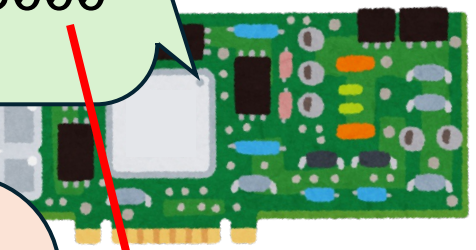
Q. 送受信キューはどのように構成されるか？

A

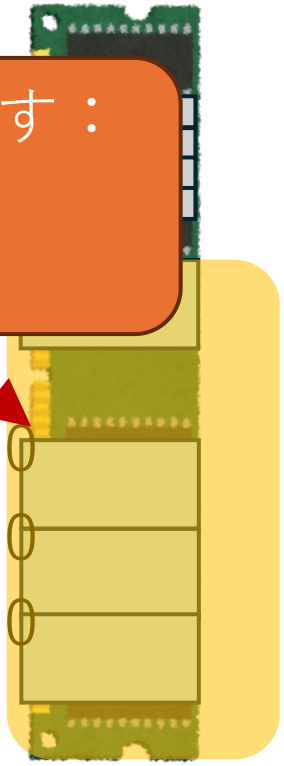
メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する

NIC のレジスタ
```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



この場合は、NIC ごとのハードウェア仕様の差異はカーネル内で対応します：
カーネルに実装された各 NIC 用のドライバが利用されるため
ユーザー空間でデバイスドライバを実行する必要はありません



NIC のレジスタとリングバッファ用配列はカーネルからのみ
アクセス可能なように設定されており、アプリがシステムコールを
通してリクエストした時に、それらへの読み書きが行われます

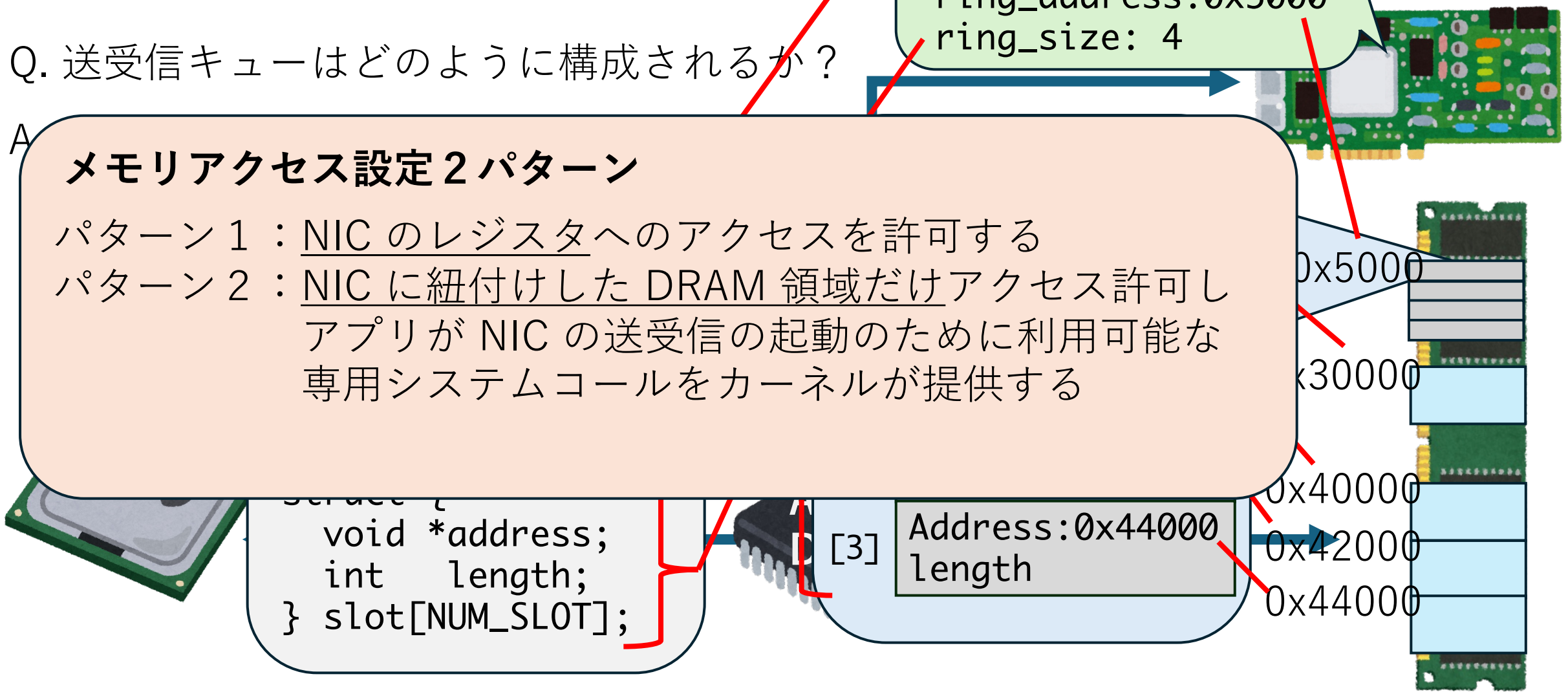
プログラムによる NIC の

Q. 送受信キューはどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する
パターン２：NIC に紐付けした DRAM 領域だけアクセス許可し
アプリが NIC の送受信の起動のために利用可能な
専用システムコールをカーネルが提供する



プログラムによる NIC の

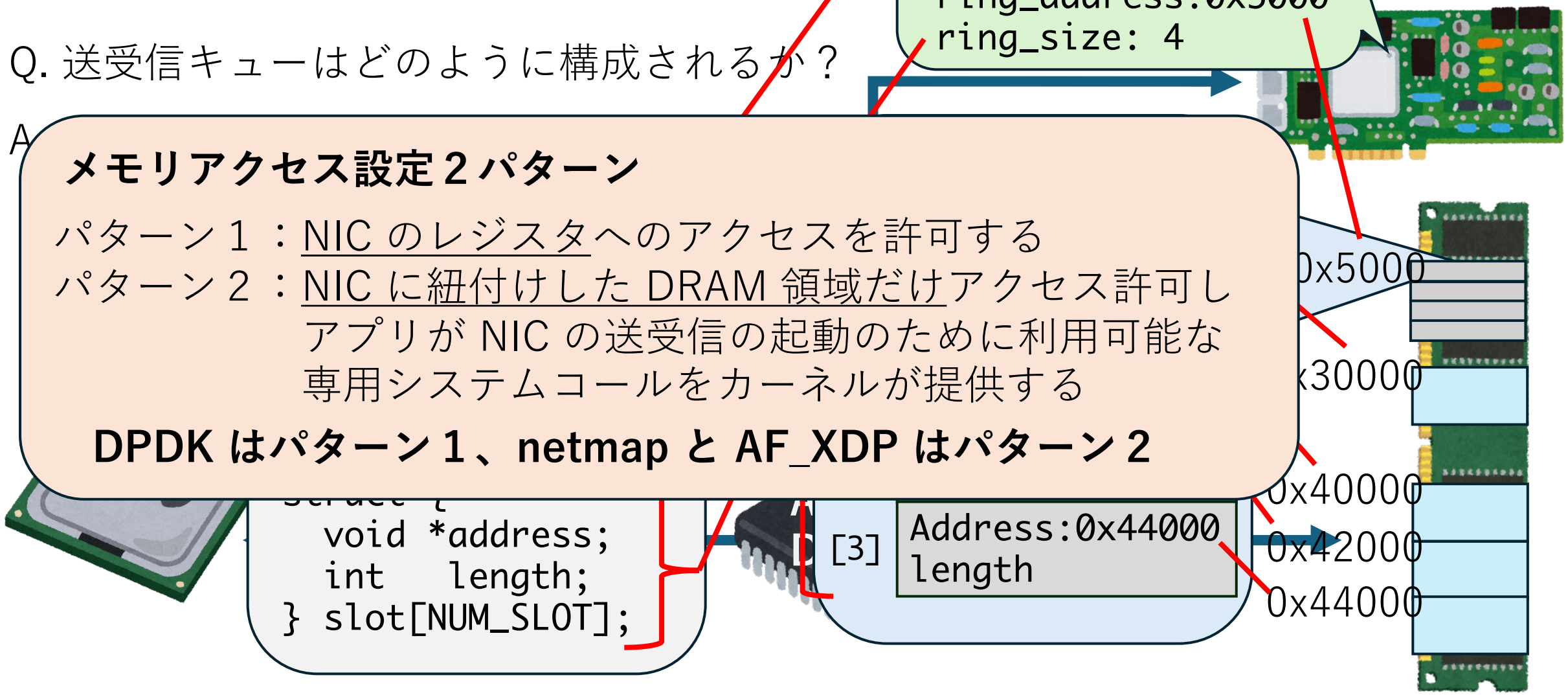
Q. 送受信キュー はどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する
パターン２：NIC に紐付けした DRAM 領域だけアクセス許可し
アプリが NIC の送受信の起動のために利用可能な
専用システムコールをカーネルが提供する

DPDK はパターン１、netmap と AF_XDP はパターン２



簡単にまとめると

- CPU 命令にはメモリの読み書き機能を提供するものがある
- DRAM と NIC は CPU のメモリ読み書き命令を通して操作可能
- カーネルはユーザー空間プログラムがアクセス可能な物理アドレスを制限でき、同じ仕組みで NIC へのアクセスを制限できる
- カーネルバイパス構成は、ユーザー空間プログラムへ NIC のレジスタへのアクセス 、もしくは NIC と紐付けられた送受信データ配置用の DRAM 領域へのアクセスを許可する

簡単にまとめると

- CPU 命令にはメモリの読み書き機能を提供するものがある
- DRAM と NIC は CPU のメモリ読み書き命令を通して操作可能
- カーネルはユーザー空間プログラムがアクセス可能な物理アドレスを制限でき、同じ仕組みで NIC へのアクセスを制限できる
- カーネルバイパス構成は、ユーザー空間プログラムへ NIC のレジスタへのアクセス 、もしくは NIC と紐付けられた送受信データ配置用の DRAM 領域へのアクセスを許可する

プログラムによるメモリアクセス

例：メモリアドレス 0x100000000 に 0x12345678 を書き込むプログラム

```
int main(void)
{
  *((int *) 0x100000000) = 0x12345678;
}
```

gcc -O0 program.c

コンパイル

a.out

ディスアセンブル

objdump -d ./a.out

```
0000000000001129 <main>:
        1131:       48 b8 00 00 00 00 01
        1138:       00 00 00
        113b:       c7 00 78 56 34 12
        1141:       00
```

```
endbr64
push   %rbp
mov    %rsp,%rbp
movabs $0x100000000,%rax

movl   $0x12345678,(%rax)
mov    $0x0,%eax
pop    %rbp
retq
nopl   0x0(%rax,%rax,1)
```

rax レジスタに 0x100000000 を設定する

rax レジスタの値により参照される
メモリアドレス（0x100000000）へ
0x12345678 を書き込む

簡単にまとめると

- CPU 命令にはメモリの読み書き機能を提供するものがある
- DRAM と NIC は CPU のメモリ読み書き命令を通して操作可能
- カーネルはユーザー空間プログラムがアクセス可能な物理アドレスを制限でき、同じ仕組みで NIC へのアクセスを制限できる
- カーネルバイパス構成は、ユーザー空間プログラムへ NIC のレジスタへのアクセス 、もしくは NIC と紐付けられた送受信データ配置用の DRAM 領域へのアクセスを許可する

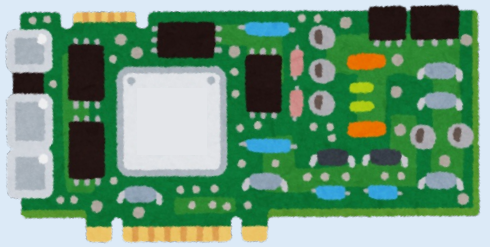
簡単にまとめると

- CPU 命令にはメモリの読み書き機能を提供するものがある
- DRAM と NIC は CPU のメモリ読み書き命令を通して操作可能
- カーネルはユーザー空間プログラムがアクセス可能な物理アドレスを制限でき、同じ仕組みで NIC へのアクセスを制限できる
- カーネルバイパス構成は、ユーザー空間プログラムへ NIC のレジスタへのアクセス 、もしくは NIC と紐付けられた送受信データ配置用の DRAM 領域へのアクセスを許可する

プログラムによる NIC の取り扱い

Q. 送受信キューはどのように構成されるか？

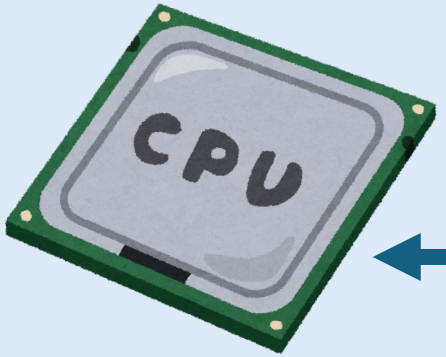
A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

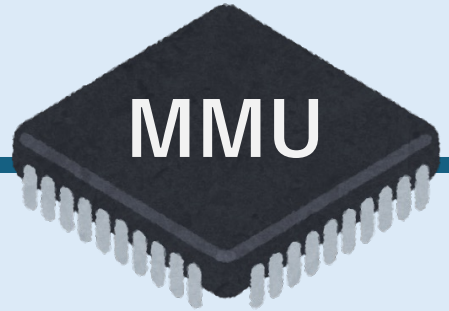


プログラム

**プログラムから
NIC のレジスタと DRAM は
MMU / Address Decoder を
通じてアクセス可能**

アドレスデコーダという
ハードウェアが
メモリアドレスに応じて
信号の送出先を振り分ける





（どのアドレスが NIC のどのレジスタに対応するかは NIC の仕様に依存）

簡単にまとめると

- CPU 命令にはメモリの読み書き機能を提供するものがある
- DRAM と NIC は CPU のメモリ読み書き命令を通して操作可能
- カーネルはユーザー空間プログラムがアクセス可能な物理アドレスを制限でき、同じ仕組みで NIC へのアクセスを制限できる
- カーネルバイパス構成は、ユーザー空間プログラムへ NIC のレジスタへのアクセス 、もしくは NIC と紐付けられた送受信データ配置用の DRAM 領域へのアクセスを許可する

簡単にまとめると

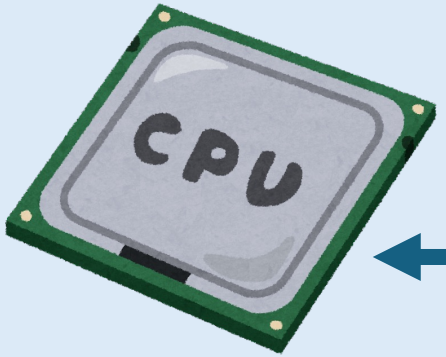
- CPU 命令にはメモリの読み書き機能を提供するものがある
- DRAM と NIC は CPU のメモリ読み書き命令を通して操作可能
- カーネルはユーザー空間プログラムがアクセス可能な物理アドレスを制限でき、同じ仕組みで NIC へのアクセスを制限できる
- カーネルバイパス構成は、ユーザー空間プログラムへ NIC のレジスタへのアクセス 、もしくは NIC と紐付けられた送受信データ配置用の DRAM 領域へのアクセスを許可する

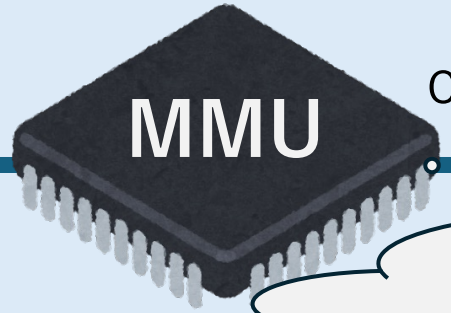
プログラムによるメモリアクセス

対応する**物理アドレス**として
0x2000 が設定されていた場合

仮想アドレス 0x100000000 へ
0x12345678 を書き込み

```
movl $0x12345678,(%rax)
```

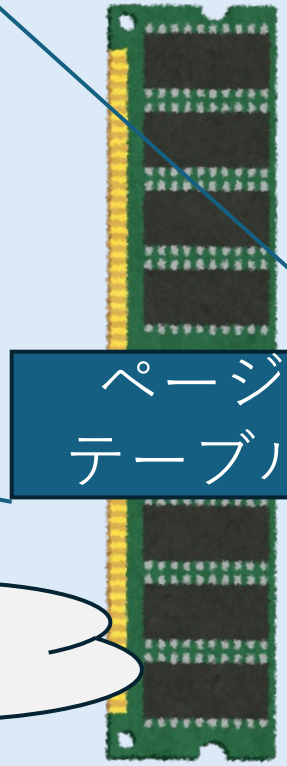




仮想	物理
0x0000	
…	
0x100000000	0x2000
…	

物理アドレス 0x2000 へ
0x12345678 を書き込み

MMU はアドレス変換に際して
cr3 で示されるページテーブルを参照



ページ
テーブル

仮想アドレス 0x100000000 は
物理アドレスの 0x2000

```
rax:0x100000000
```
cr3:ページテーブルの物理アドレス

簡単にまとめると

- CPU 命令にはメモリの読み書き機能を提供するものがある
- DRAM と NIC は CPU のメモリ読み書き命令を通して操作可能
- カーネルはユーザー空間プログラムがアクセス可能な物理アドレスを制限でき、同じ仕組みで NIC へのアクセスを制限できる
- カーネルバイパス構成は、ユーザー空間プログラムへ NIC のレジスタへのアクセス 、もしくは NIC と紐付けられた送受信データ配置用の DRAM 領域へのアクセスを許可する

簡単にまとめると

- CPU 命令にはメモリの読み書き機能を提供するものがある
- DRAM と NIC は CPU のメモリ読み書き命令を通して操作可能
- カーネルはユーザー空間プログラムがアクセス可能な物理アドレスを制限でき、同じ仕組みで NIC へのアクセスを制限できる
- カーネルバイパス構成は、ユーザー空間プログラムへ NIC のレジスタへのアクセス 、もしくは NIC と紐付けられた送受信データ配置用の DRAM 領域へのアクセスを許可する

プログラムによる NIC の

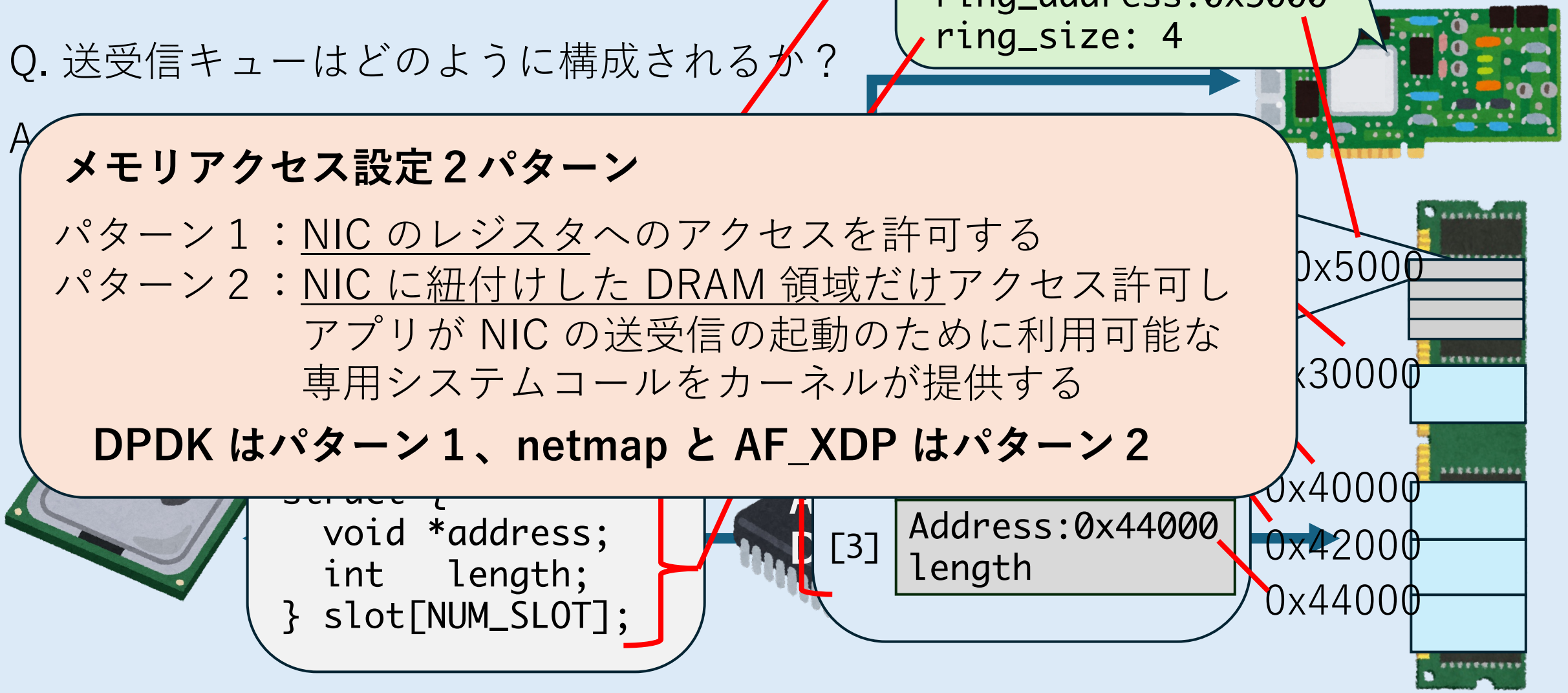
Q. 送受信キューはどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する
パターン２：NIC に紐付けした DRAM 領域だけアクセス許可し
アプリが NIC の送受信の起動のために利用可能な
専用システムコールをカーネルが提供する

DPDK はパターン１、netmap と AF_XDP はパターン２



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

- 一般的な（おそらく暗黙の）想定
  - ハードウェアに脆弱性および悪意はない
  - カーネルに脆弱性はなく悪意のあるコードは実行されない
  - ユーザー空間で動作するプログラムの悪意や脆弱性の可能性は想定するものの、操作性の観点から防御策は現実的な範囲に限定する

CPU のモードと基本的な運用方針

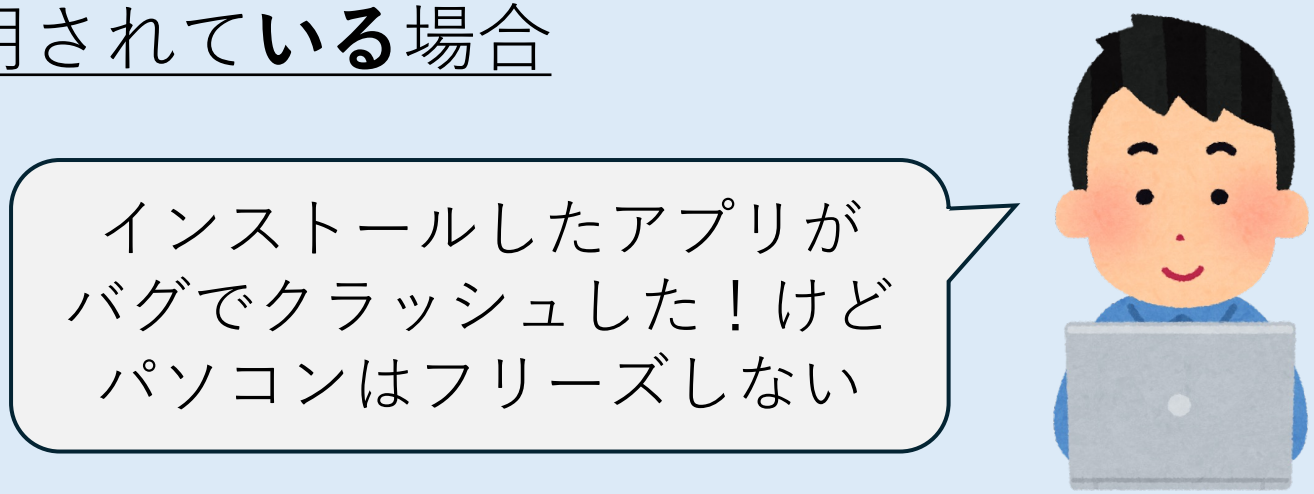
- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

この方針が適用されて**いる**場合

アプリのクラッシュの影響はパソコン全体には波及しない



CPU のモードと基本的な運用方針

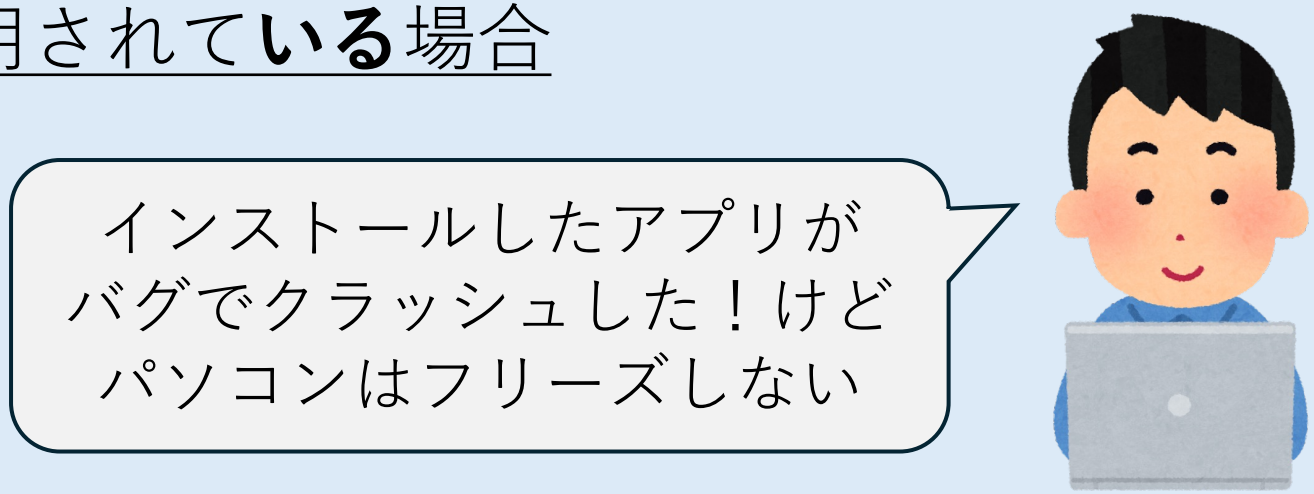
- CPU のモードは大まかに分けて二種類
  1. **特権モード**：大体の CPU 機能を使える（システムの破壊が容易）
  2. **非特権モード**：できることが制限されている（破壊操作が難しい）

特権モードはカーネルモード、非特権モードはユーザーモードと呼ばれることもあります

- 基本的な運用方針：**なるべく破壊操作が難しいことを目指す**

この方針が適用されて**いる**場合

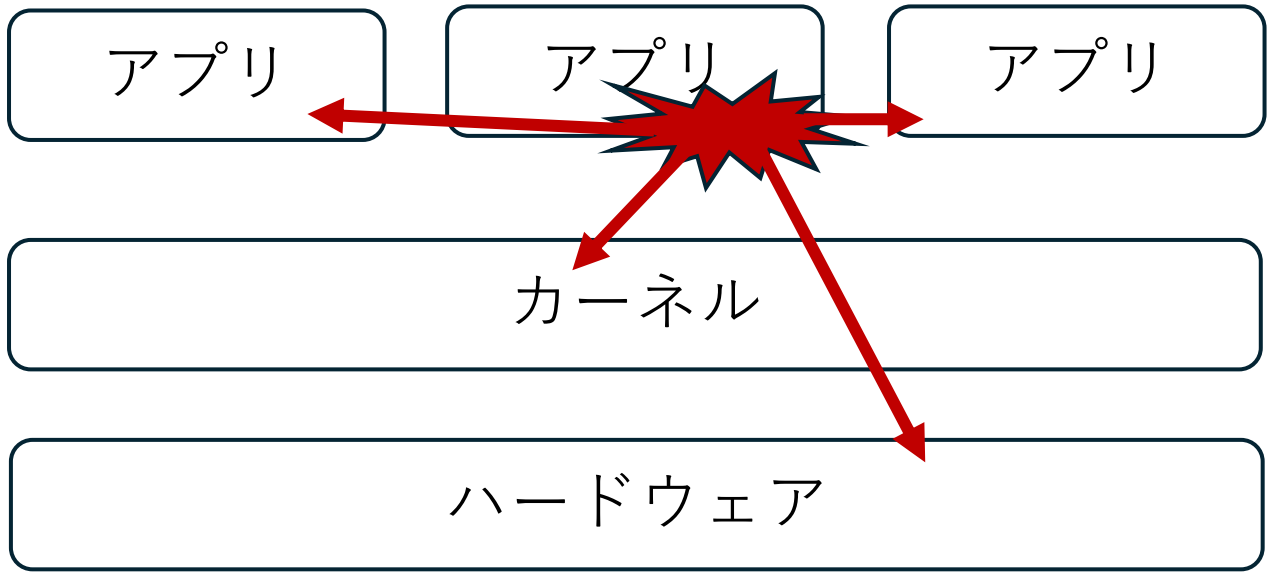
アプリのクラッシュの影響はパソコン全体には波及しない



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

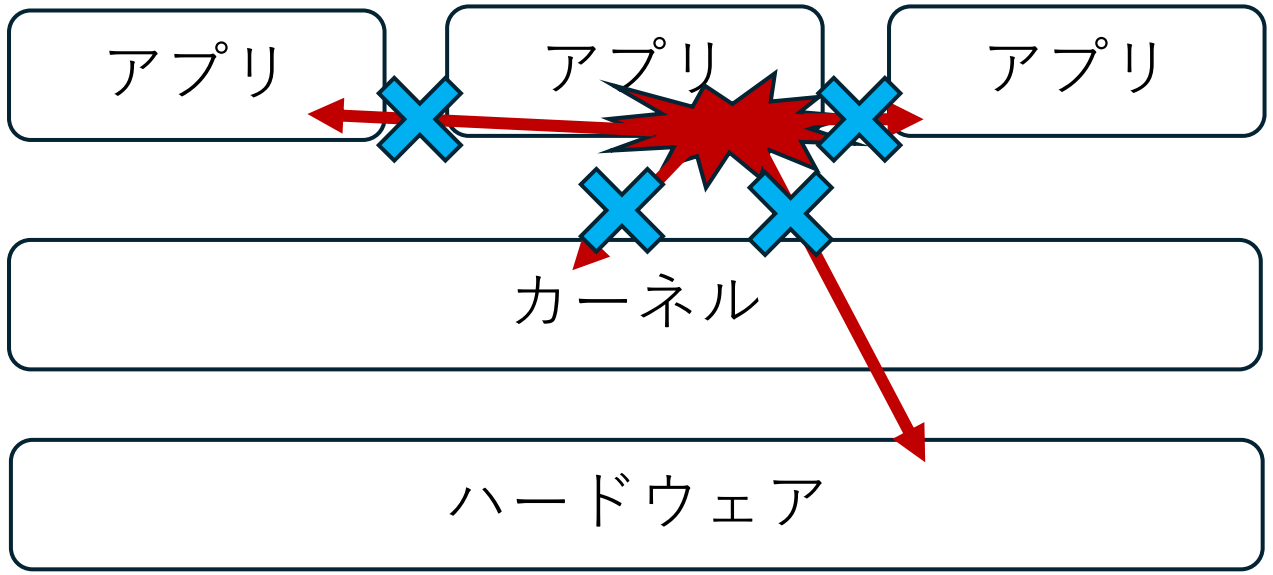
- 基本的な目標：一つのアプリのバグや脆弱性の影響がハードウェア、カーネル、別のアプリへ波及しにくくする
- カーネルバイパス構成を適用すると何が変わる？



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

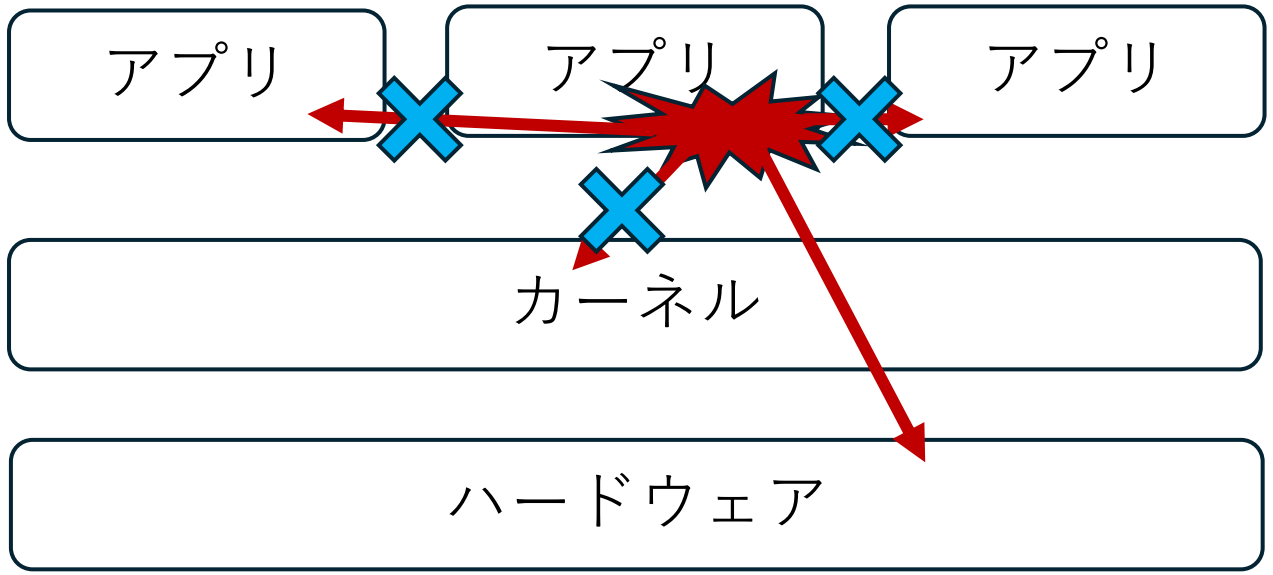
- 基本的な目標：一つのアプリのバグや脆弱性の影響がハードウェア、カーネル、別のアプリへ波及しにくくする



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

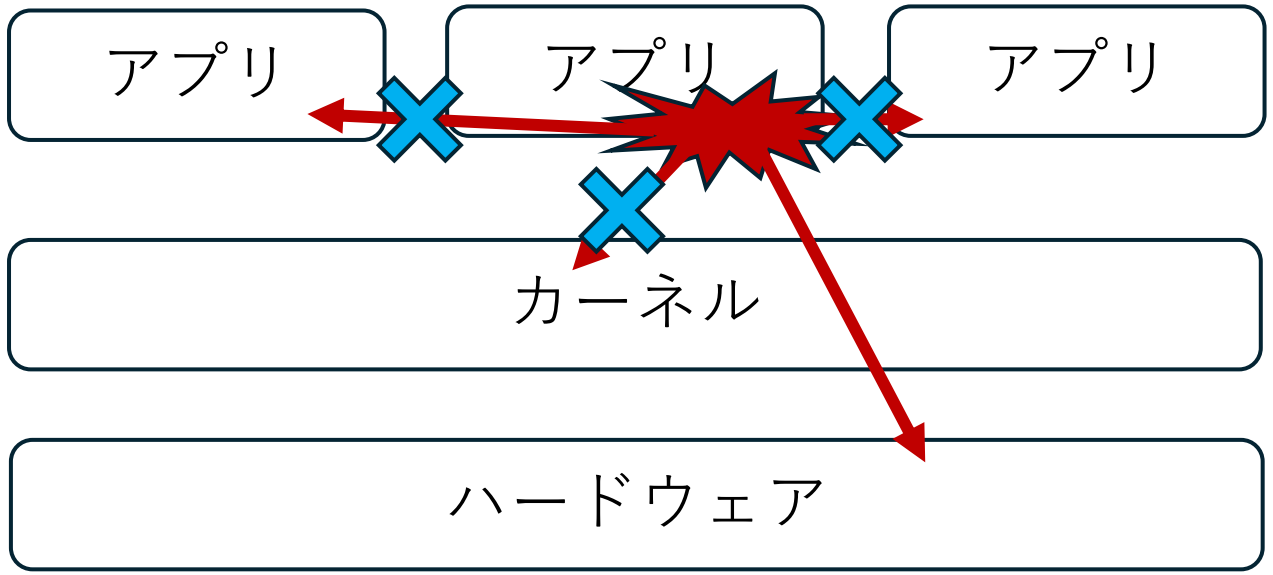
- 基本的な目標：一つのアプリのバグや脆弱性の影響がハードウェア、カーネル、別のアプリへ波及しにくくする
- カーネルバイパス構成を適用すると何が変わる？



セキュリティについての考え方

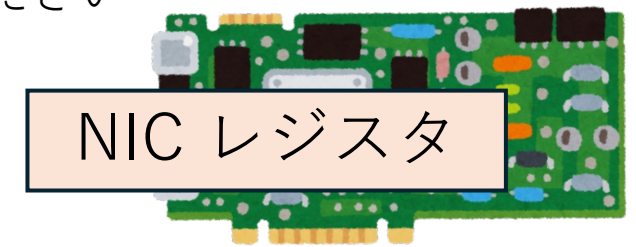
これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

- 基本的な目標：一つのアプリのバグや脆弱性の影響がハードウェア、カーネル、別のアプリへ波及しにくくする
- **影響の波及を防ぐポイント：そもそも複数の要素が同一のリソースへアクセスしないようにする**



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください



プロセス１

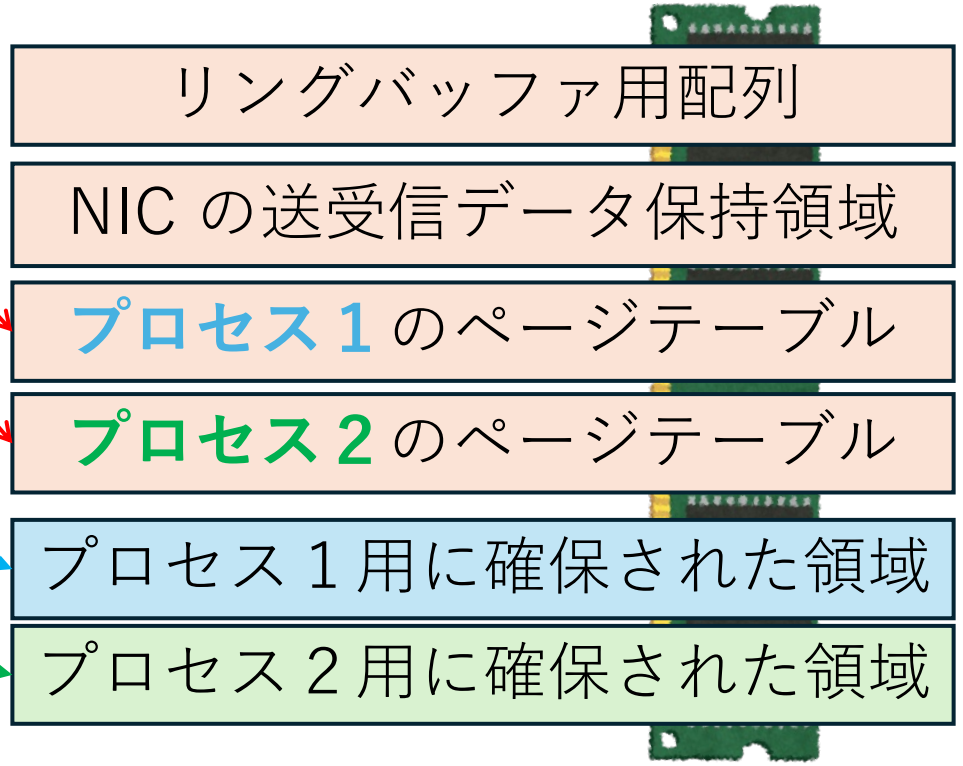
プロセス２

カーネル

設定

アクセス可能

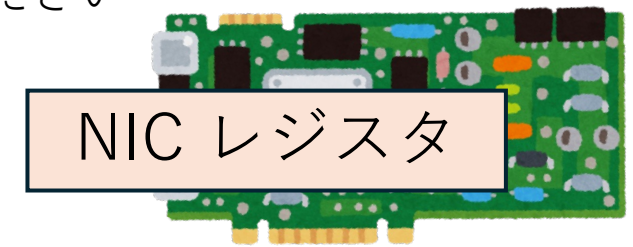
アクセス可能



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

複数の要素が同一のリソースへアクセスしないようにする



プロセス１

プロセス２

カーネル

設定

アクセス可能

アクセス可能

リングバッファ用配列

NIC の送受信データ保持領域

プロセス１のページテーブル

プロセス２のページテーブル

プロセス１用に確保された領域

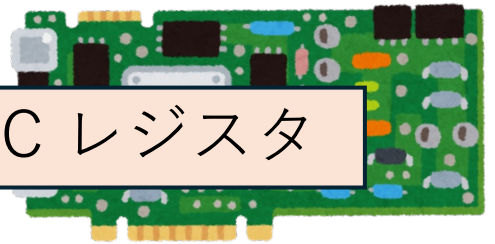
プロセス２用に確保された領域

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

複数の要素が同一のリソースへアクセスしないようにする

一般的な構成ではカーネルしか
NIC のレジスタにアクセスしない



NIC レジスタ

プロセス１

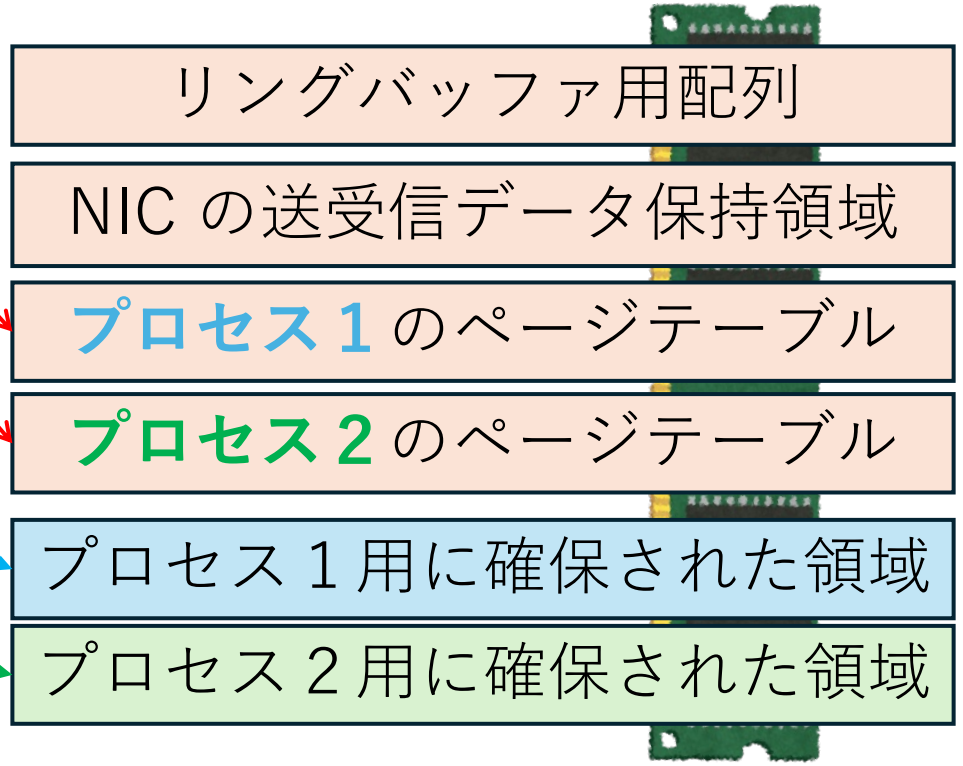
プロセス２

カーネル

設定

アクセス可能

アクセス可能



リングバッファ用配列

NIC の送受信データ保持領域

プロセス１のページテーブル

プロセス２のページテーブル

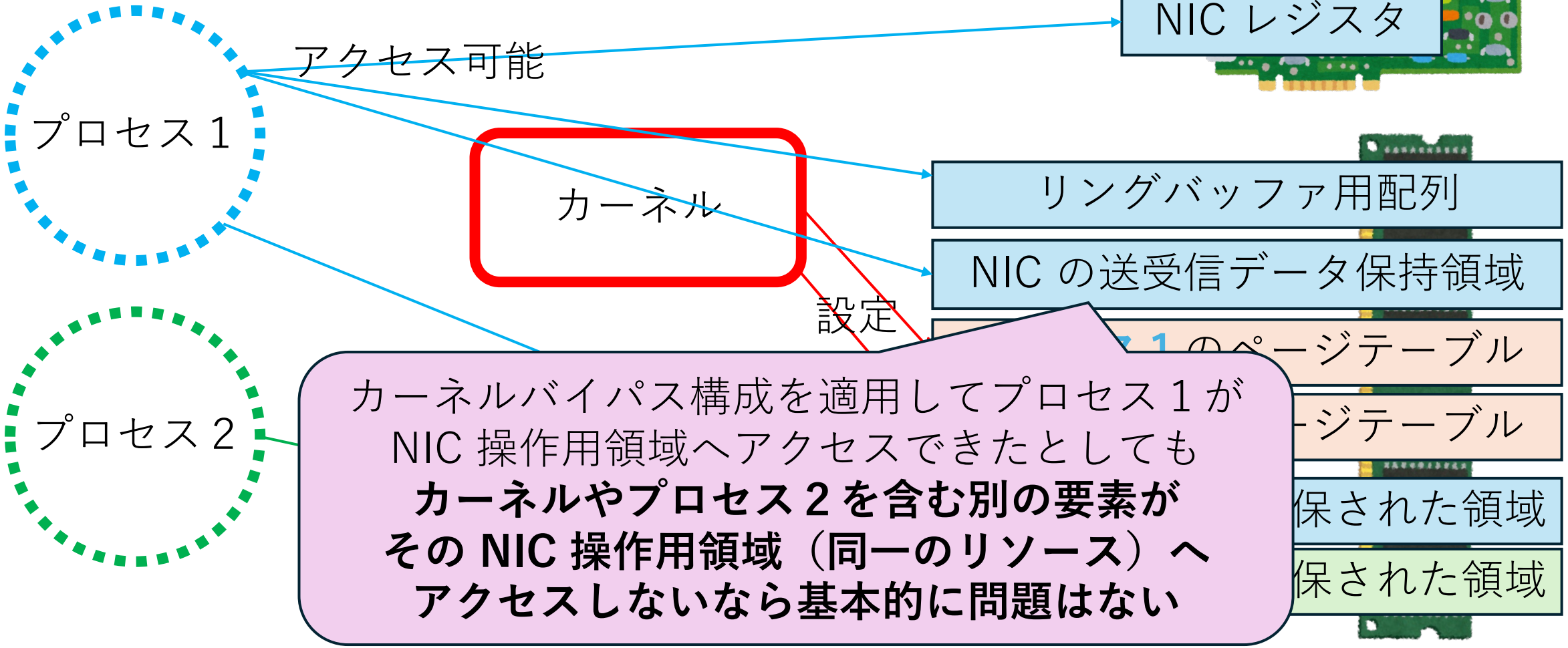
プロセス１用に確保された領域

プロセス２用に確保された領域

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

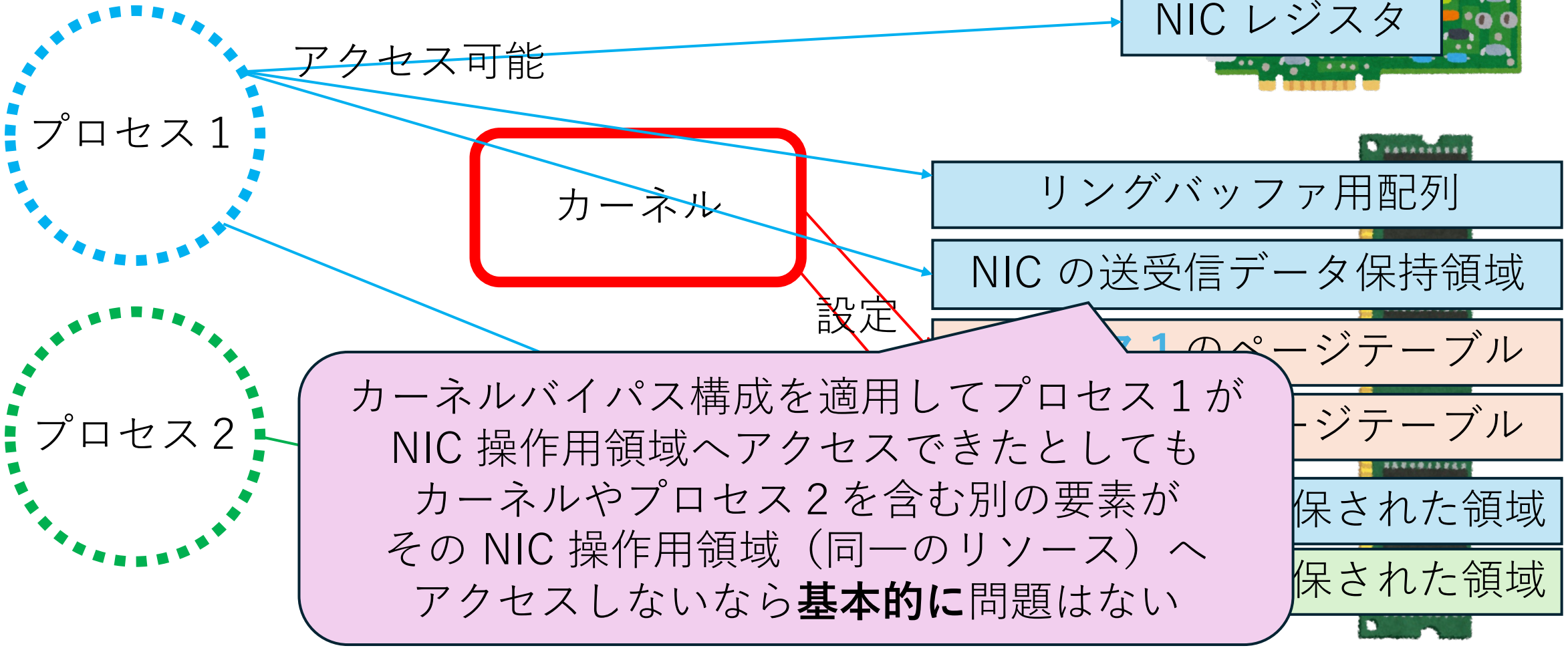
複数の要素が同一のリソースへアクセスしないようにする



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

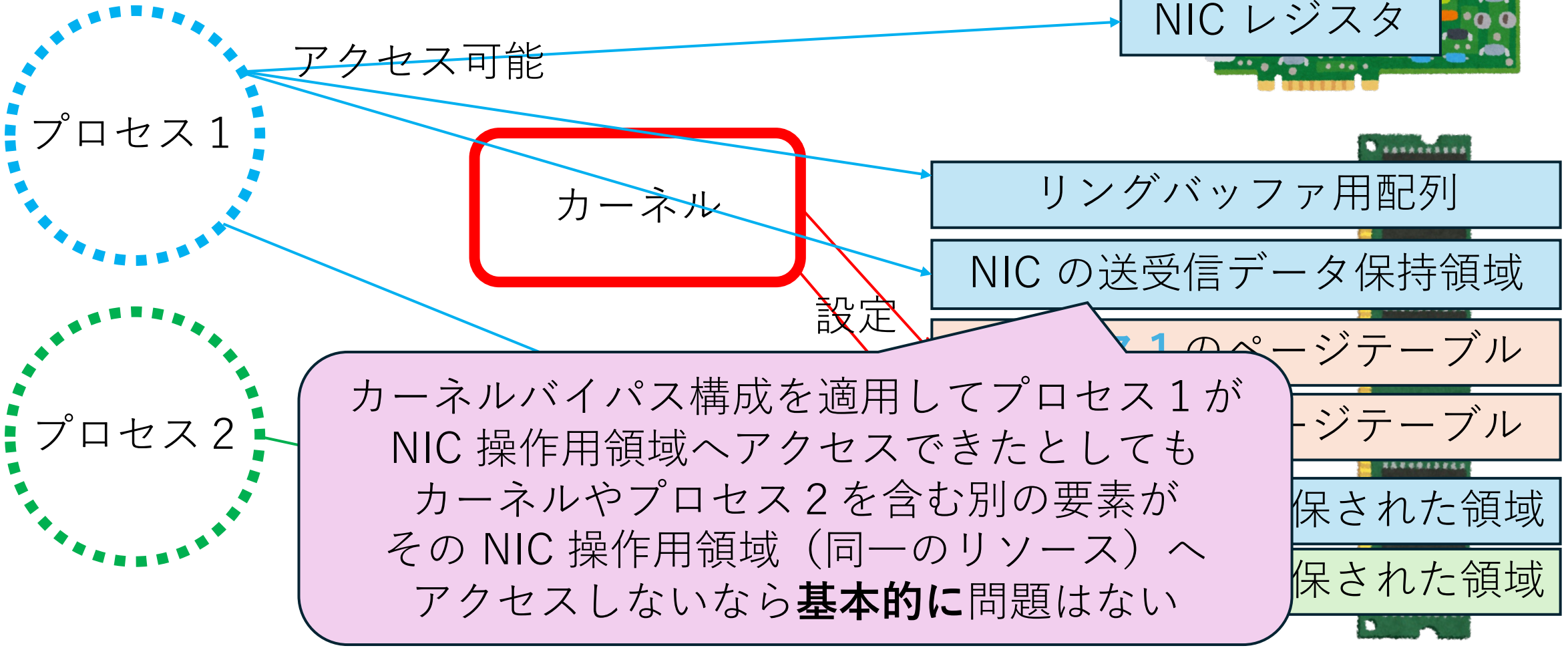
複数の要素が同一のリソースへアクセスしないようにする

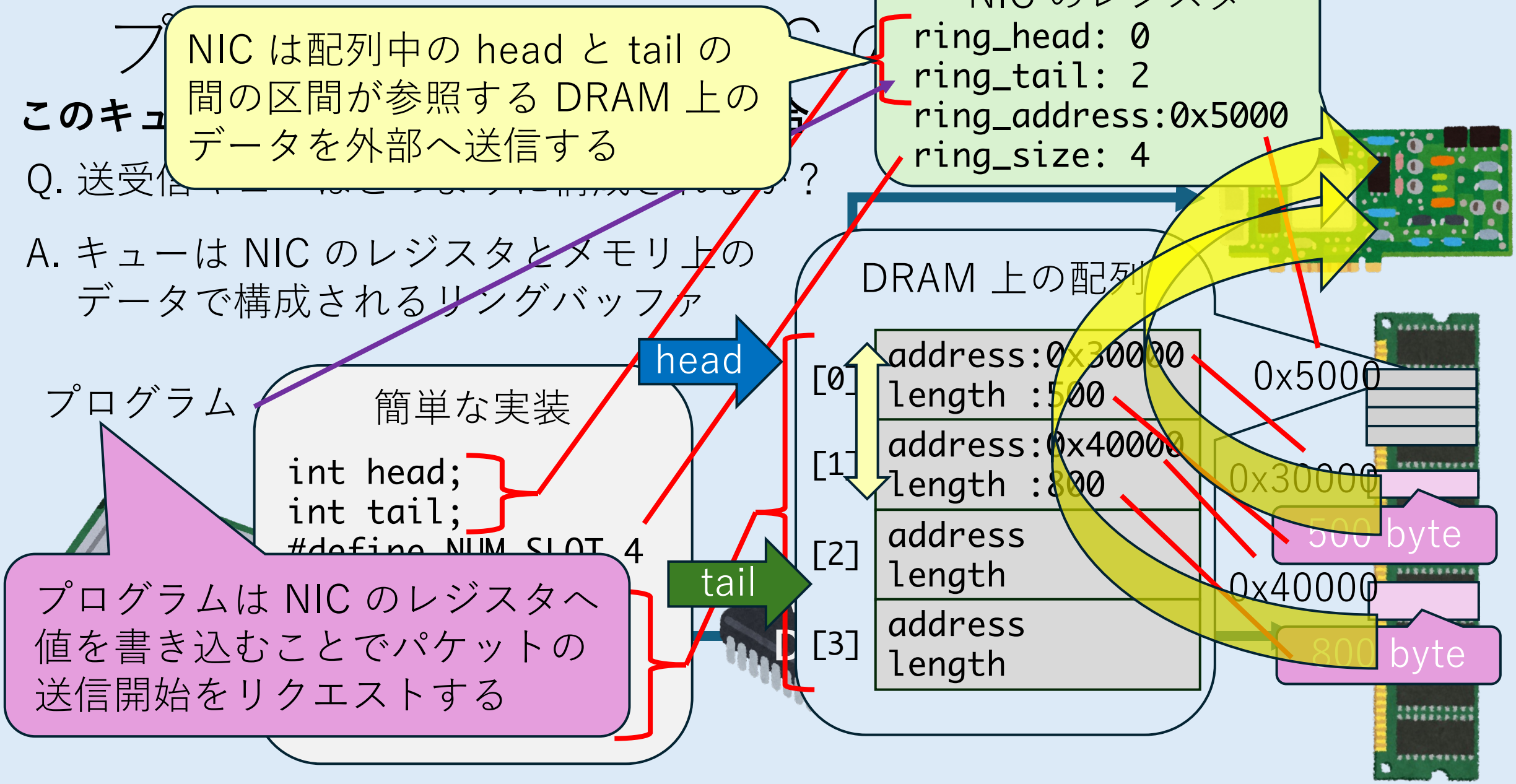


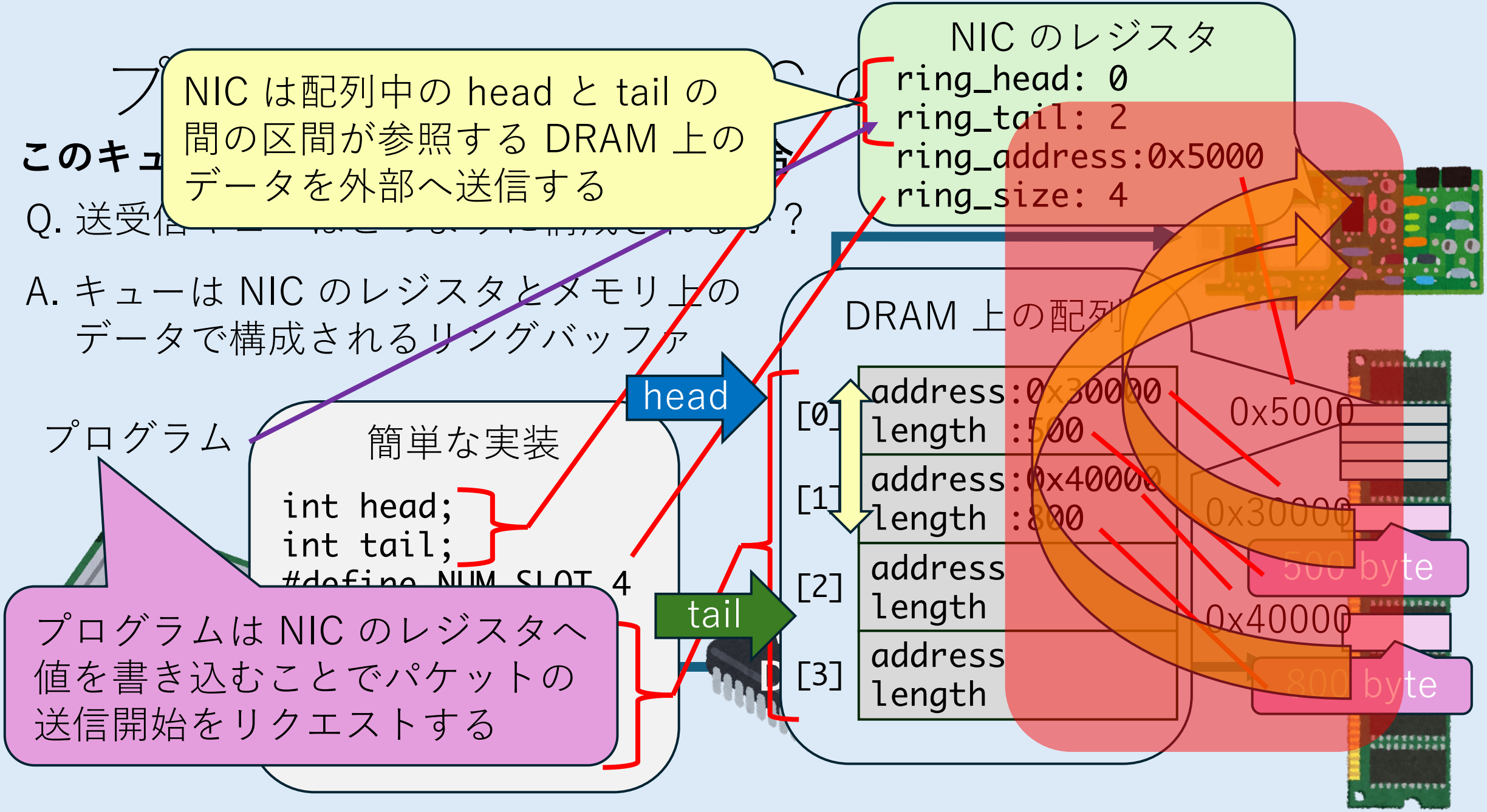
セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス




NIC は配列中の head と tail の
間の区間が参照する DRAM 上の
データを外部へ送信する
NIC のレジスタ
ring_head: 0
ring_tail: 2
ring_address:0x5000
ring_size: 4
このキュ
Q. 送受信キュー
A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ
DRAM 上の配列
head
tail
プログラム
簡単な実装
int head;
int tail;
#define NUM_SLOT 4
[0]
[1]
[2]
[3]
address:0x30000
length :500
address:0x40000
length :800
address
length
address
length
0x5000
0x30000
500 byte
0x40000
800 byte
プログラムは NIC のレジスタへ
値を書き込むことでパケットの
送信開始をリクエストする


NIC は配列中の head と tail の
間の区間が参照する DRAM 上の
データを外部へ送信する
NIC のレジスタ
ring_head: 0
ring_tail: 2
ring_address:0x5000
ring_size: 4
このキュ
Q. 送受信キュー
A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ
DRAM 上の配列
head
tail
プログラム
簡単な実装
int head;
int tail;
#define NUM_SLOT 4
[0]
[1]
[2]
[3]
address:0x30000
length :500
address:0x40000
length :800
address
length
address
length
0x5000
0x30000
500 byte
0x40000
800 byte
プログラムは NIC のレジスタへ
値を書き込むことでパケットの
送信開始をリクエストする

プログ

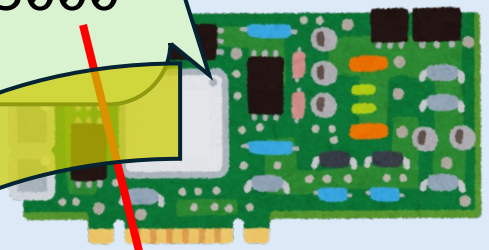
このキューが受信

Q. 送受信キューはどのように構成さ？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

NIC が受信したパケットを
配列の head で参照される
DRAM 領域へ書き込む

NIC のレジスタ
```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

head

tail

[0]	address:0x30000 length
[1]	address:0x40000 length
[2]	address:0x42000 length
[3]	Address:0x44000 length

0x5000

0x30000

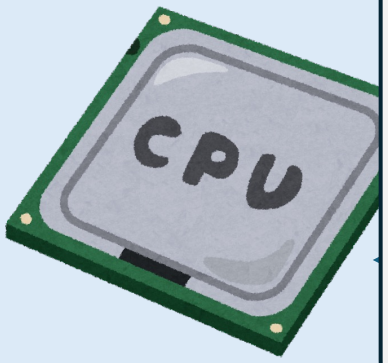
100 byte

0x40000

0x42000

0x44000

プログラム



簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

プログ

このキューが受信

Q. 送受信キューはどのように構成さ？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

NIC が受信したパケットを
配列の head で参照される
DRAM 領域へ書き込む

NIC のレジスタ
```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```

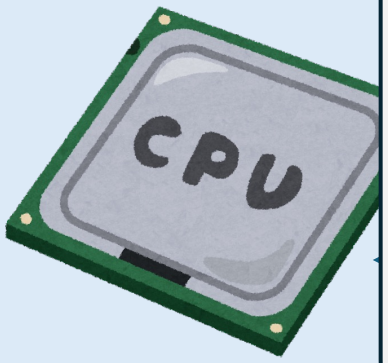
DRAM 上の配列

head
tail

[0]	address:0x30000 length
[1]	address:0x40000 length
[2]	address:0x42000 length
[3]	Address:0x44000 length

0x5000
0x30000
100 byte
0x40000
0x42000
0x44000

プログラム



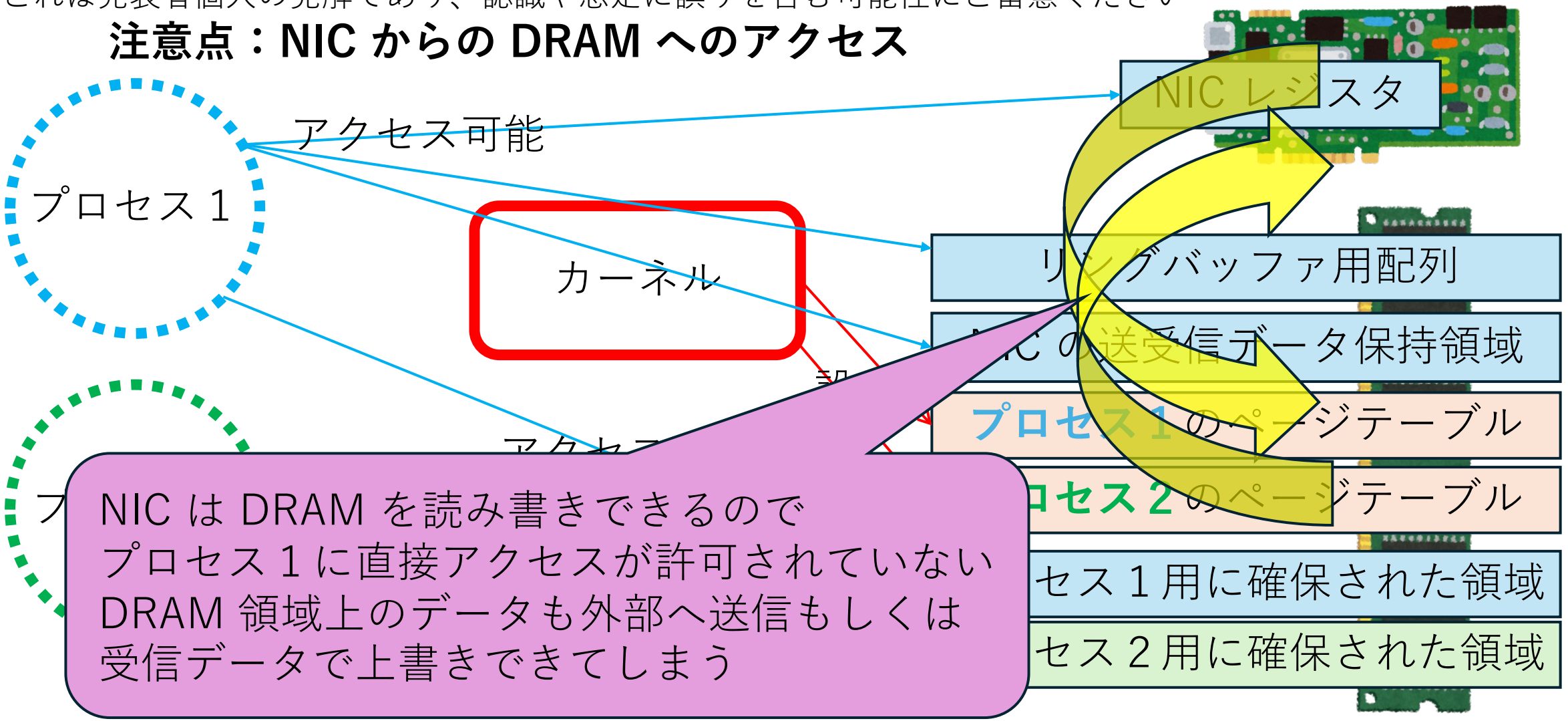
簡単な実装

```
int head;
int tail;
#define NUM_SLOT 4
struct {
  void *address;
  int   length;
} slot[NUM_SLOT];
```

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

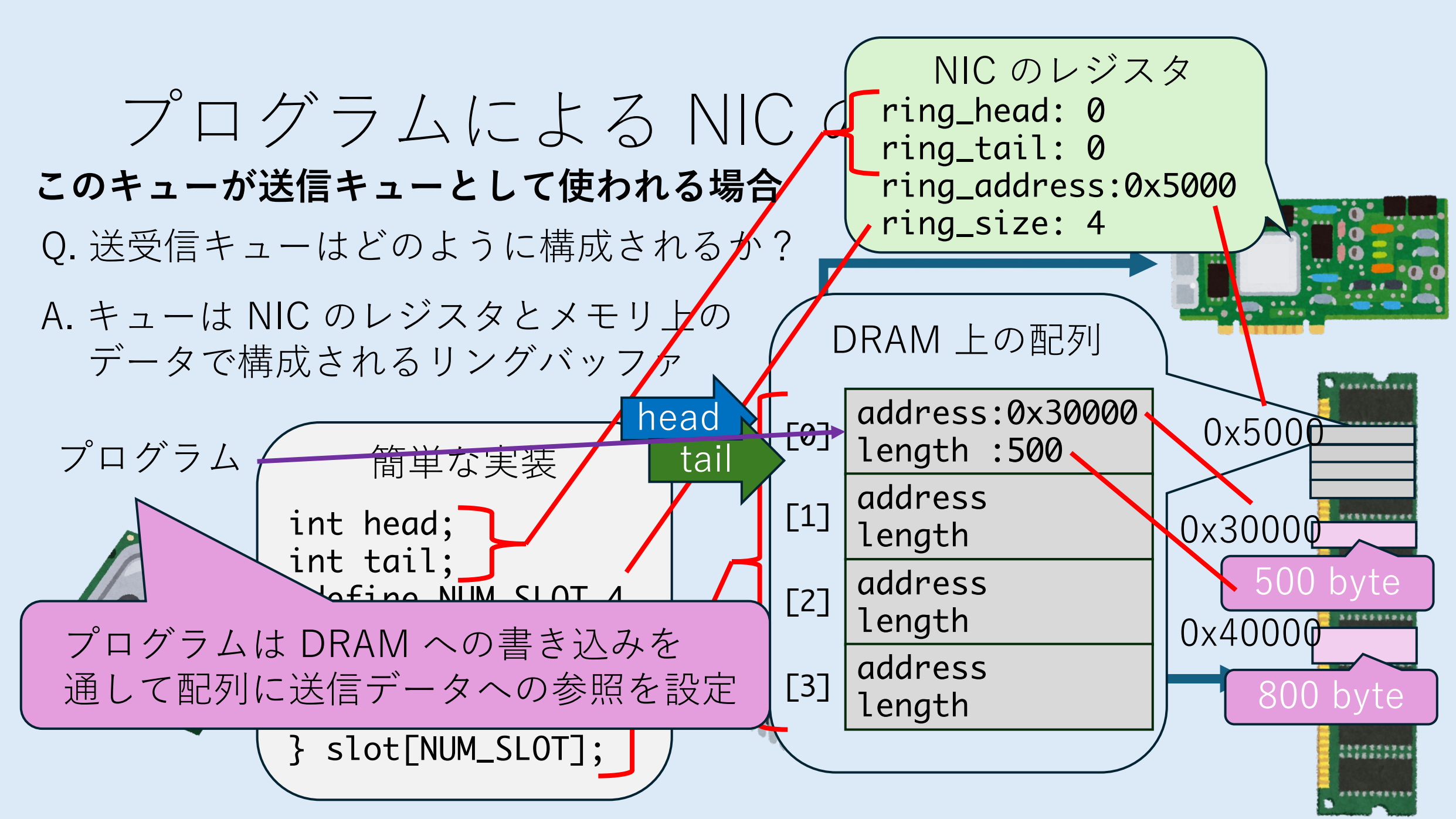


プログラムによる NIC の

このキューが送信キューとして使われる場合

Q. 送受信キューはどのように構成されるか？

A. キューは NIC のレジスタとメモリ上の
データで構成されるリングバッファ

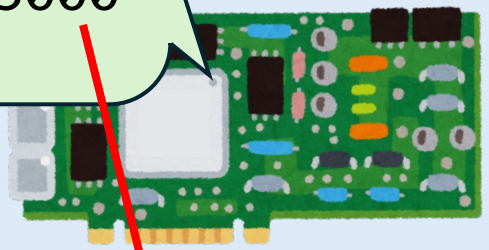


プログラムによる NIC の

このキューが送信キューとして使われる場合

物理アドレス 0x30000 の領域は
プロセス１に割り当てられていなくても
プロセス１は配列に参照を設定することで
NIC からの送信対象に設定できる

NIC のレジスタ
```
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4
```



DRAM 上の配列

[0]	address:0x30000 length :500
[1]	address length
[2]	address length
[3]	address length

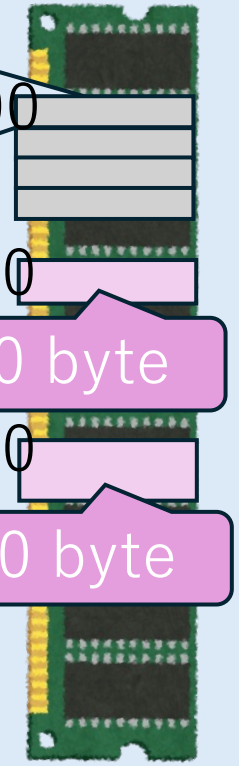
0x5000

0x30000

500 byte

0x40000

800 byte



プログラム

簡単な実装

head

tail

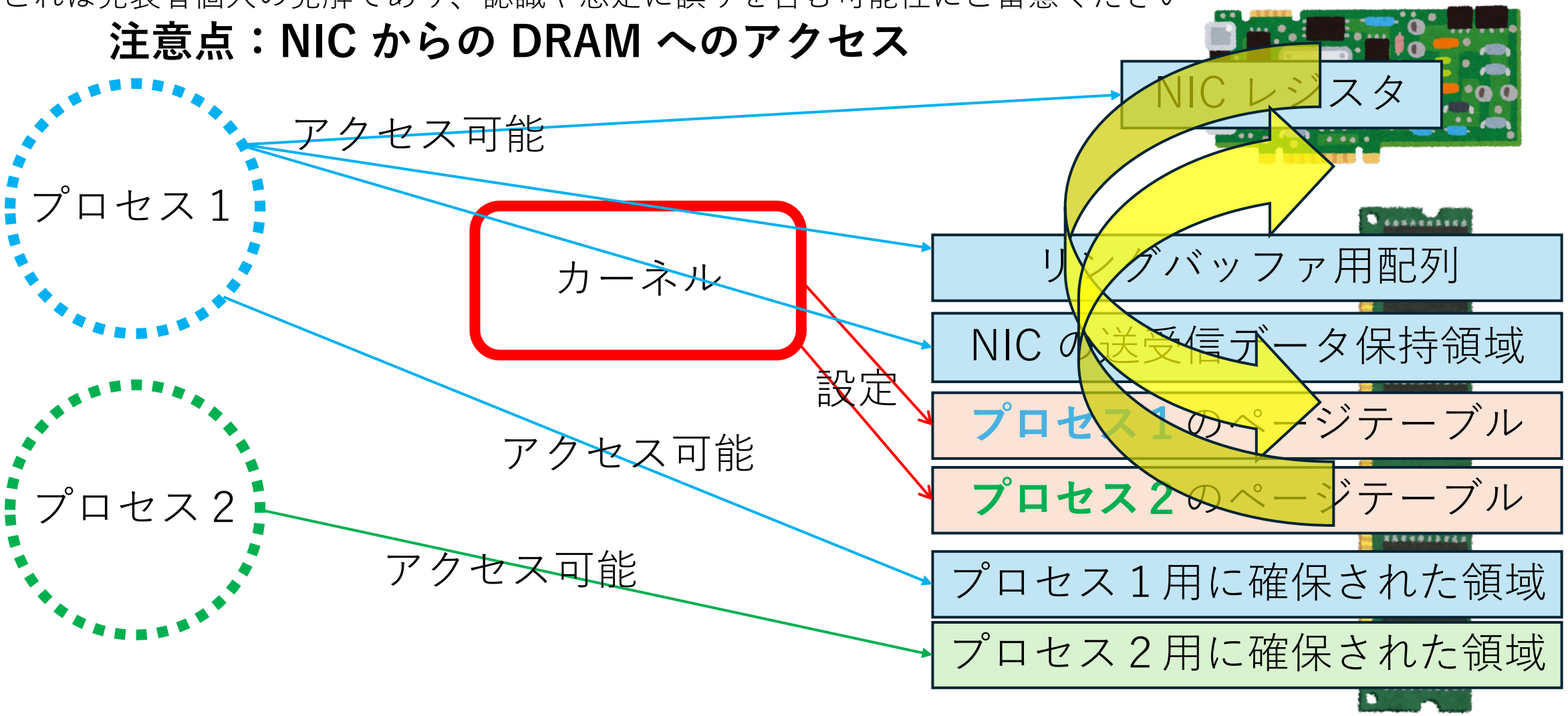
```
int head;
int tail;
#define NUM_SLOT 4
} slot[NUM_SLOT];
```

プログラムは DRAM への書き込みを
通して配列に送信データへの参照を設定

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

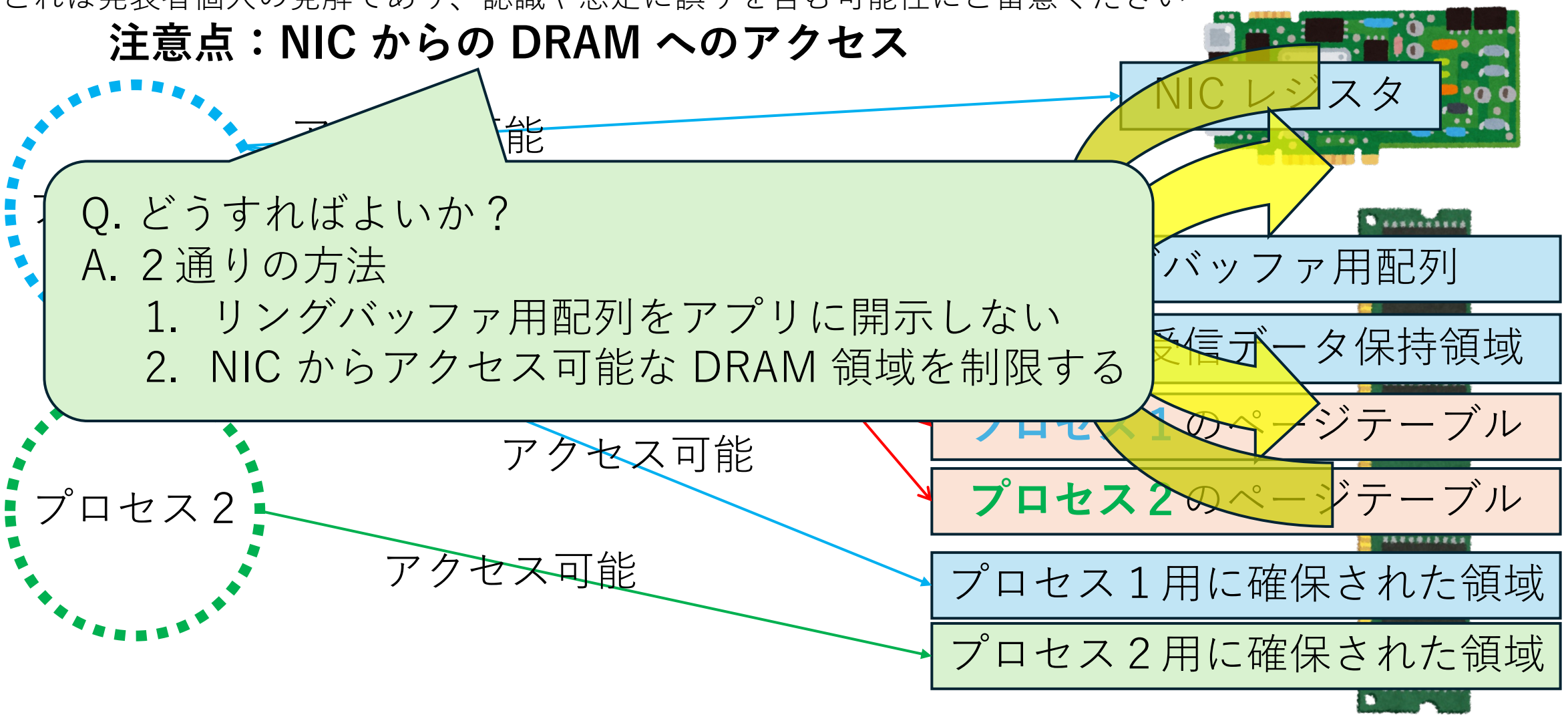
注意点：NIC からの DRAM へのアクセス



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

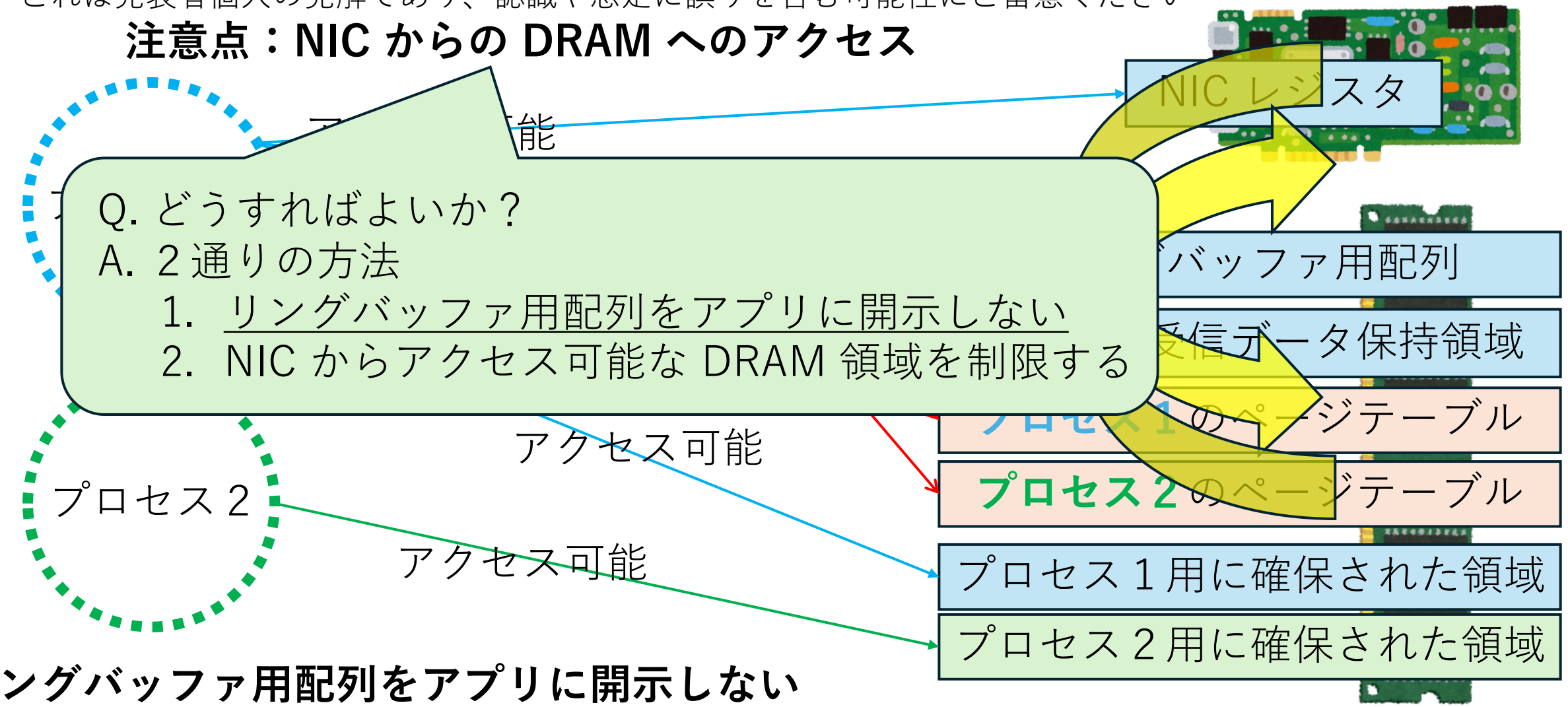
注意点：NIC からの DRAM へのアクセス



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

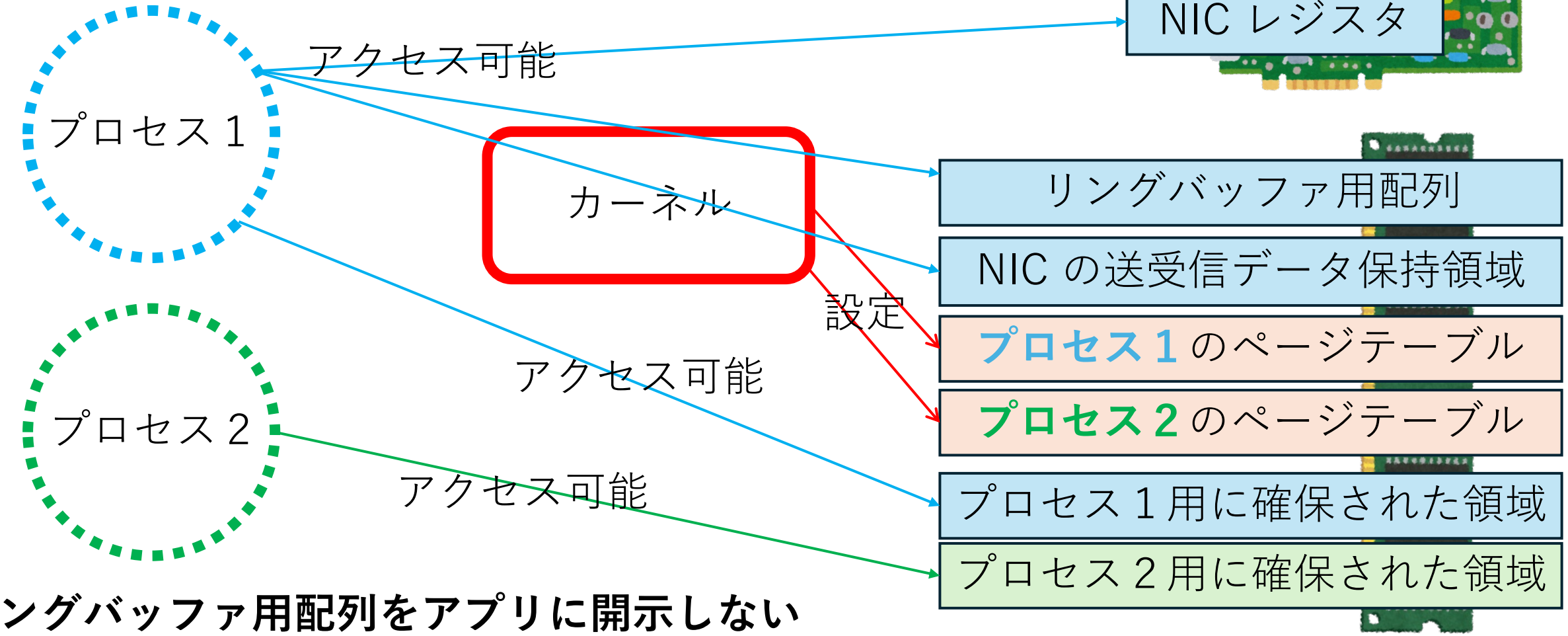


リングバッファ用配列をアプリに開示しない

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

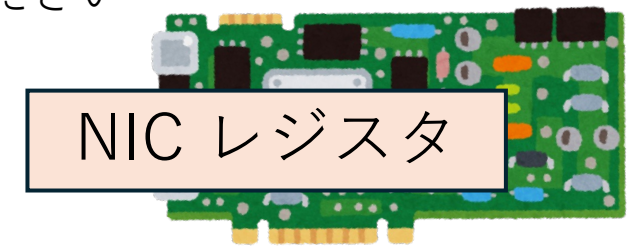


リングバッファ用配列をアプリに開示しない

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス



プロセス１

プロセス２

アクセス可能

カーネル

設定

アクセス可能

アクセス可

リングバッファ用配列

NIC の送受信データ保持領域

プロセス

ページテーブル

プロセス２用に確保された領域

プロセス１のアプリに対しては NIC の
送受信データ保持領域のみへアクセスを許可

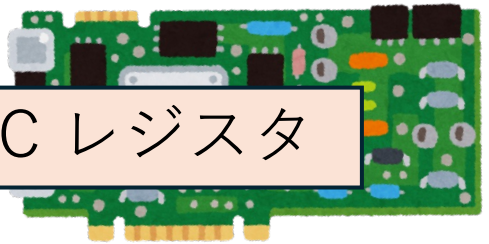
リングバッファ用配列をアプリに開示しない

セキュリティについての考え方

これは発表ください

注

プロセス１のアプリはカーネルへリクエスト
（専用のシステムコールの呼び出し）を通して
代わりに NIC を操作してもらう



NIC レジスタ

プロセス１

アクセス可能

リクエスト

カーネル

操作

設定

リングバッファ用配列

NIC の送受信データ保持領域

プロセス

ページテーブル

プロセス２

アクセス可

プロセス１のアプリに対しては NIC の
送受信データ保持領域のみへアクセスを許可

プロセス２用に確保された領域

リングバッファ用配列をアプリに開示しない

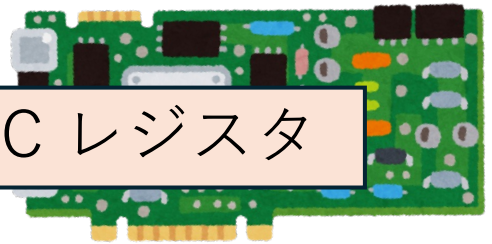
セキュリティについての考え方

これは発表

ください

注

プロセス１のアプリはカーネルへリクエスト（専用のシステムコールの呼び出し）を通して代わりに NIC を操作してもらう



NIC レジスタ

プロセス１

アクセス可能

リクエスト

カーネル

操作

設定

リングバッファ用配列

NIC の送受信データ保持領域

プロセス１用ページテーブル

プロセス２用に確保された領域

想定として
アプリは信頼されず
カーネルは信頼されている
というところがポイントです

プロセス１のアプリに対しては NIC の送受信データ保持領域のみへアクセスを許可

リングバッファ用配列をアプリに開示しない

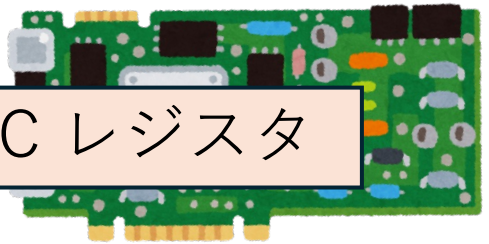
セキュリティについての考え方

これは発表

ください

注

プロセス１のアプリはカーネルへリクエスト
（専用のシステムコールの呼び出し）を通して
代わりに NIC を操作してもらう



NIC レジスタ

プロセス１

アクセス可能

リクエスト

操作

仮にアプリをカーネルと同じく信頼する想定であれば
リングバッファ用配列を直接開示して問題ないです

想定として
アプリは信頼されず
カーネルは信頼されている
というところがポイントです

プロセス

ページテーブル

プロセス１のアプリに対しては NIC の
送受信データ保持領域のみへアクセスを許可

プロセス２用に確保された領域

リングバッファ用配列をアプリに開示しない

プログラムによる NIC の

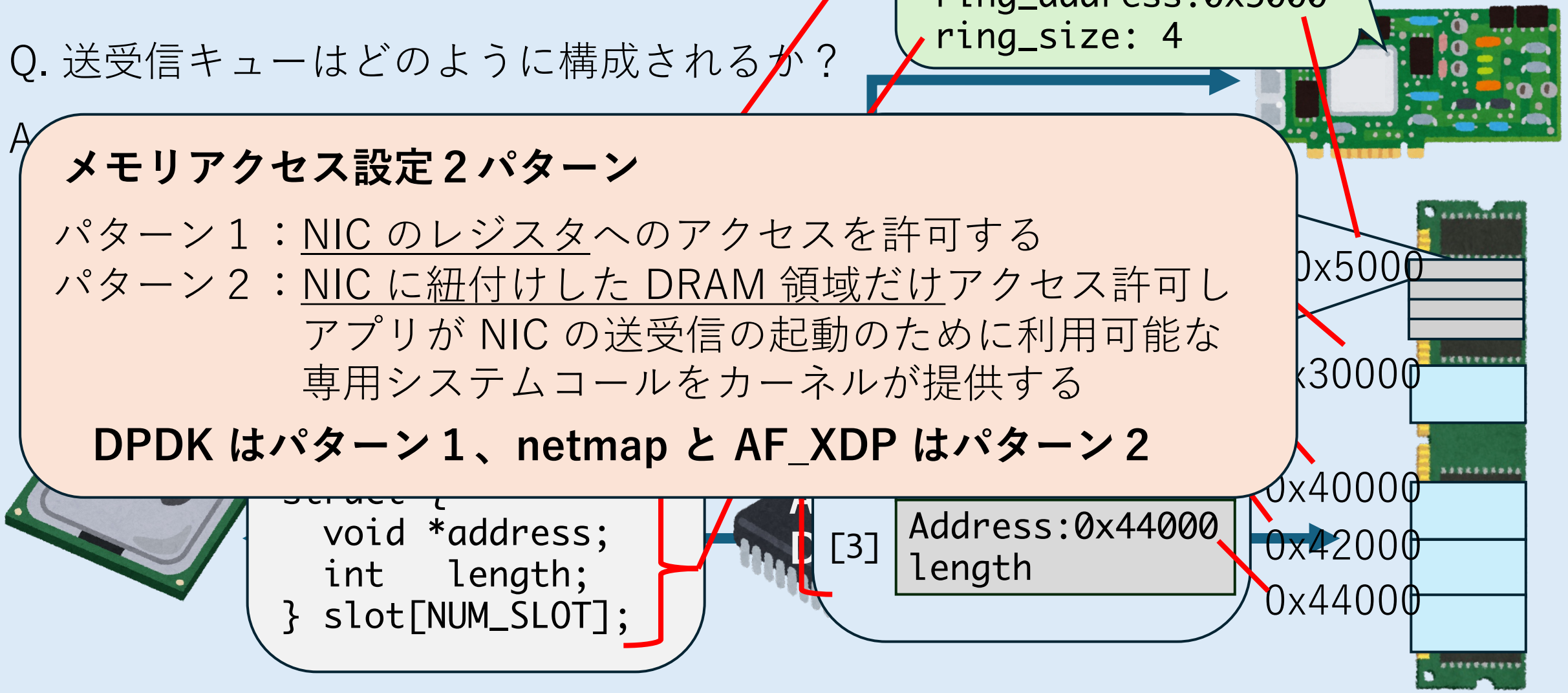
Q. 送受信キューはどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する
パターン２：NIC に紐付けした DRAM 領域だけアクセス許可し
アプリが NIC の送受信の起動のために利用可能な
専用システムコールをカーネルが提供する

DPDK はパターン１、netmap と AF_XDP はパターン２



プログラムによる NIC の

Q. 送受信キューはどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する

パターン２：NIC に紐付けした DRAM 領域だけアクセス許可し
アプリが NIC の送受信の起動のために利用可能な
専用システムコールをカーネルが提供する

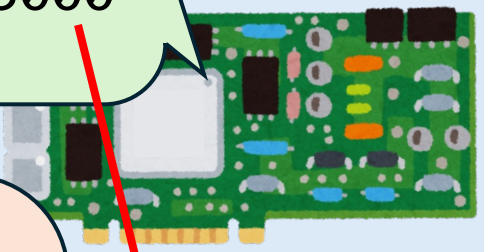
DPDK はパターン１、netmap と AF_XDP はパターン２

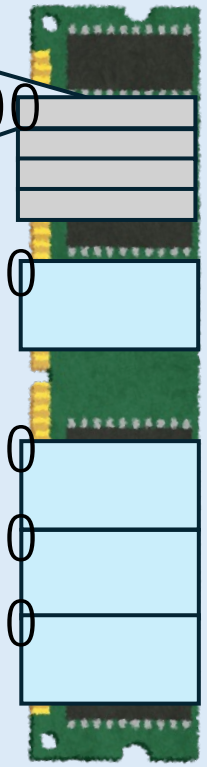
```
  void *address;
  int   length;
} slot[NUM_SLOT];
```

[3]

Address:0x44000
length

NIC のレジスタ
ring_head: 0
ring_tail: 0
ring_address:0x5000
ring_size: 4





0x5000

x30000

0x40000

0x42000

0x44000

プログラムによる NIC の

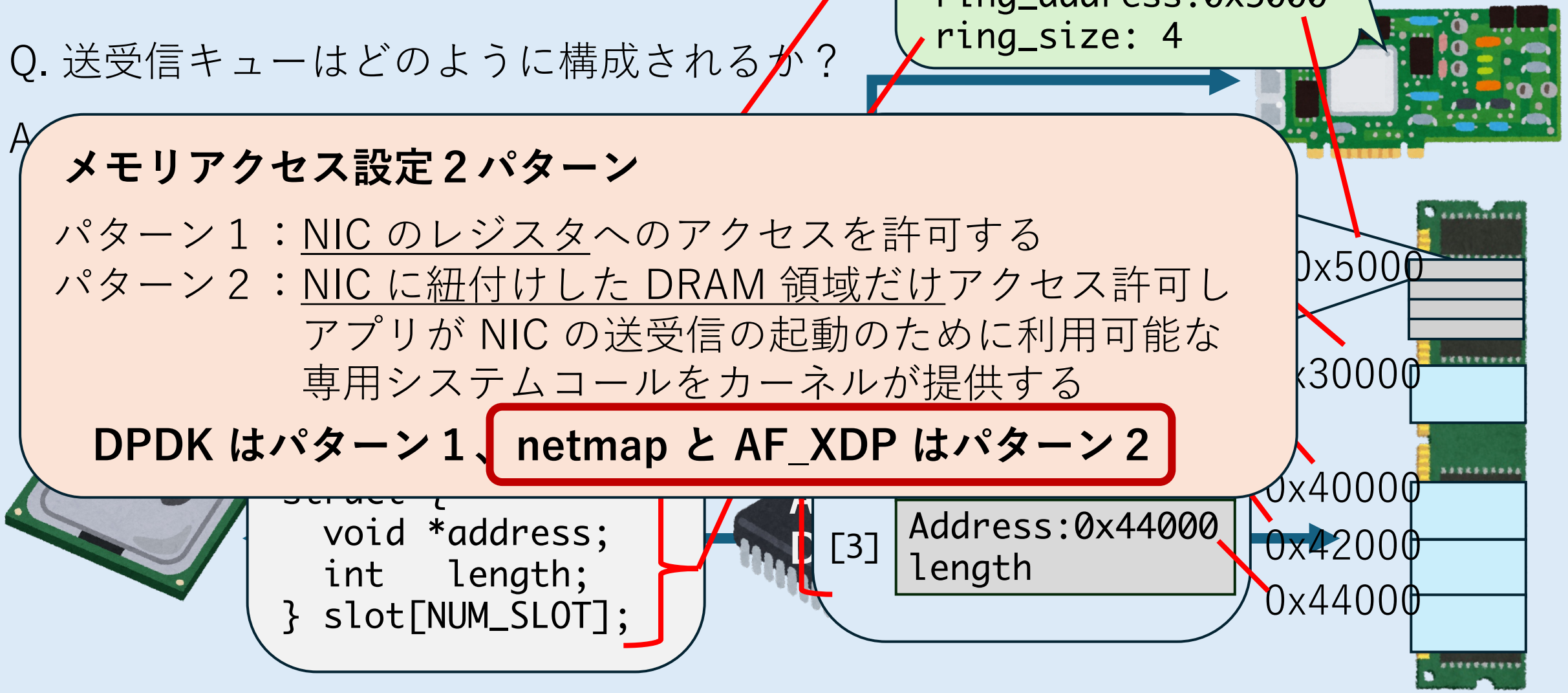
Q. 送受信キューはどのように構成されるか？

A

メモリアクセス設定２パターン

パターン１：NIC のレジスタへのアクセスを許可する
パターン２：NIC に紐付けした DRAM 領域だけアクセス許可し
アプリが NIC の送受信の起動のために利用可能な
専用システムコールをカーネルが提供する

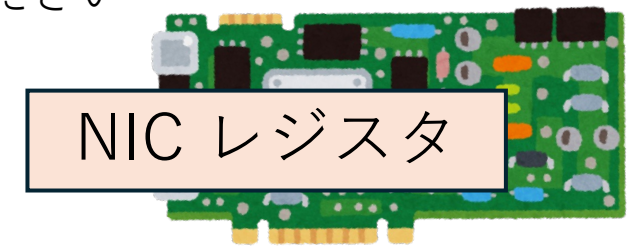
DPDK はパターン１、netmap と AF_XDP はパターン２



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス



プロセス１

プロセス２

アクセス可能

カーネル

設定

アクセス可能

アクセス可

リングバッファ用配列

NIC の送受信データ保持領域

プロセス

ページテーブル

プロセス２用に確保された領域

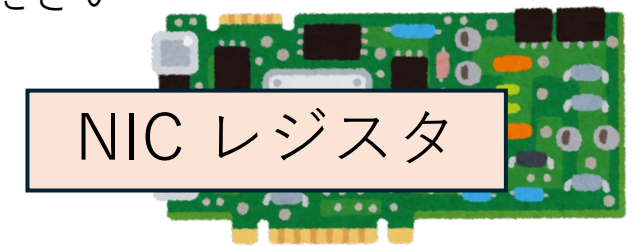
プロセス１のアプリに対しては NIC の
送受信データ保持領域のみへアクセスを許可

リングバッファ用配列をアプリに開示しない

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス



プロセス１

プロセス２

アクセス可能

カーネル

設定

netmap か AF_XDP を利用すると
この構成が適用されます

NIC の送受信データ保持領域

プロセス１用に確保された領域

プロセス２用に確保された領域

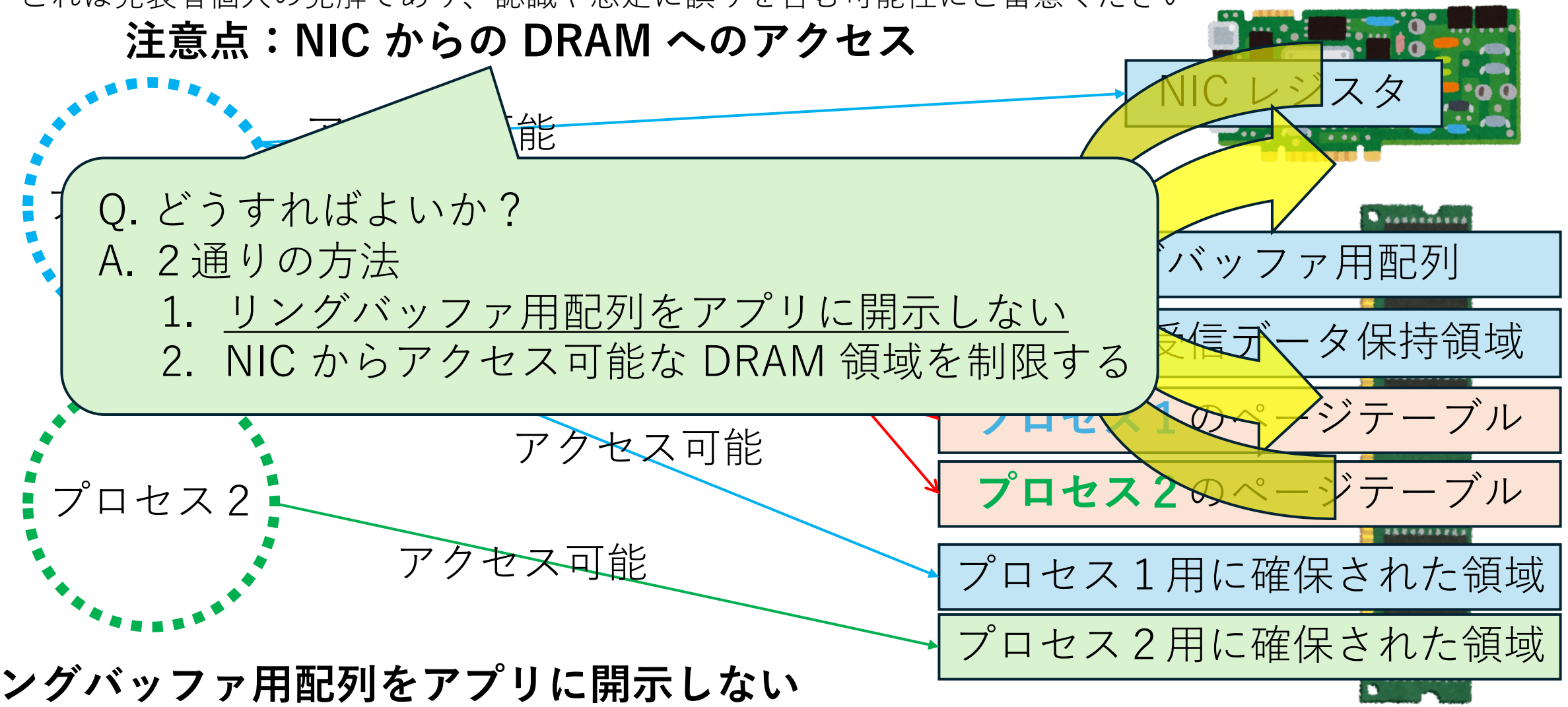
プロセス１のアプリに対しては NIC の
送受信データ保持領域のみへアクセスを許可

リングバッファ用配列をアプリに開示しない

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

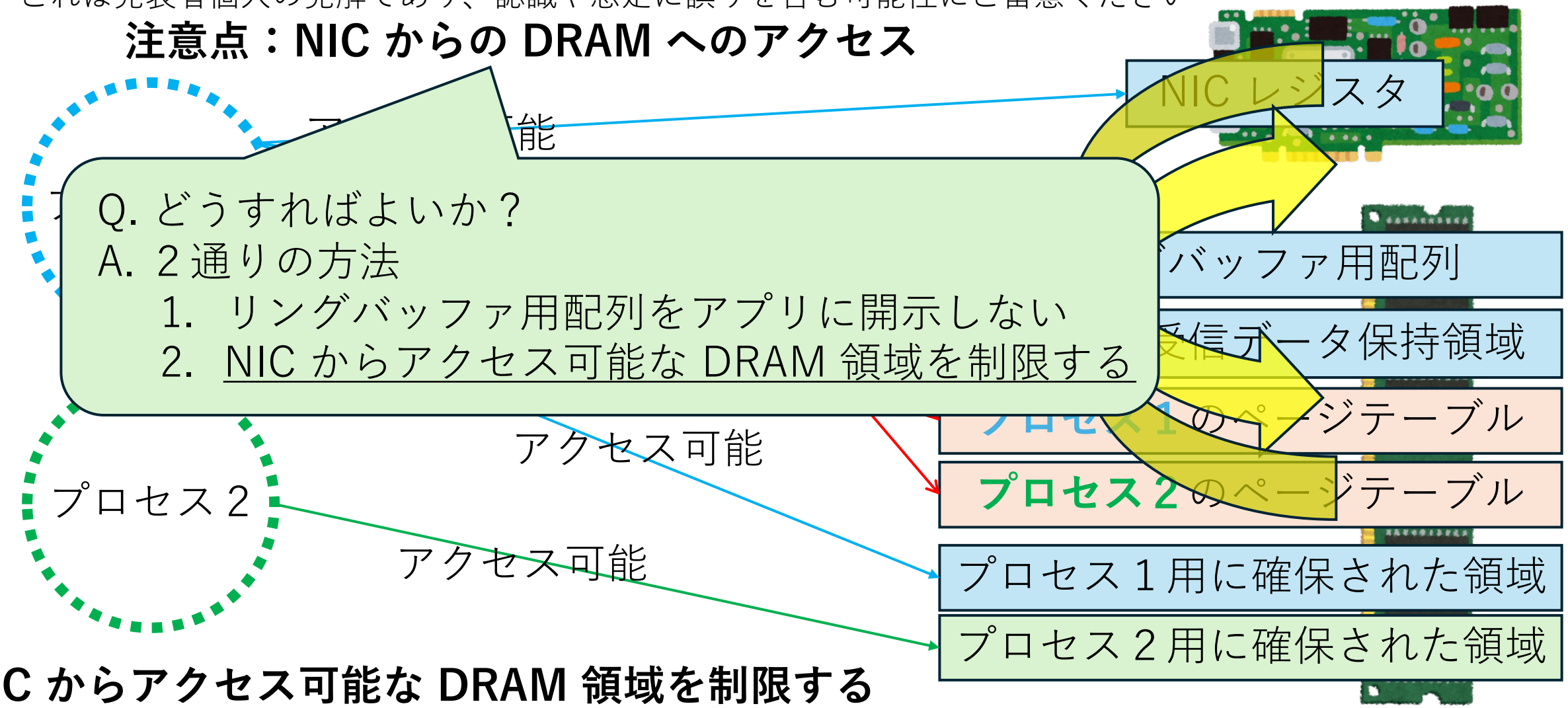


リングバッファ用配列をアプリに開示しない

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

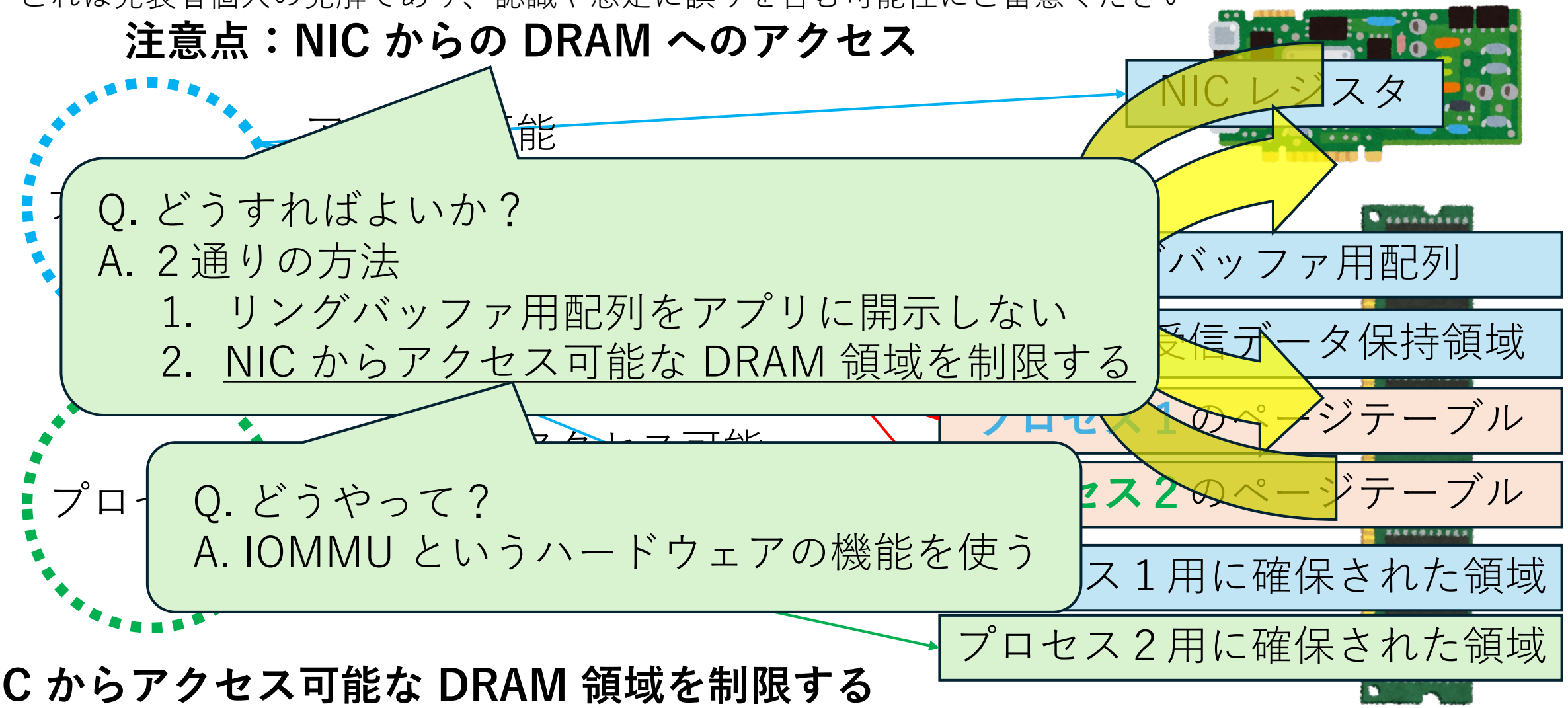


NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

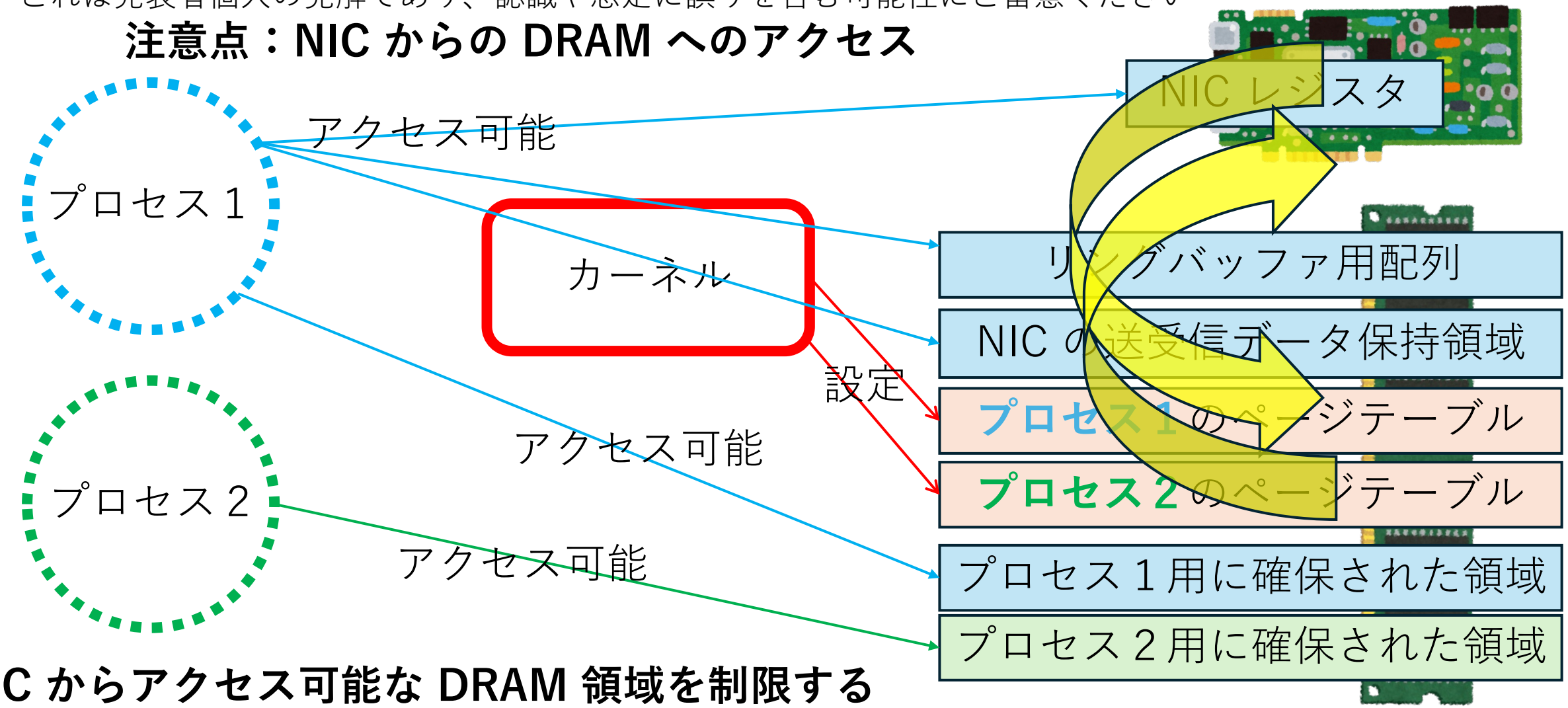


NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

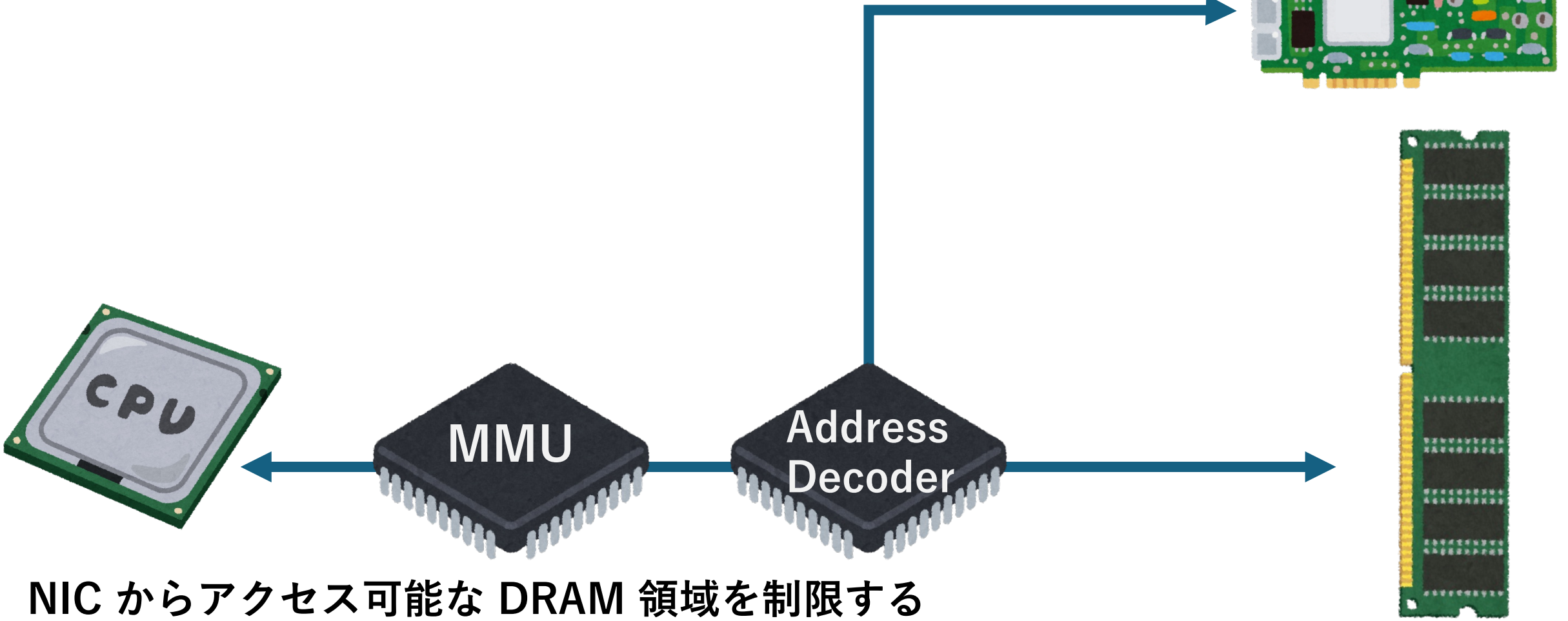


NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

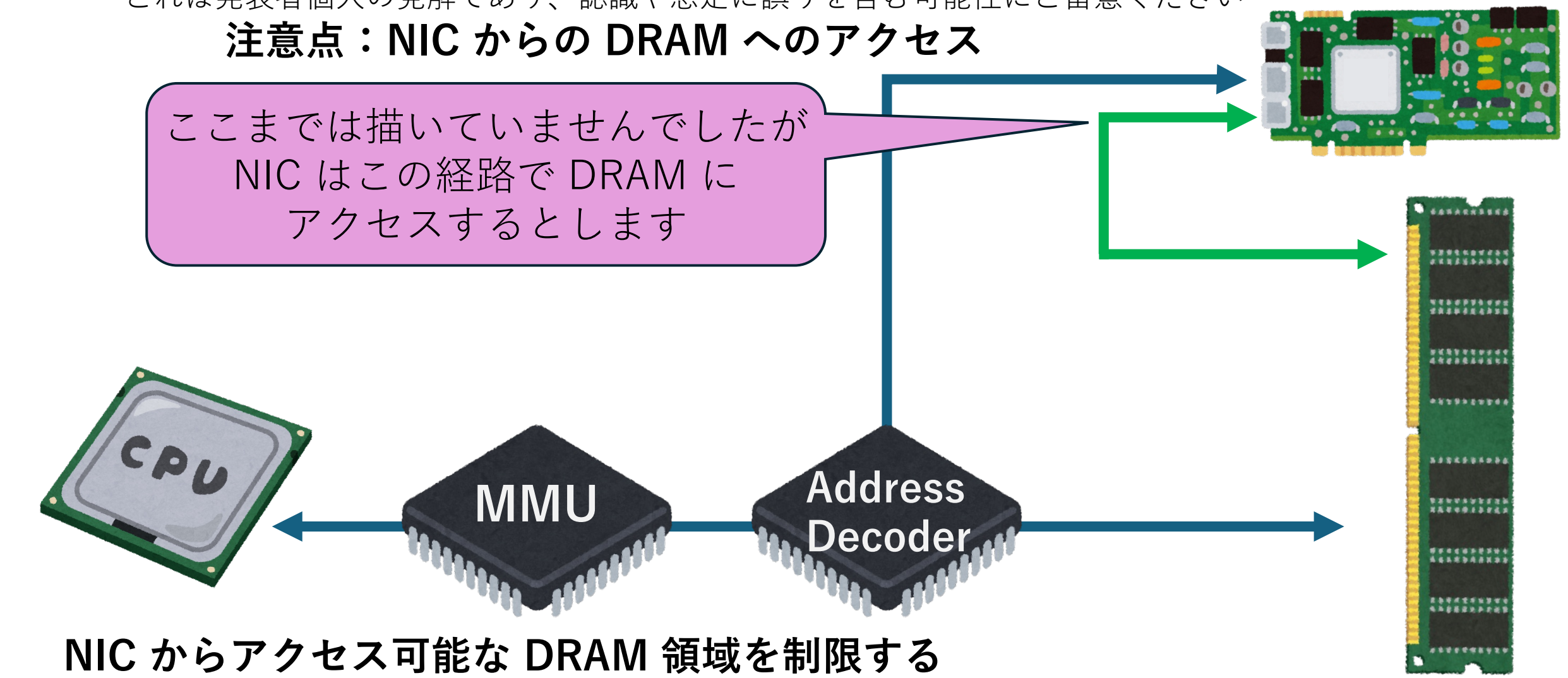


NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

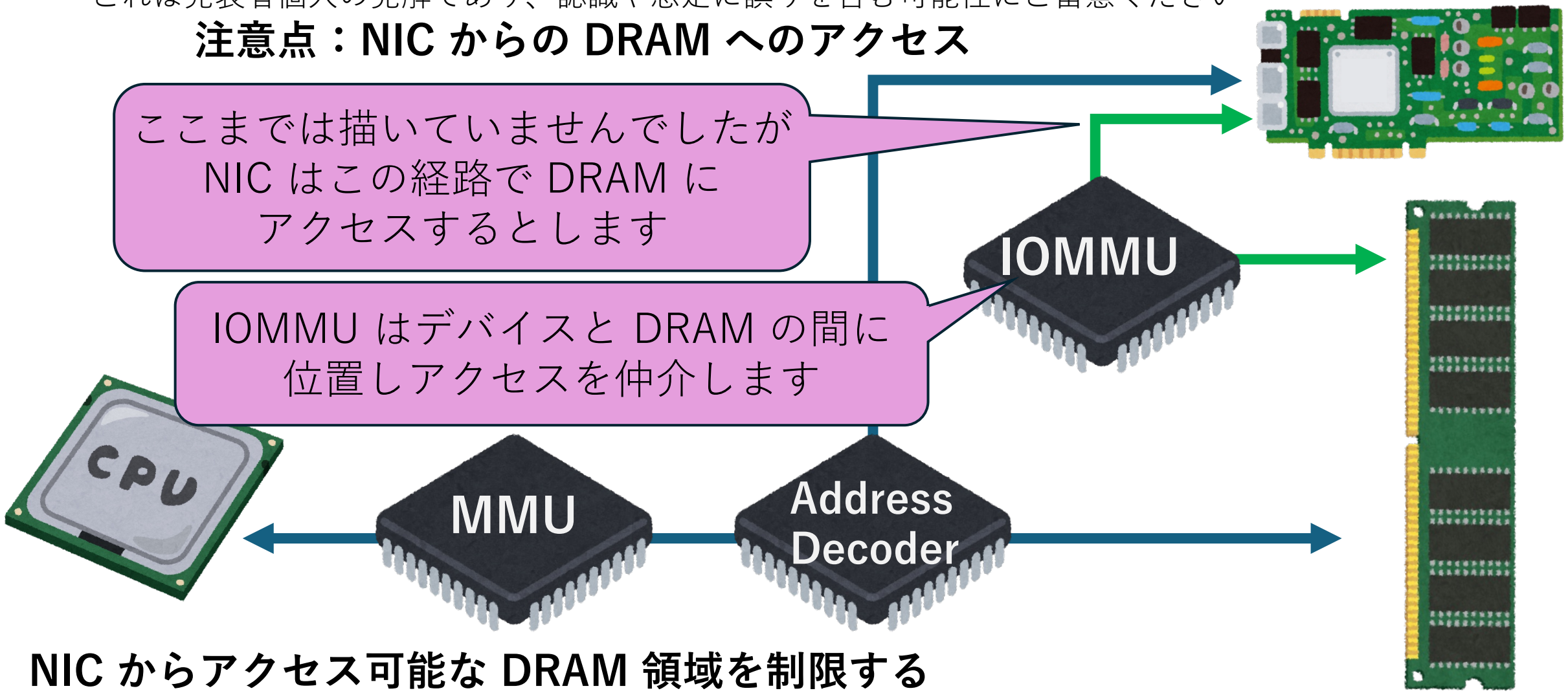


NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

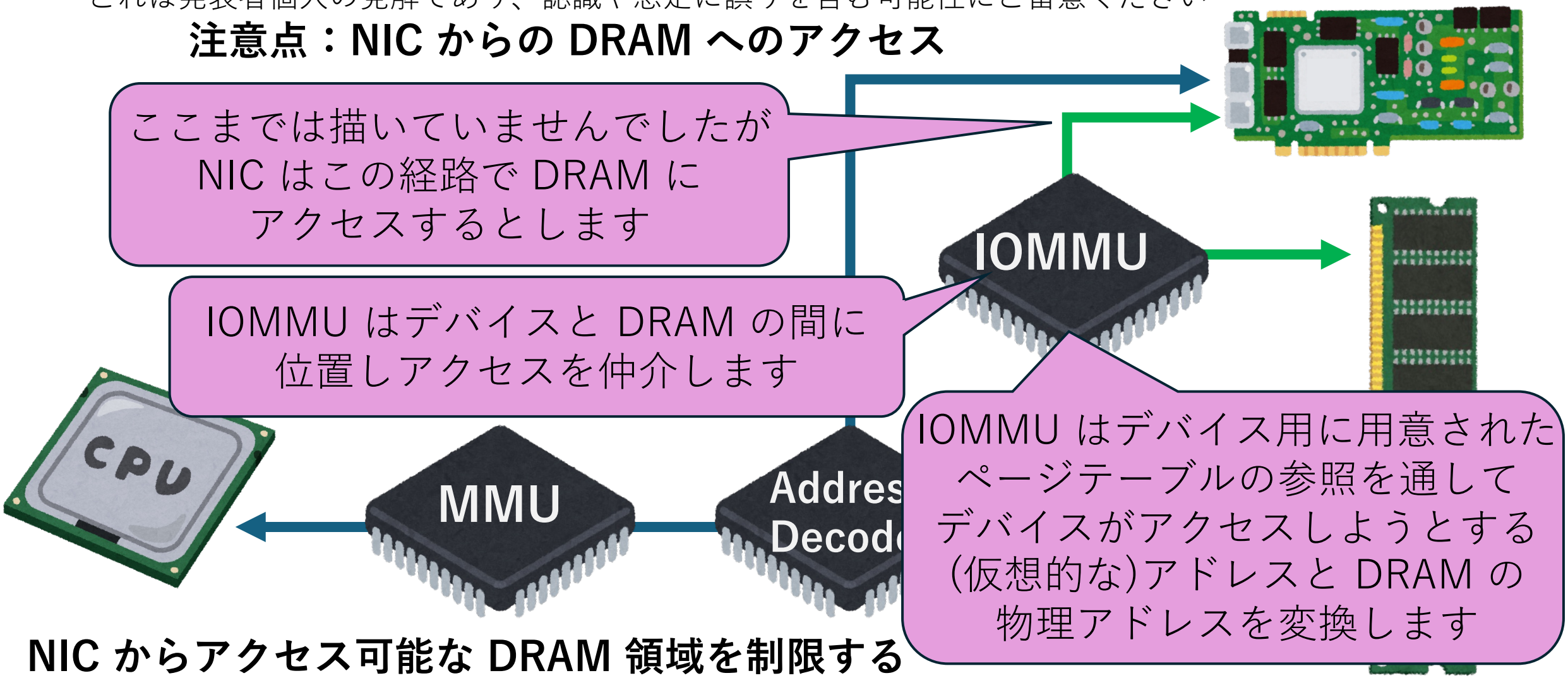


NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

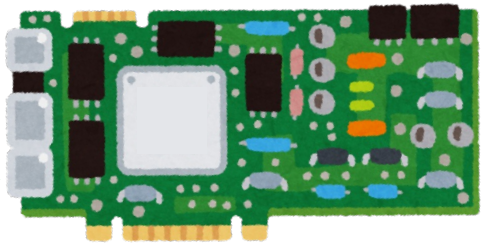


NIC からアクセス可能な DRAM 領域を制限する

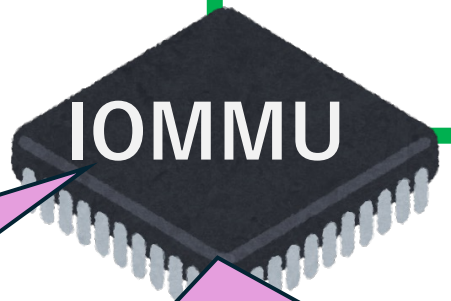
セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス



ここまでは描いていませんでしたが
NIC はこの経路で DRAM に
アクセスするとします



IOMMU はデバイスと DRAM の間に
位置しアクセスを仲介します

IOMMU はデバイス用に用意された
ページテーブルの参照を通して
デバイスがアクセスしようとする
(仮想的な)アドレスと DRAM の
物理アドレスを変換します

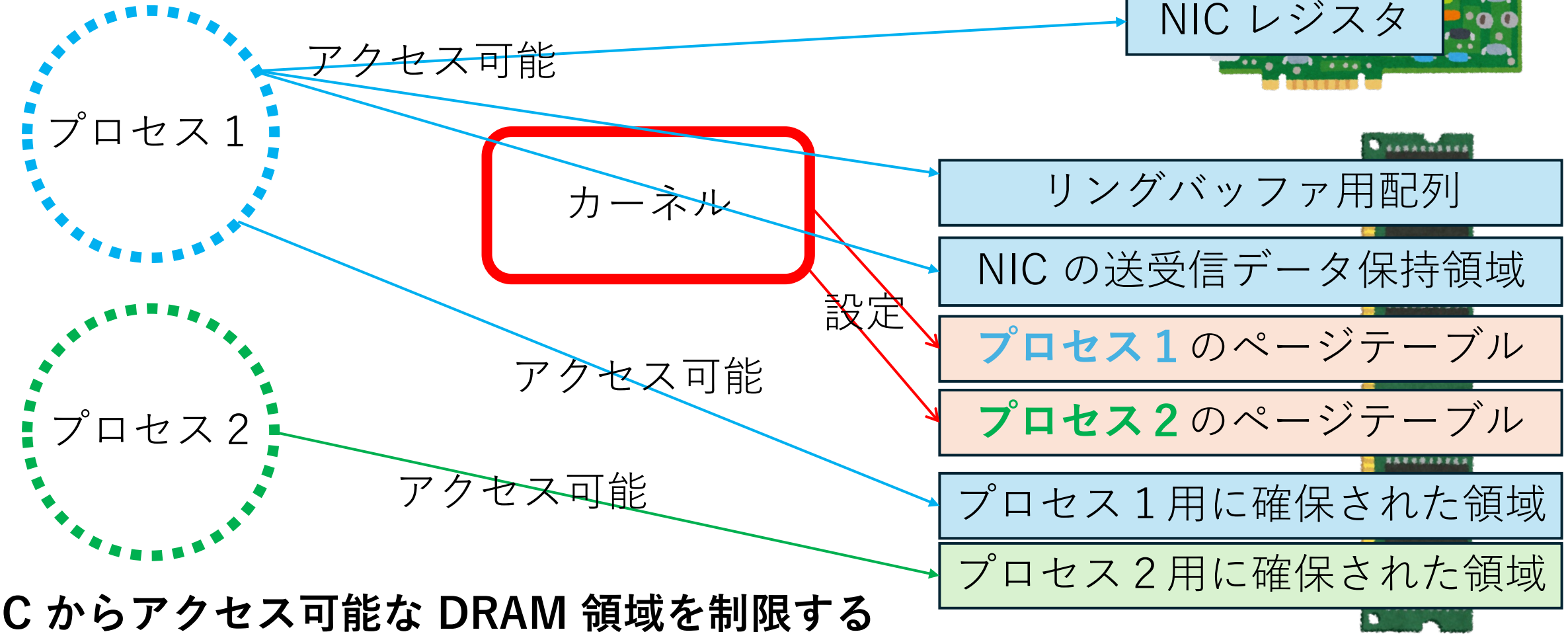
IOMMU が有効な場合には NIC からは
このページテーブルに記載のある
DRAM の物理アドレスにしか
アクセスできません

NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

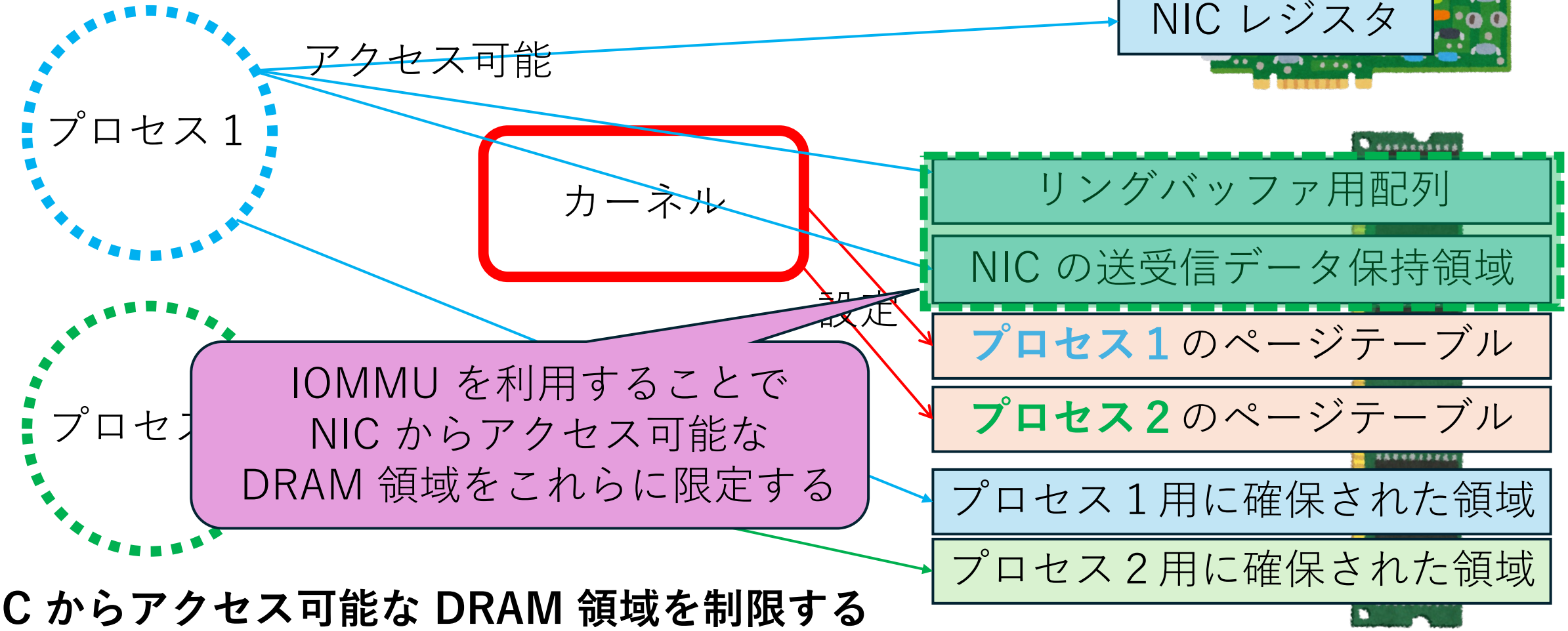


NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス

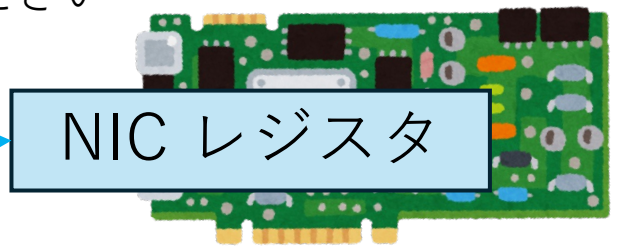


NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

注意点：NIC からの DRAM へのアクセス



NIC レジスタ

アクセス可能

プロセス 1

Linux では DPDK 利用時には
vfio という機能を使うことで
この設定を適用できます

設定

プロセス

IOMMU を利用することで
NIC からアクセス可能な
DRAM 領域をこれらに限定する

リングバッファ用配列

NIC の送受信データ保持領域

プロセス１のページテーブル

プロセス２のページテーブル

プロセス１用に確保された領域

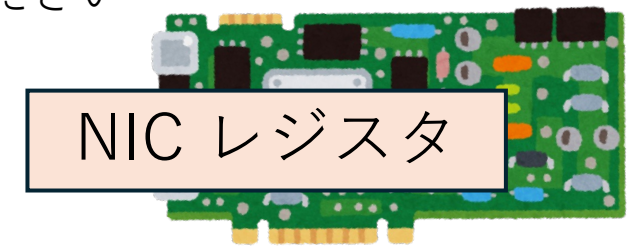
プロセス２用に確保された領域

NIC からアクセス可能な DRAM 領域を制限する

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

推奨されない（基本的に採用されない）設定例



プロセス１

プロセス２

カーネル

設定

アクセス可能

アクセス可能

リングバッファ用配列

NIC の送受信データ保持領域

プロセス１のページテーブル

プロセス２のページテーブル

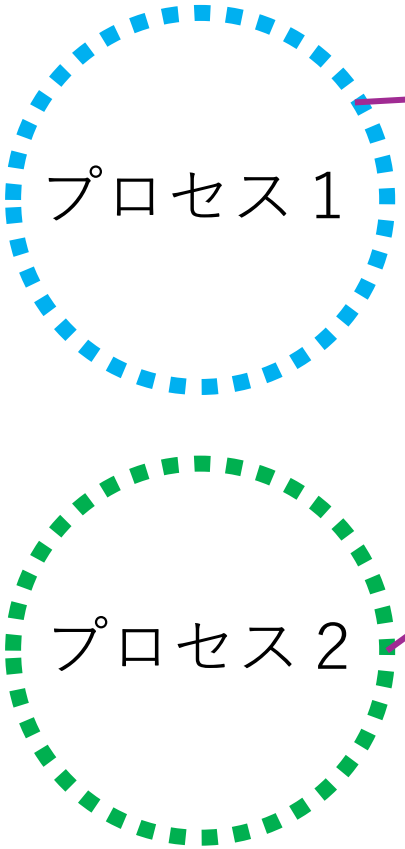
プロセス１用に確保された領域

プロセス２用に確保された領域

セキュリティについての考え方

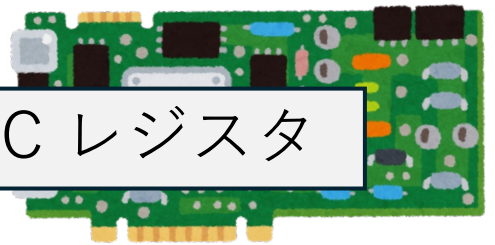
これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

推奨されない（基本的に採用されない）設定例



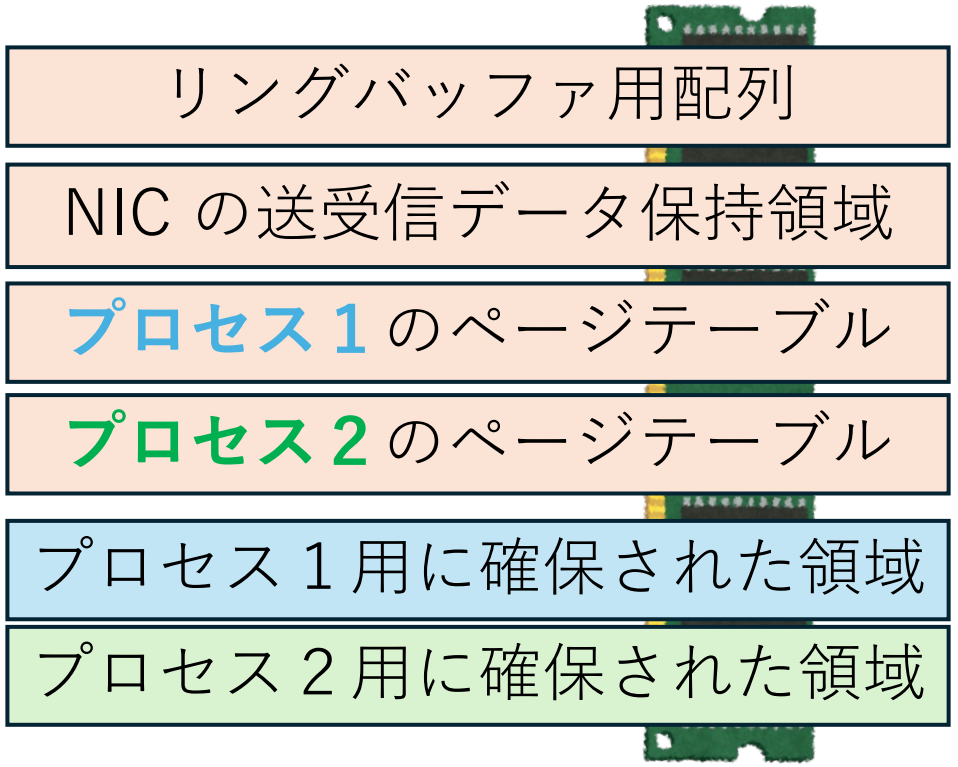
アクセス可能

アクセス可能



カーネル

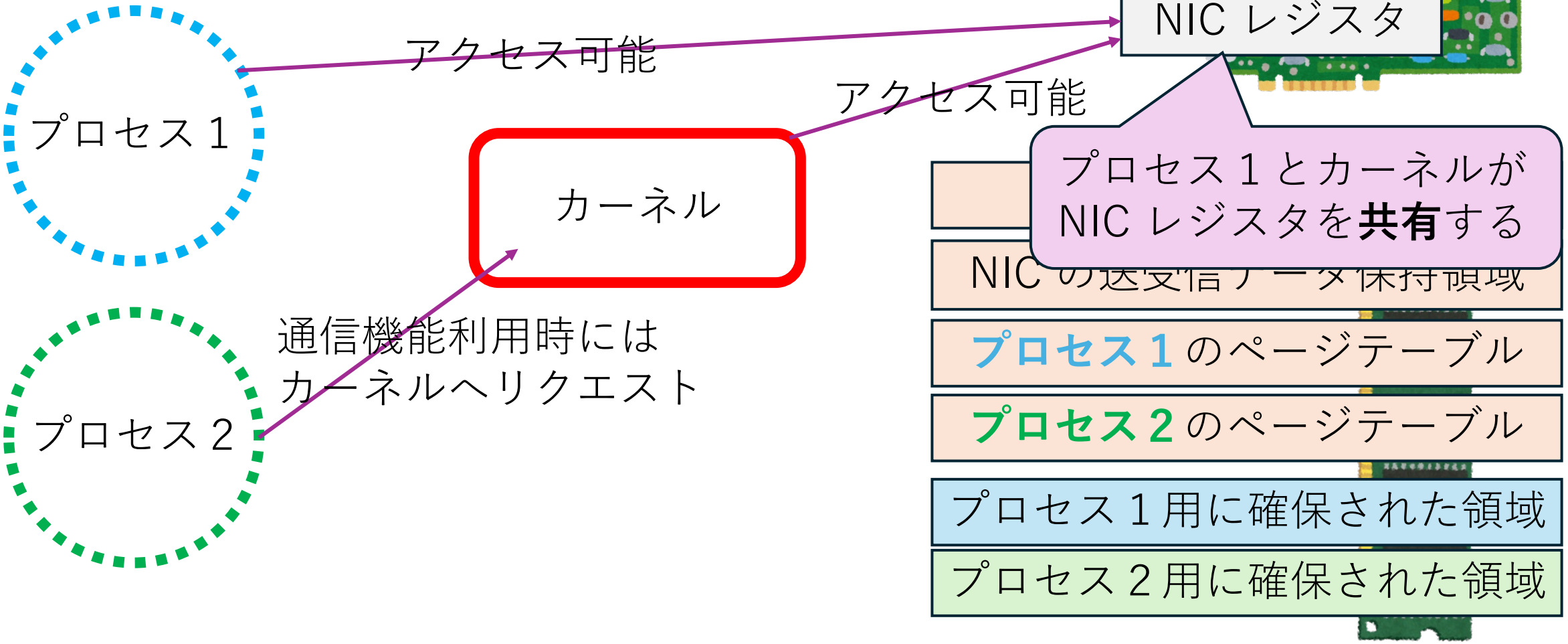
通信機能利用時には
カーネルへリクエスト



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

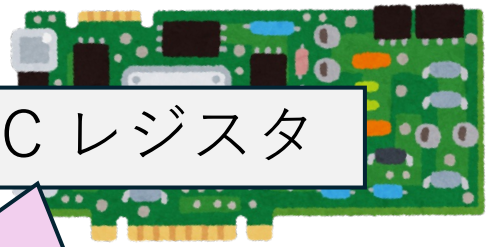
推奨されない（基本的に採用されない）設定例 1



セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

推奨されない（基本的に採用されない）設定例 1



NIC レジスタ

プロセス１

アクセス可能

アクセス可能

カーネル

利用時には
へリクエスト

プロセス１とカーネルが
NIC レジスタを**共有**する

NIC の送受信データ保持領域

プロセス１のページテーブル

プロセス２のページテーブル

プロセス１用に確保された領域

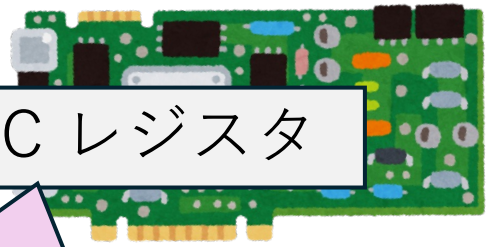
プロセス２用に確保された領域

<u>何故推奨されないか？</u>
プロセス１上のアプリを
信頼しない想定だから

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

推奨されない（基本的に採用されない）設定例 1



NIC レジスタ

アクセス可能

アクセス可能

プロセス１

カーネル

プロセス１とカーネルが
NIC レジスタを**共有**する

NIC の送受信データ保持領域

プロセス１のページテーブル

カーネルのページテーブル

プロセス１用に確保された領域

プロセス２用に確保された領域

利用時には

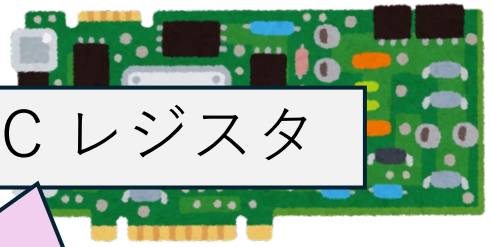
何故推奨されないか？
プロセス１上のアプリを
信頼しない想定だから

バグや脆弱性によるカーネルや
他の要素への波及を懸念するため

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

推奨されない（基本的に採用されない）設定例 1



NIC レジスタ

アクセス可能

アクセス可能

プロセス１

カーネル

プロセス１とカーネルが
NIC レジスタを**共有**する

NIC の送受信データ保持領域

ページテーブル

ページテーブル

保された領域

保された領域

利用時には

何故推奨されないか？
プロセス１上のアプリを
信頼しない想定だから

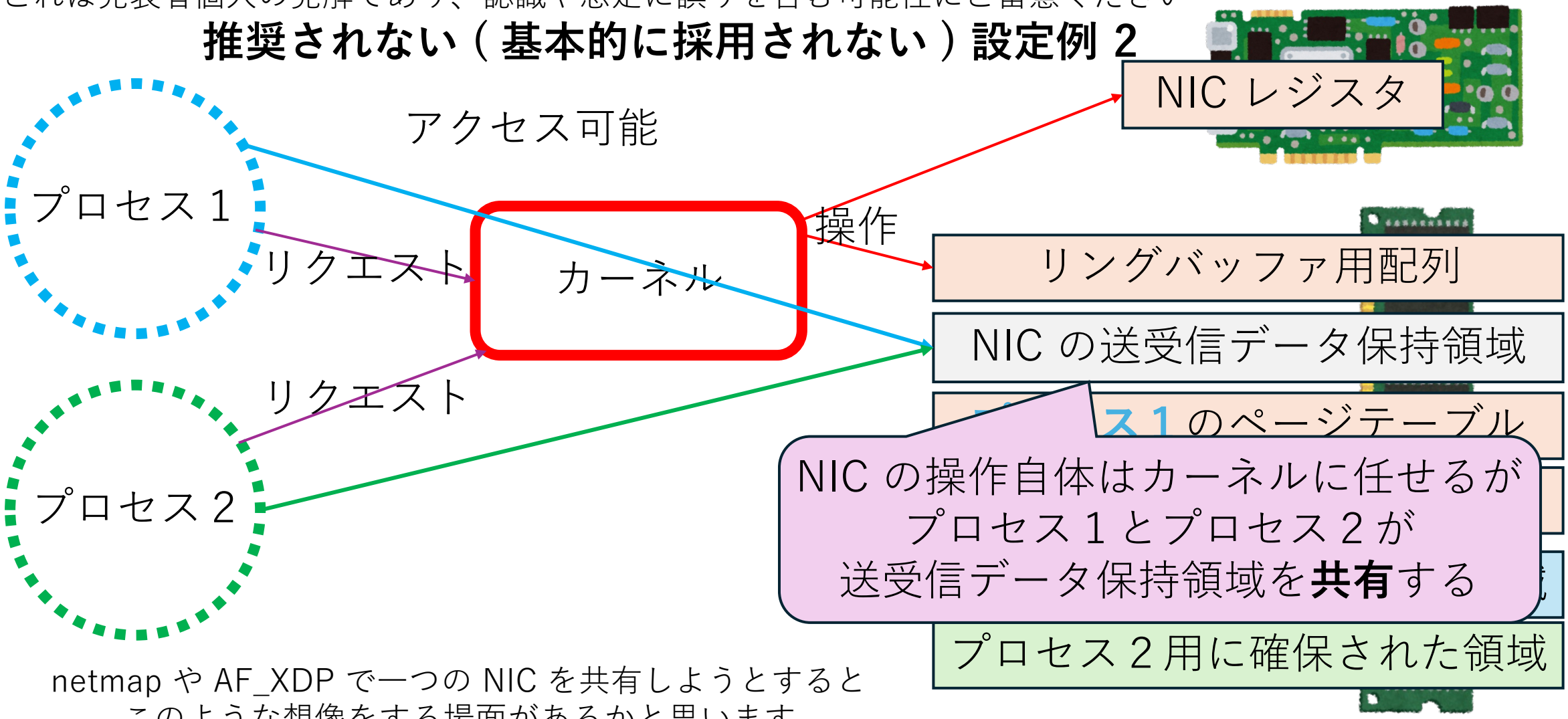
バグや脆弱性によるカーネルや
他の要素への波及を懸念するため

プロセス１上のアプリをカーネルと同様に信頼するという
想定を適用するならこの設定には問題がないことになります

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

推奨されない（基本的に採用されない）設定例 2

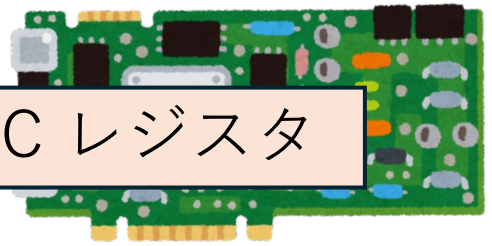


netmap や AF_XDP で一つの NIC を共有しようとすると
このような想像をする場面があるかと思います

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

推奨されない（基本的に採用されない）設定例 2



NIC レジスタ

アクセス可能

プロセス１

操作

リングバッファ用配列

NIC の送受信データ保持領域

ス１のページテーブル

<u>何故推奨されないか？</u>

同じ送受信データ保持領域に
プロセス１とプロセス２宛のデータが
受信される可能性があり
両プロセスからそれが見えるので
バグや脆弱性の影響の波及を
懸念するなら非推奨です

NIC の操作自体はカーネルに任せるが
プロセス１とプロセス２が
送受信データ保持領域を**共有**する

プロセス２用に確保された領域

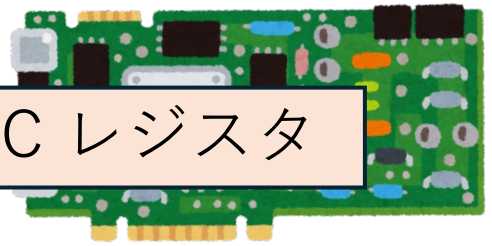
netmap や AF_XDP で一つの NIC を共有しようとすると
このような想像をする場面があるかと思います

セキュリティについての考え方

これは発表者個人の見解であり、認識や想定に誤りを含む可能性にご留意ください

推奨されない（基本的に採用されない）設定例 2

プロセス１とプロセス２が同様に
信頼される想定なら問題なしです



NIC レジスタ

プロセス１

操作

リングバッファ用配列

NIC の送受信データ保持領域

プロセス１のページテーブル

何故推奨されないか？

同じ送受信データ保持領域に
プロセス１とプロセス２宛のデータが
受信される可能性があり
両プロセスからそれが見えるので
バグや脆弱性の影響の波及を
懸念するなら非推奨です

NIC の操作自体はカーネルに任せるが
プロセス１とプロセス２が
送受信データ保持領域を**共有**する

プロセス２用に確保された領域

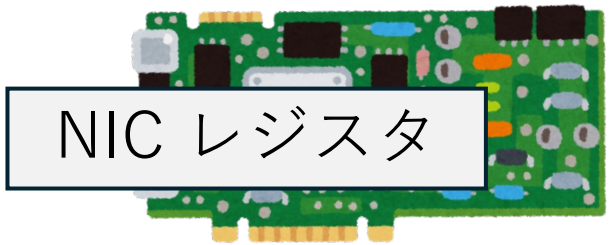
netmap や AF_XDP で一つの NIC を共有しようとすると
このような想像をする場面があるかと思います

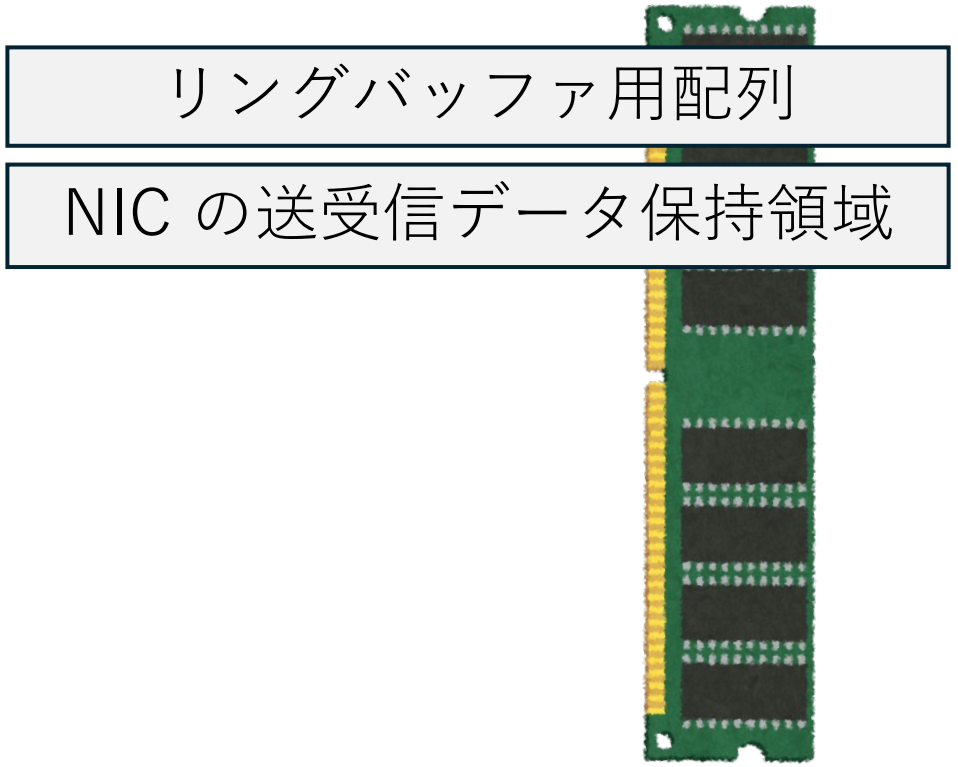
一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする

プロセス１

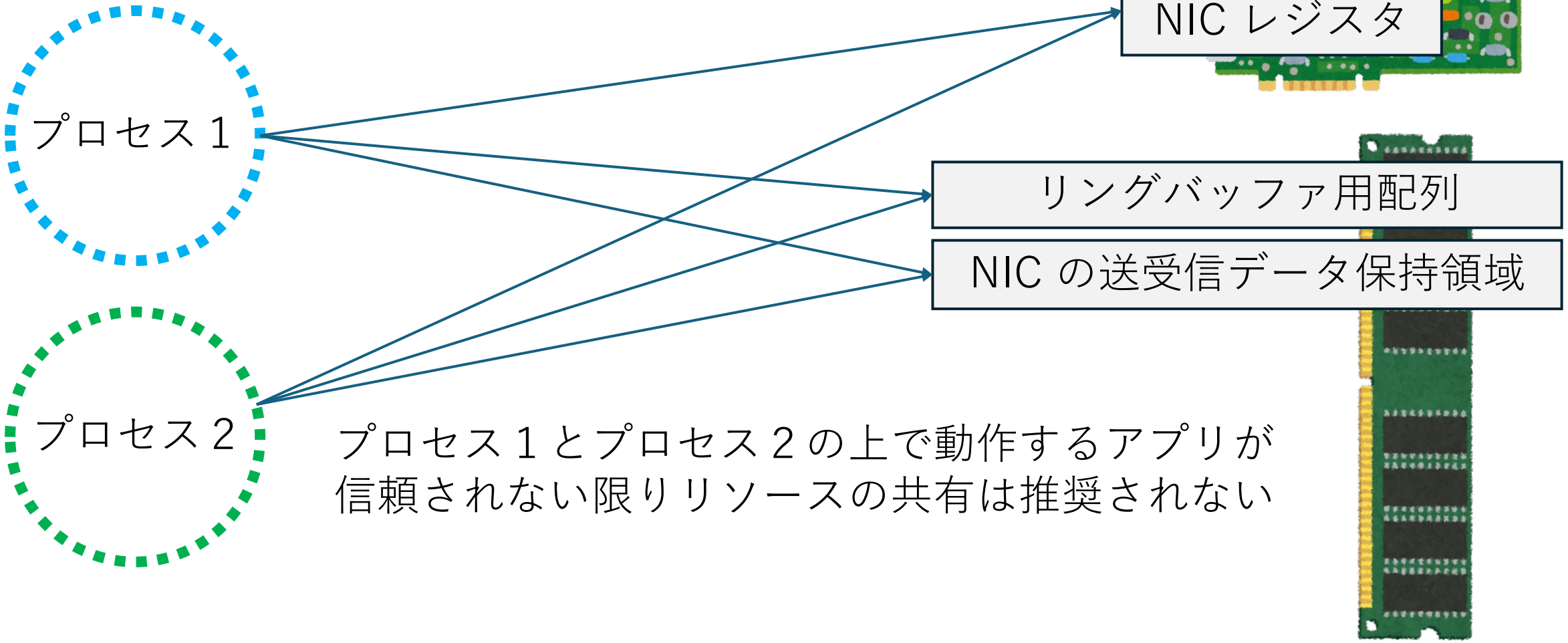
プロセス２





一つの NIC を共有する方法

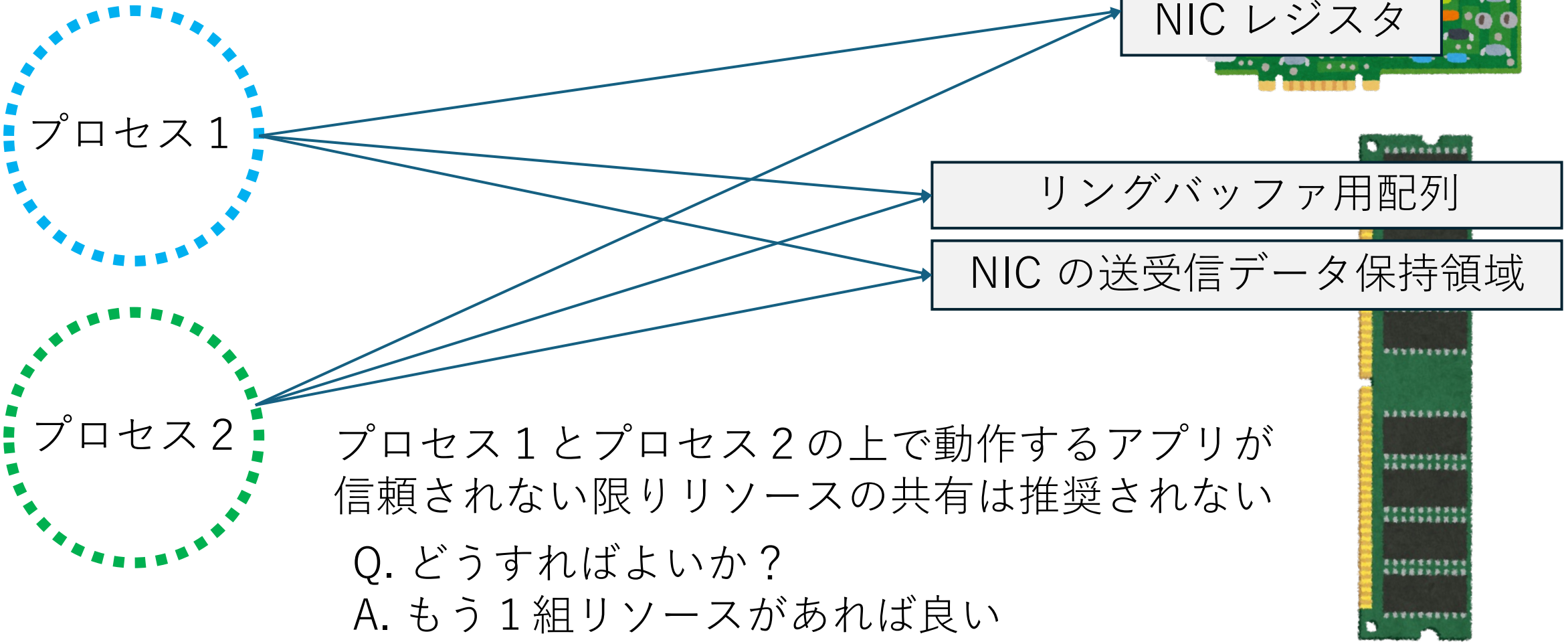
複数の要素が同一のリソースへアクセスしないようにする



プロセス１とプロセス２の上で動作するアプリが
信頼されない限りリソースの共有は推奨されない

一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする

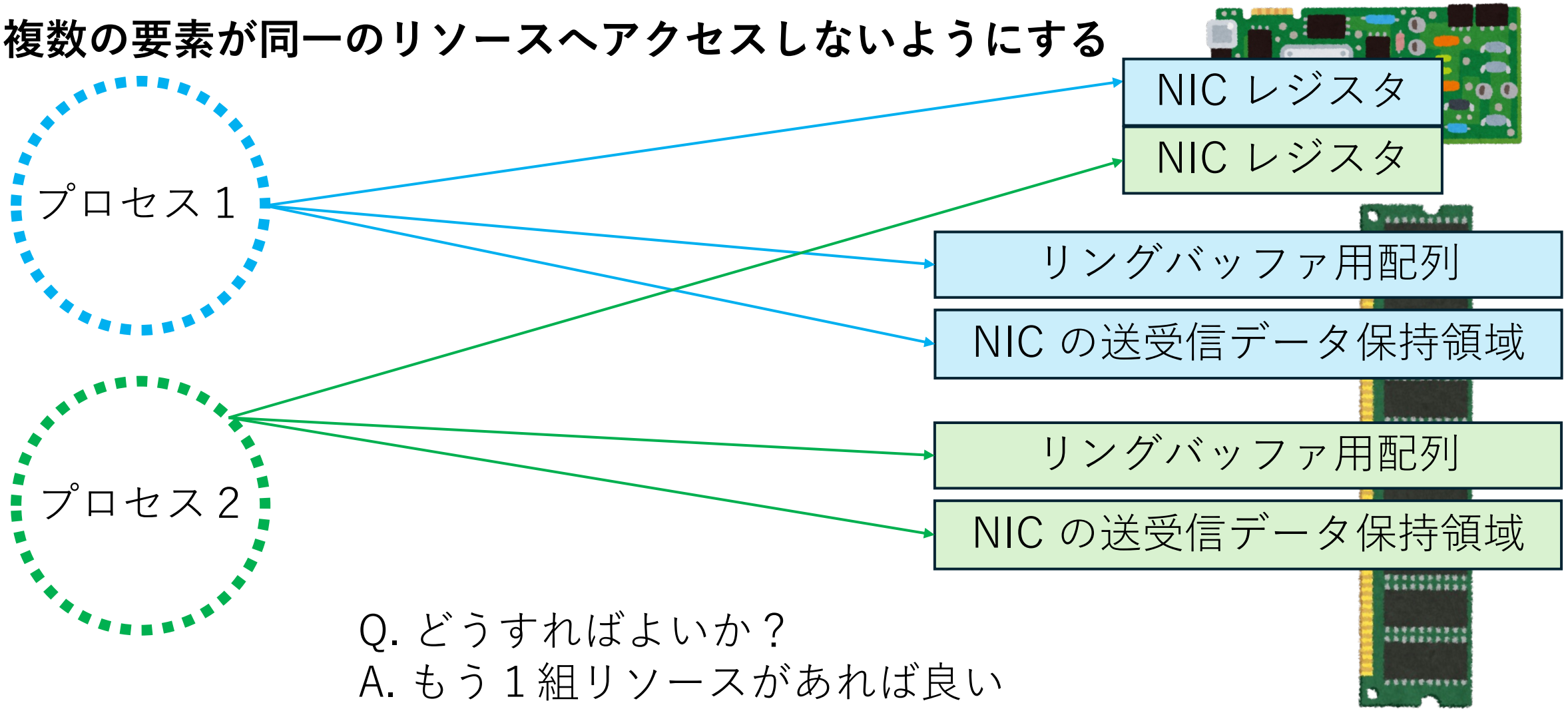


プロセス１とプロセス２の上で動作するアプリが
信頼されない限りリソースの共有は推奨されない

Q. どうすればよいか？
A. もう１組リソースがあれば良い

一つの NIC を共有する方法

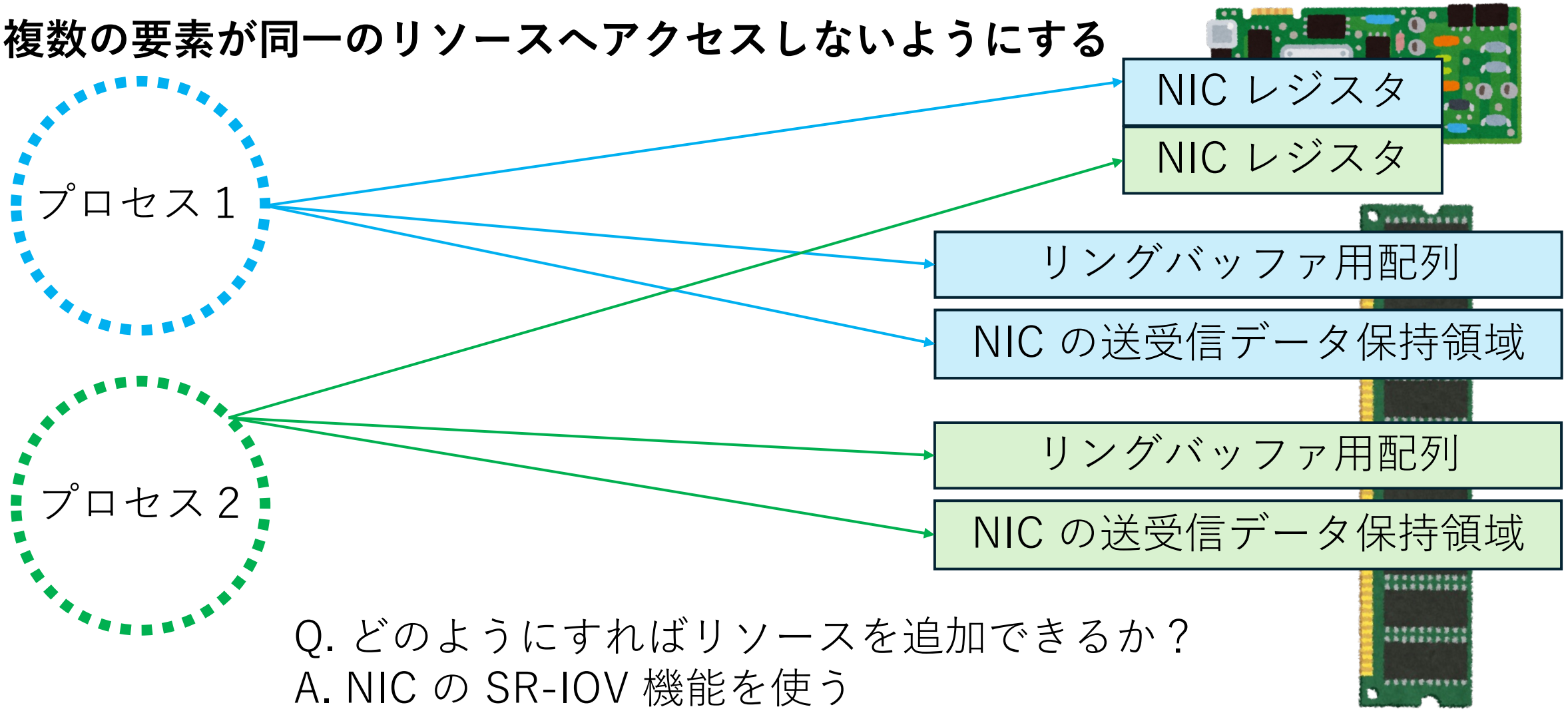
複数の要素が同一のリソースへアクセスしないようにする



Q. どうすればよいか？
A. もう１組リソースがあれば良い

一つの NIC を共有する方法

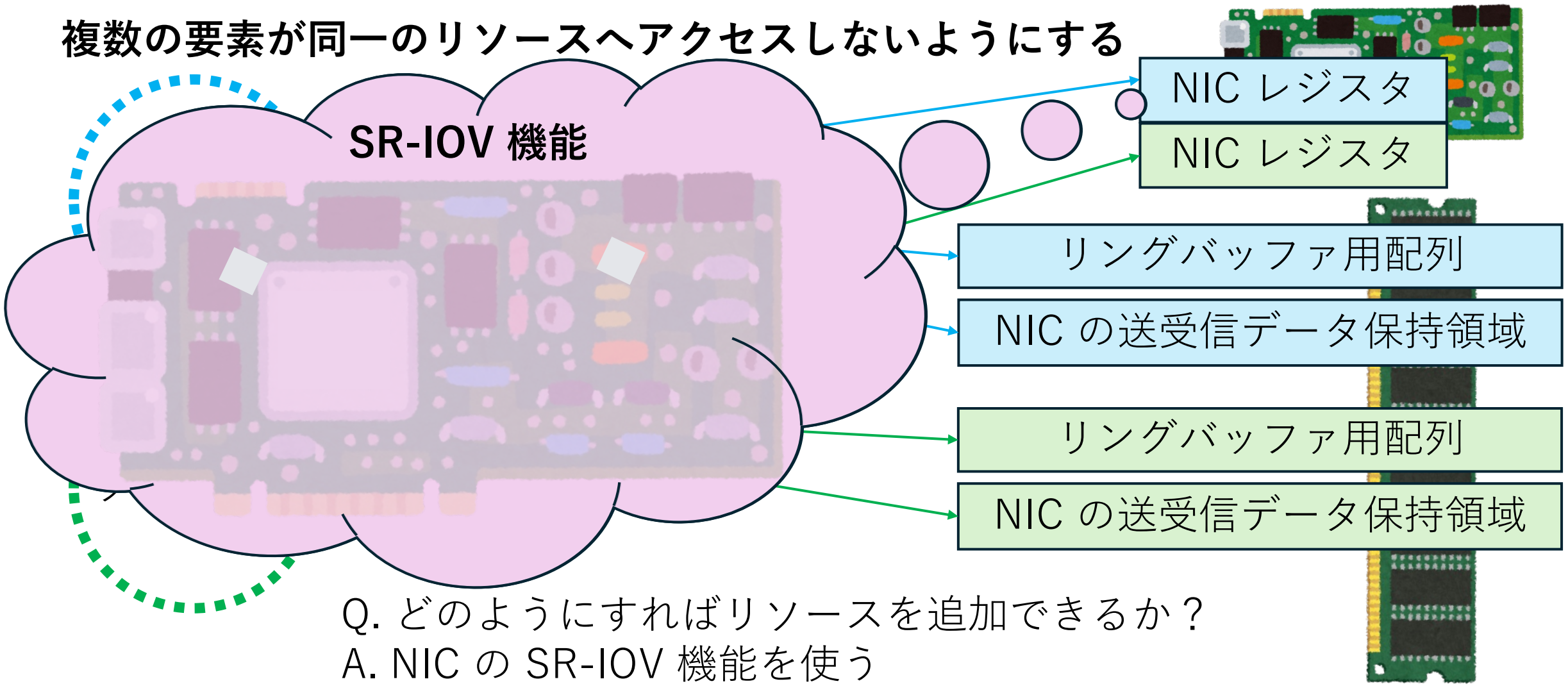
複数の要素が同一のリソースへアクセスしないようにする



Q. どのようにすればリソースを追加できるか？
A. NIC の SR-IOV 機能を使う

一つの NIC を共有する方法

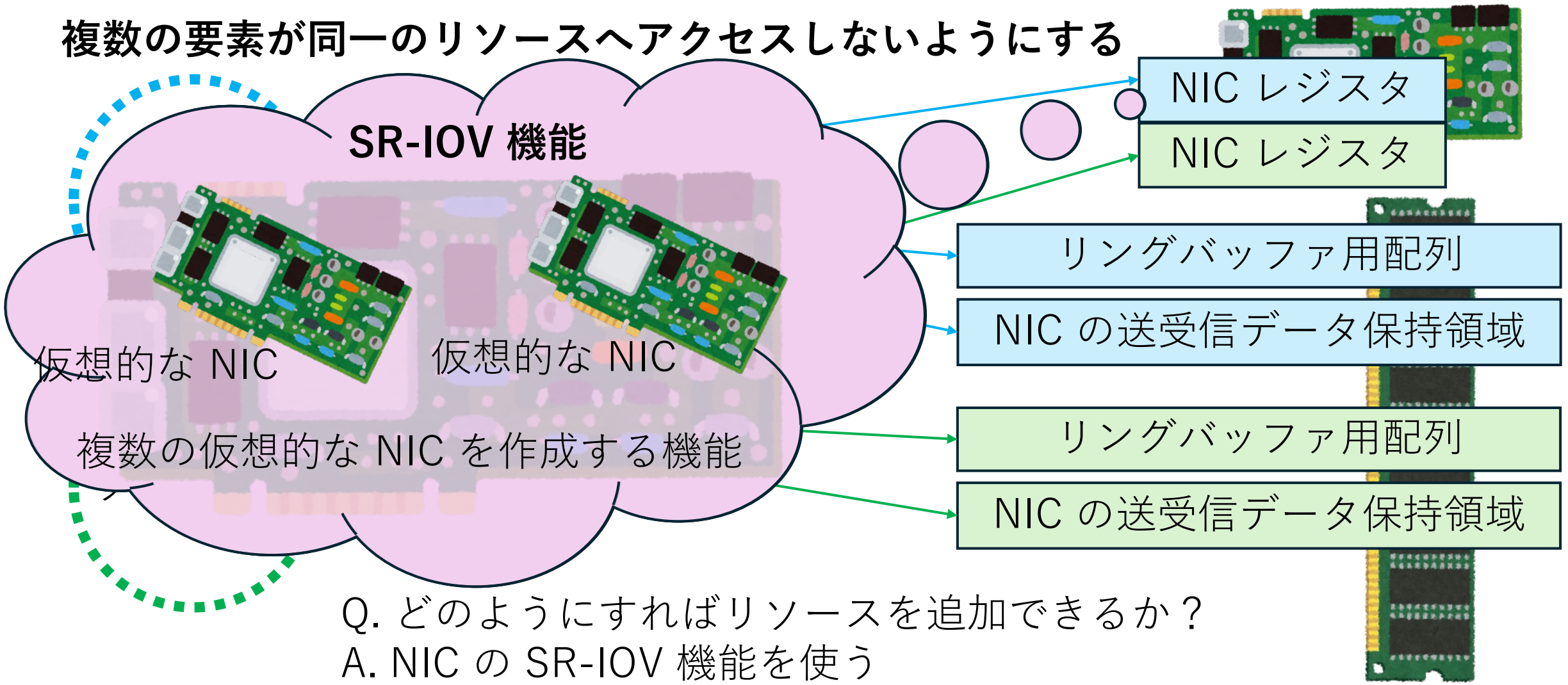
複数の要素が同一のリソースへアクセスしないようにする



Q. どのようにすればリソースを追加できるか？
A. NIC の SR-IOV 機能を使う

一つの NIC を共有する方法

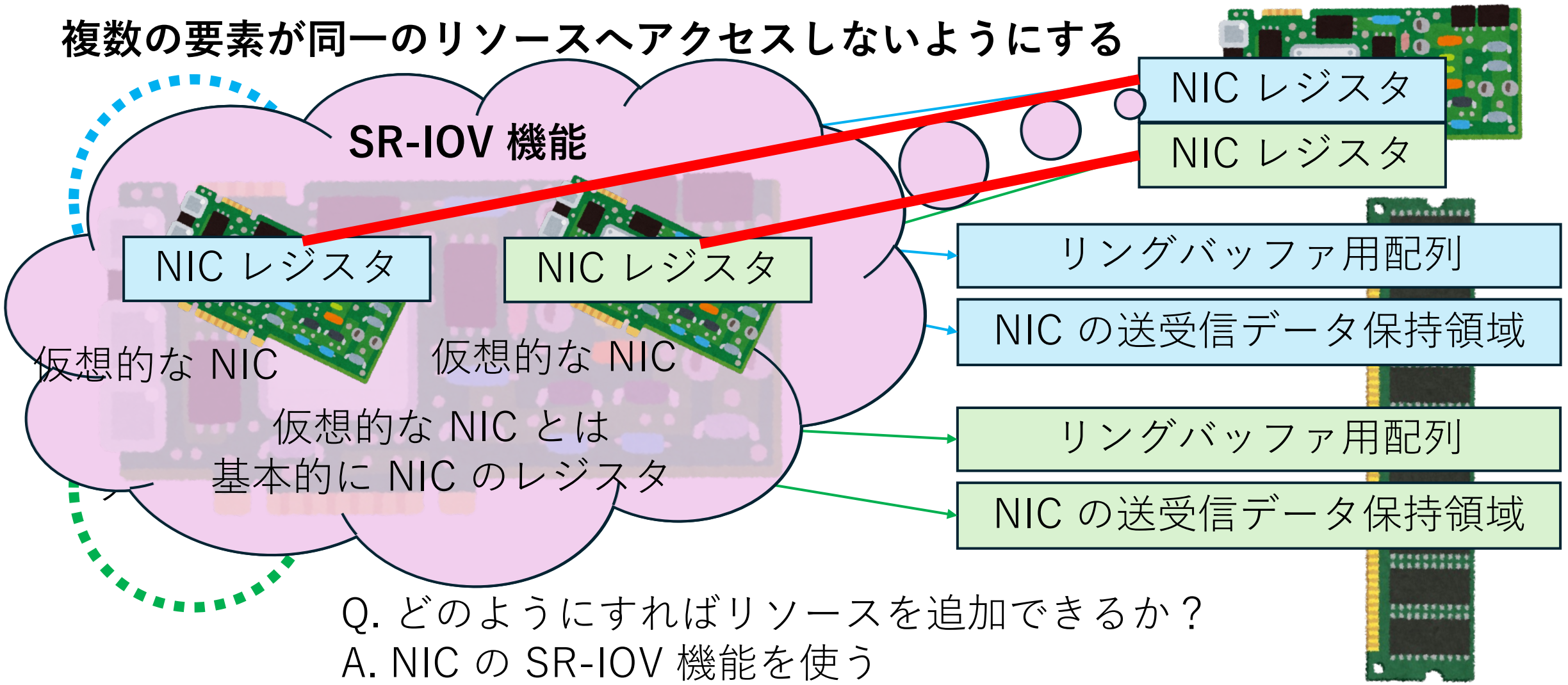
複数の要素が同一のリソースへアクセスしないようにする



Q. どのようにすればリソースを追加できるか？
A. NIC の SR-IOV 機能を使う

一つの NIC を共有する方法

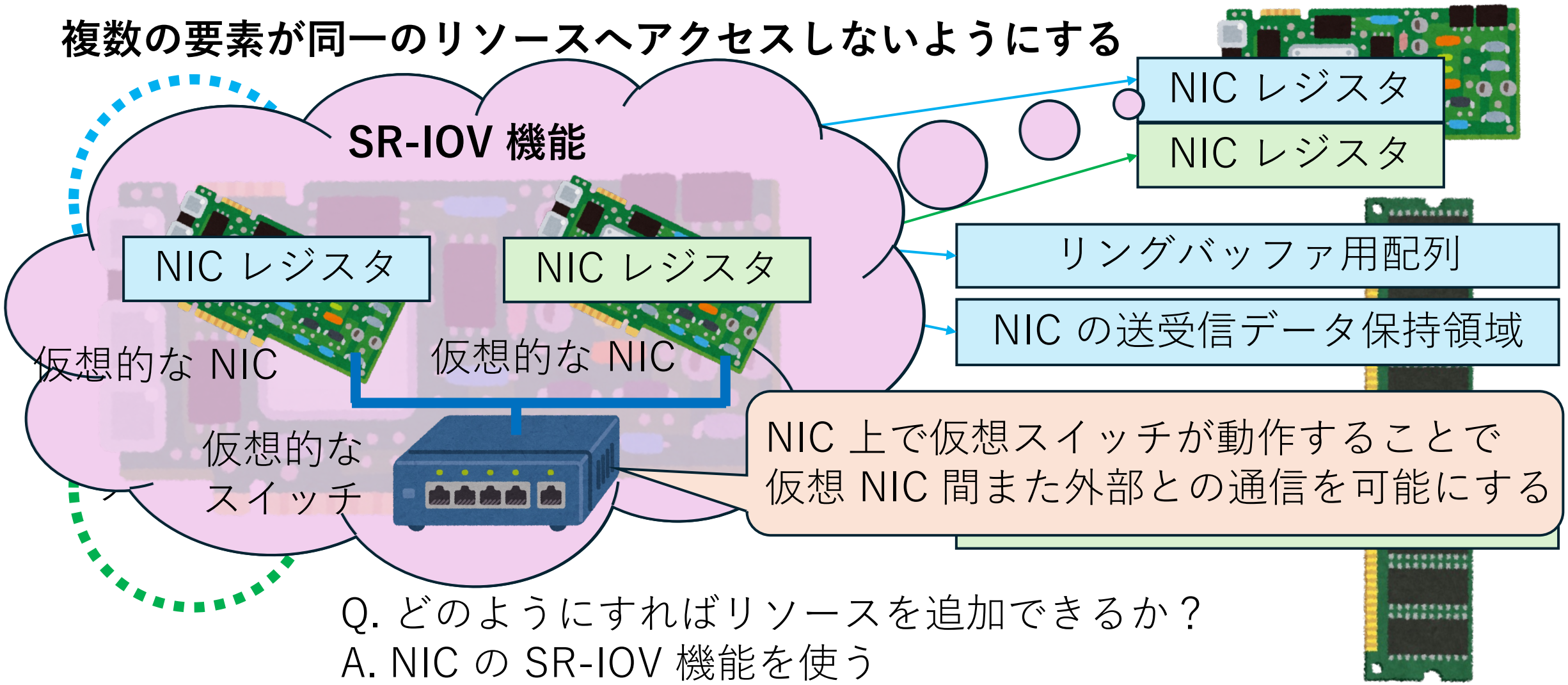
複数の要素が同一のリソースへアクセスしないようにする



Q. どのようにすればリソースを追加できるか？
A. NIC の SR-IOV 機能を使う

一つの NIC を共有する方法

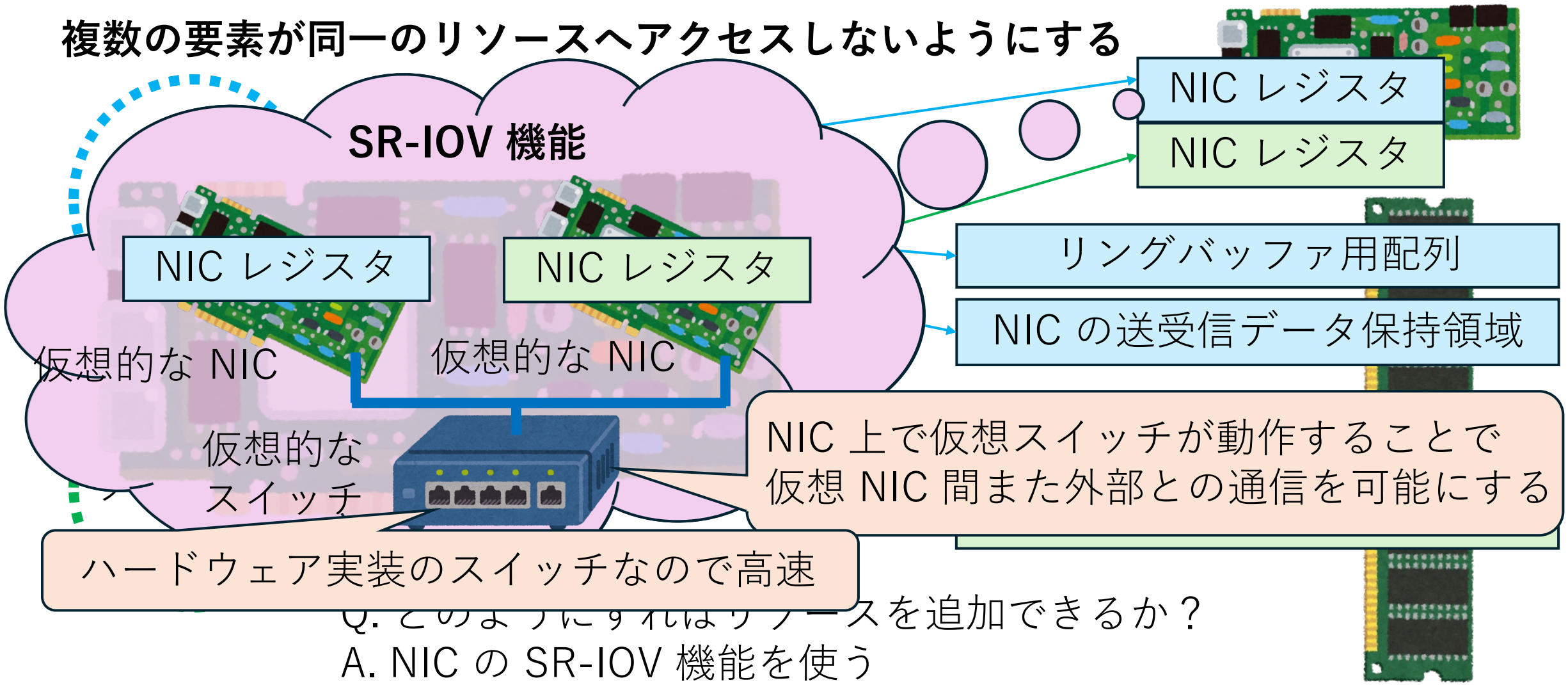
複数の要素が同一のリソースへアクセスしないようにする



Q. どのようにすればリソースを追加できるか？
A. NIC の SR-IOV 機能を使う

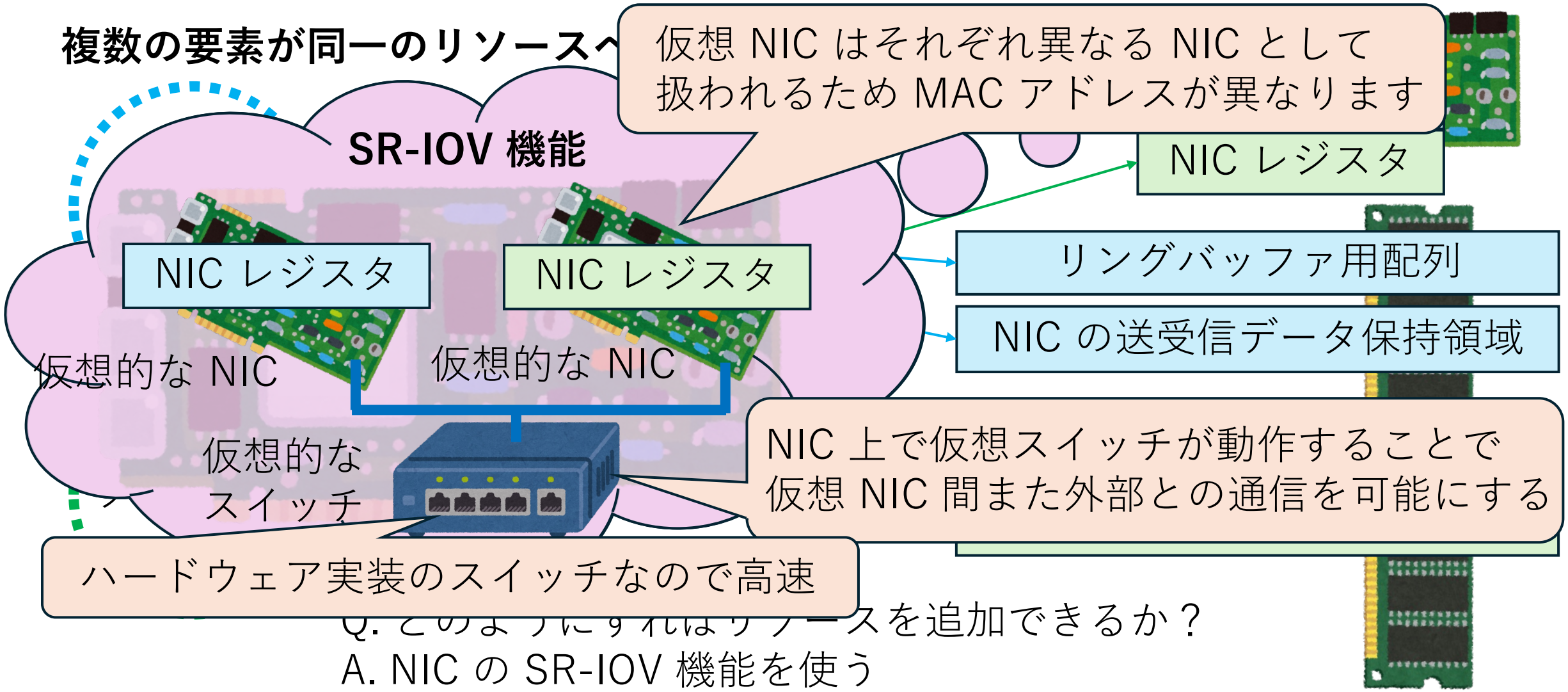
一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする



Q. どのようにすれはリソースを追加できるか？
A. NIC の SR-IOV 機能を使う

一つの NIC を共有する方法



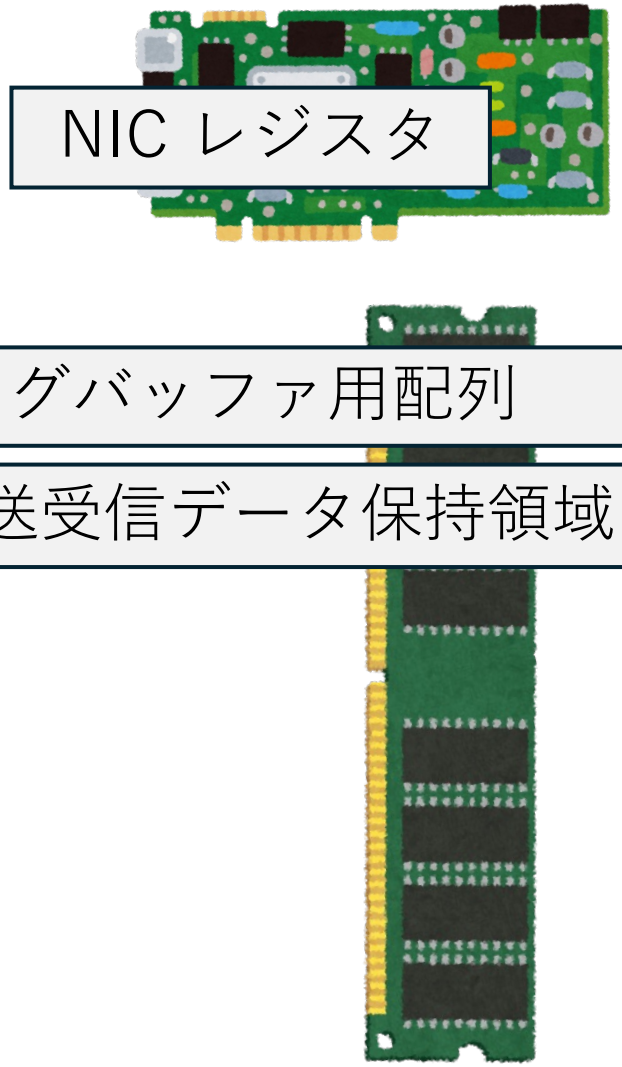
Q. どのようにすればリソースを追加できるか？
A. NIC の SR-IOV 機能を使う

一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする

プロセス 1

プロセス 2



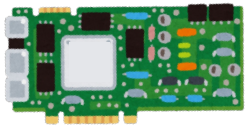
Q. NIC が SR-IOV に対応していなかったら？
A. 仮想 NIC をソフトウェアで実装する

一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする

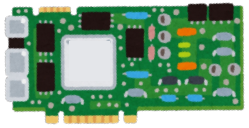
プロセス 1

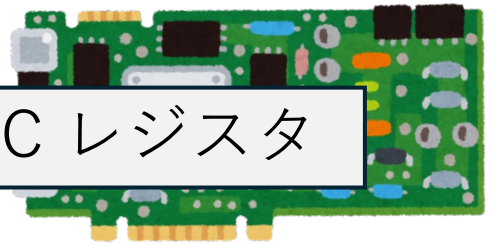
仮想 NIC



プロセス 2

仮想 NIC





NIC レジスタ

リングバッファ用配列

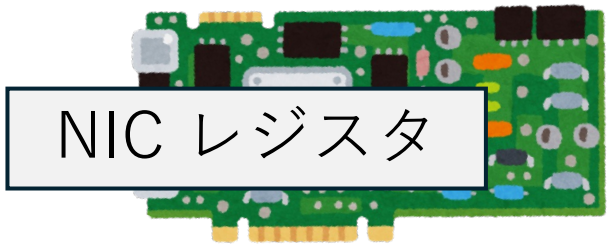
NIC の送受信データ保持領域

Q. NIC が SR-IOV に対応していなかったら？

A. 仮想 NIC をソフトウェアで実装する

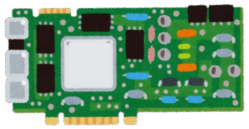
一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする



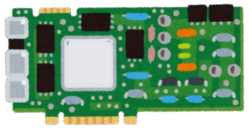
プロセス 1

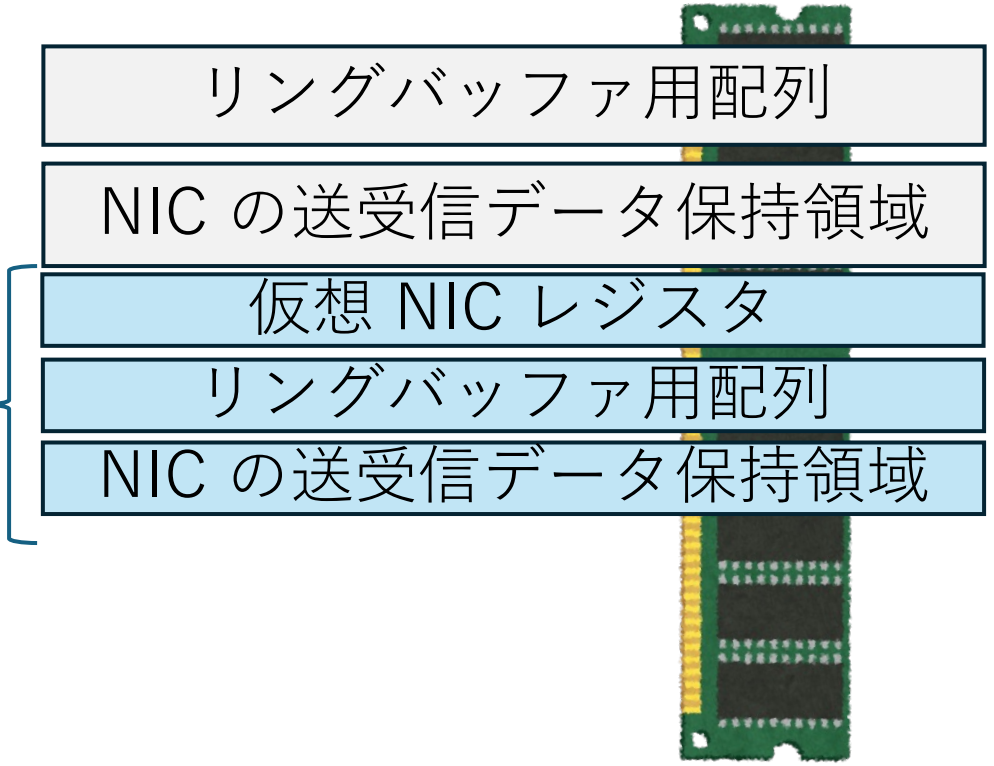
仮想 NIC



プロセス 2

仮想 NIC

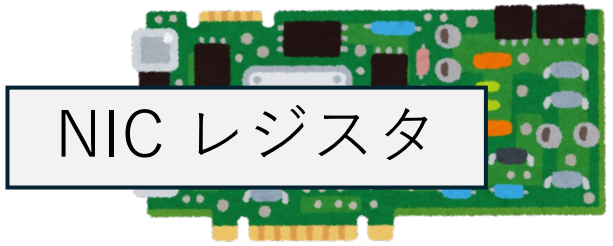




Q. NIC が SR-IOV に対応していなかったら？
A. 仮想 NIC をソフトウェアで実装する

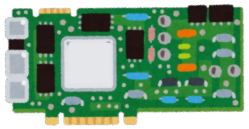
一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする



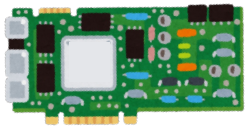
プロセス 1

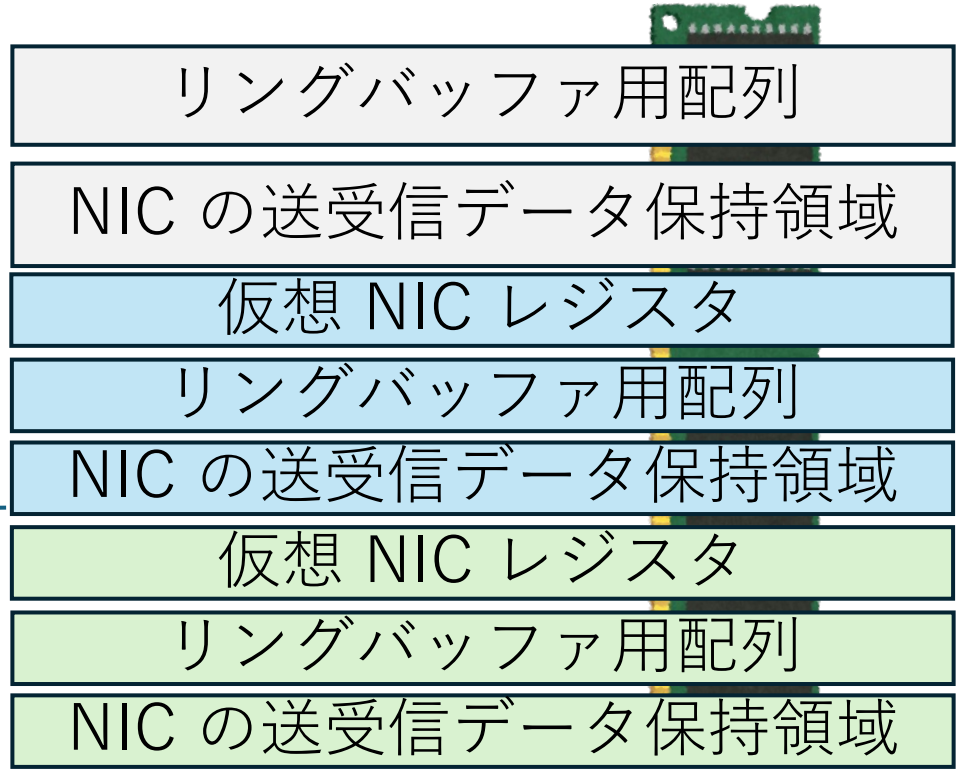
仮想 NIC



プロセス 2

仮想 NIC

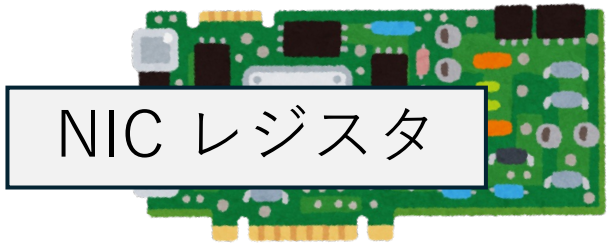




Q. NIC が SR-IOV に対応していなかったら？
A. 仮想 NIC をソフトウェアで実装する

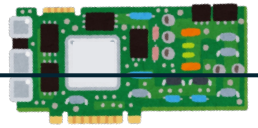
一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする



プロセス 1

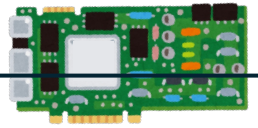
仮想 NIC



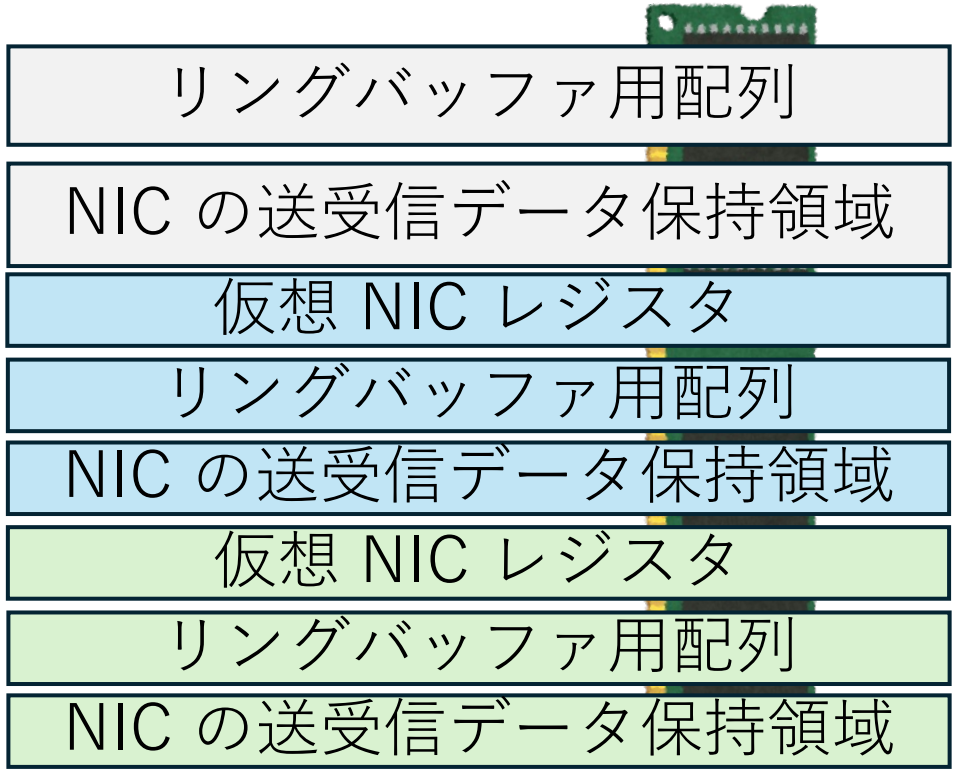
エミュレータ実装

プロセス 2

仮想 NIC



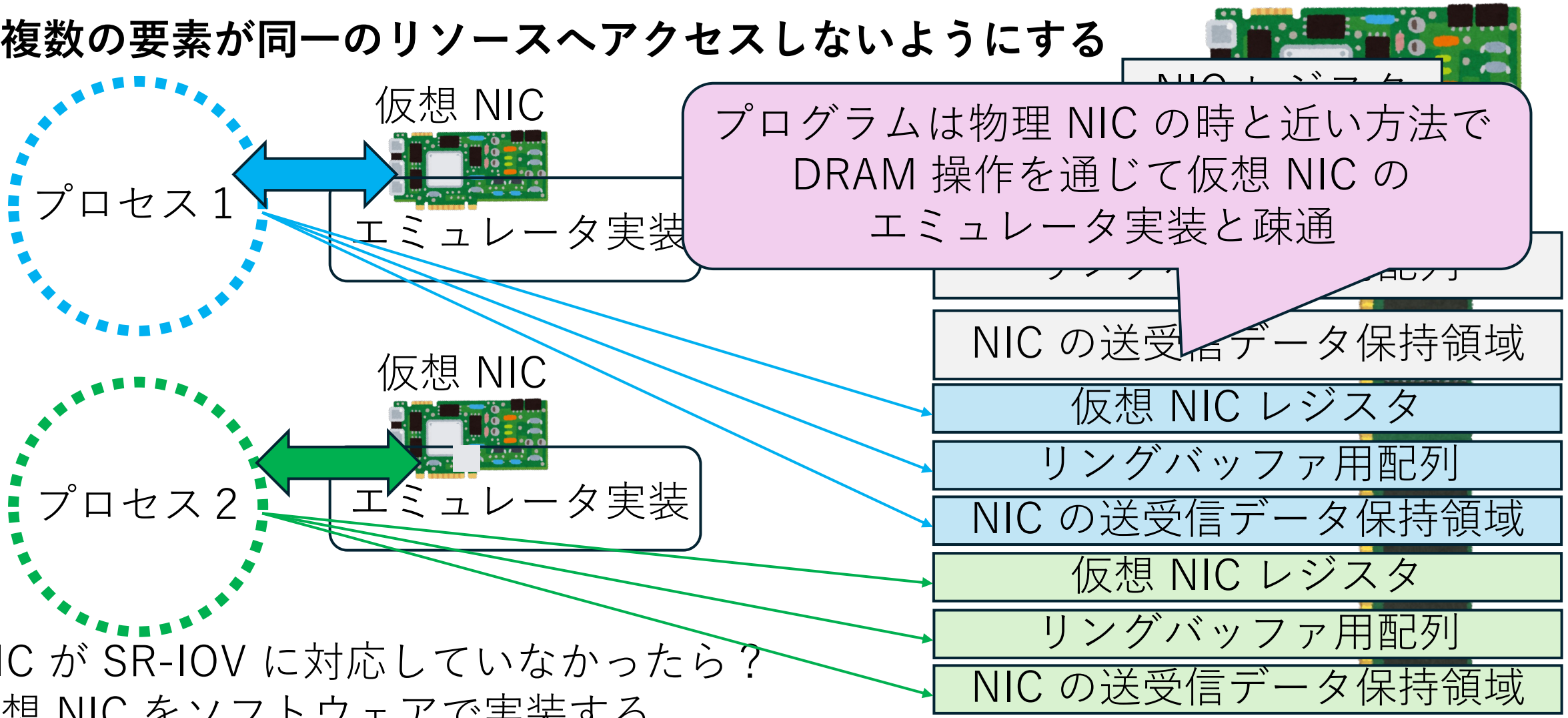
エミュレータ実装



Q. NIC が SR-IOV に対応していなかったら？
A. 仮想 NIC をソフトウェアで実装する

一つの NIC を共有する方法

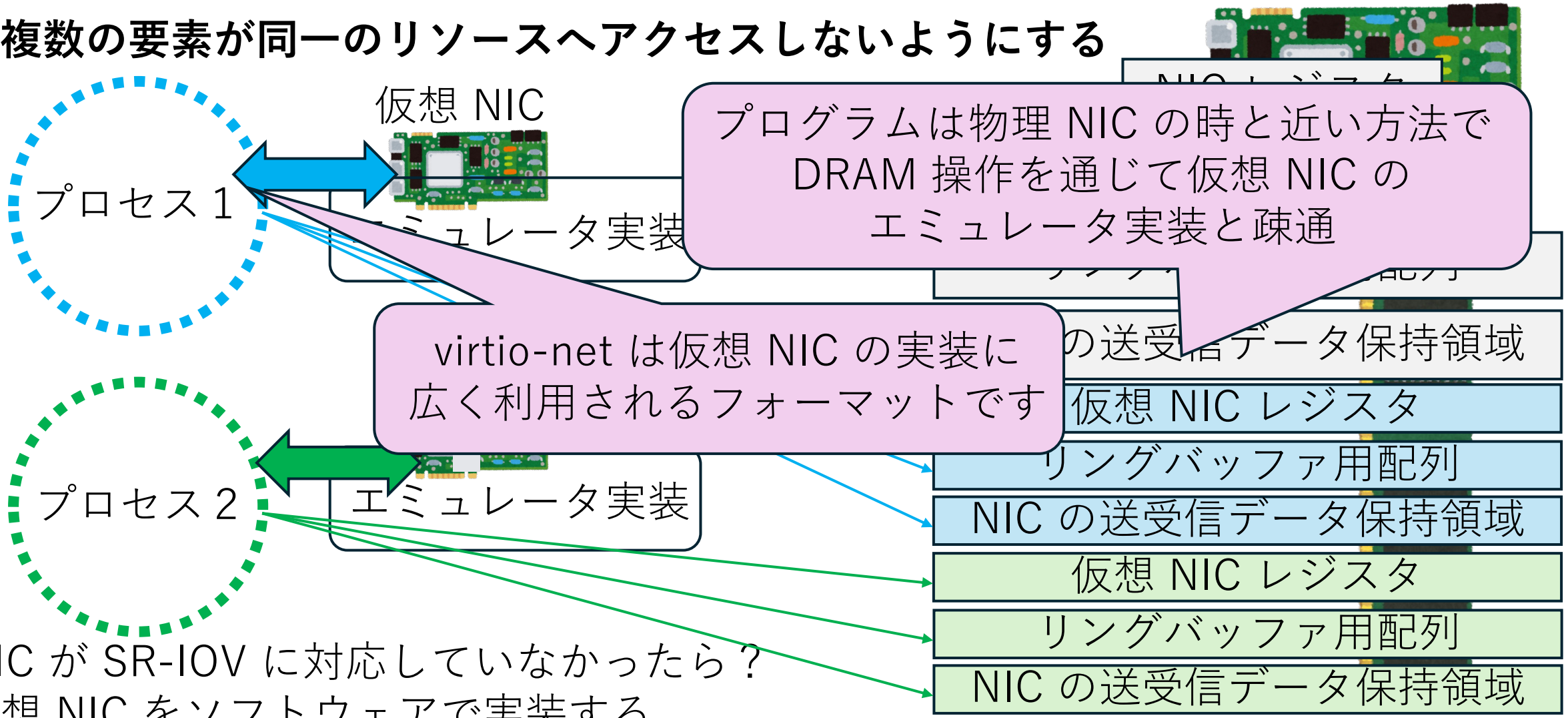
複数の要素が同一のリソースへアクセスしないようにする



Q. NIC が SR-IOV に対応していなかったら？
A. 仮想 NIC をソフトウェアで実装する

一つの NIC を共有する方法

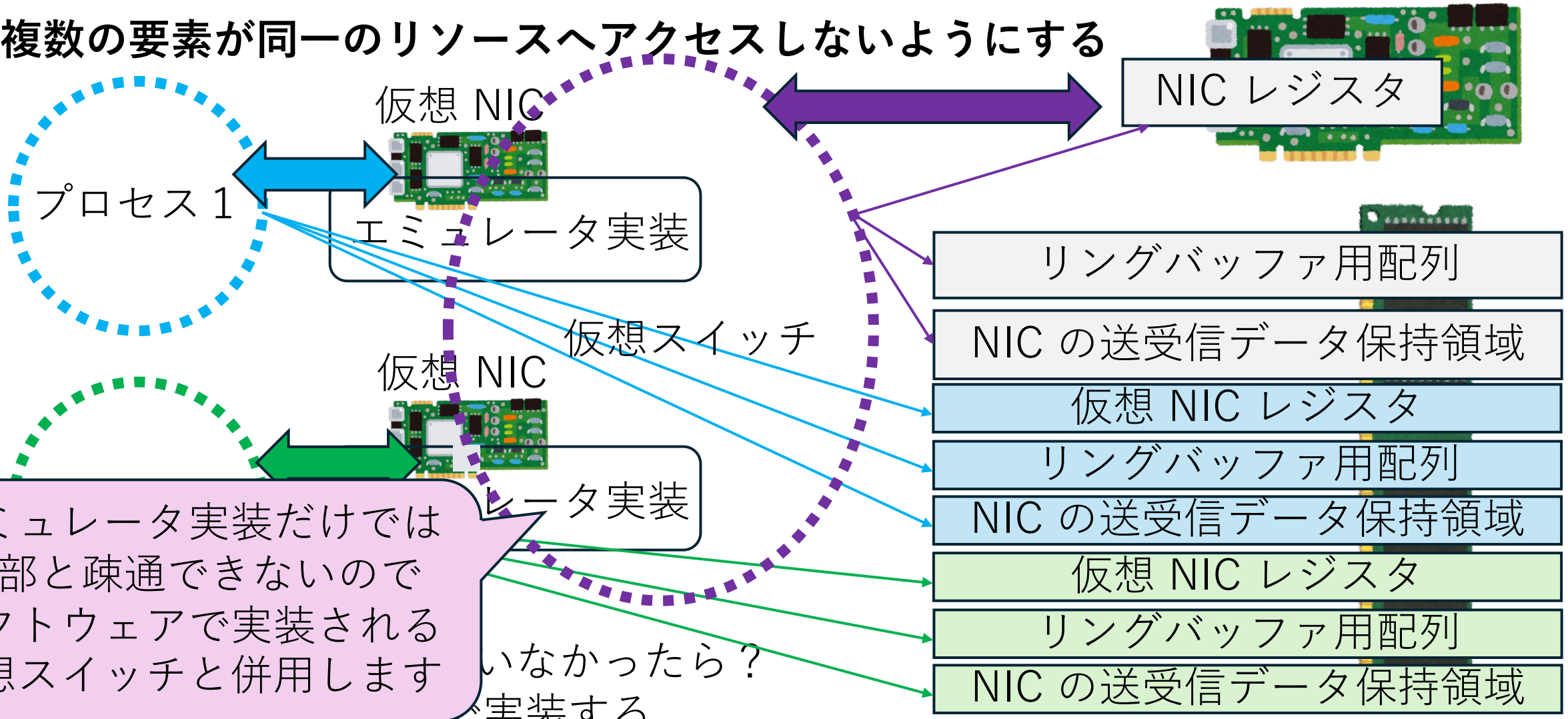
複数の要素が同一のリソースへアクセスしないようにする



Q. NIC が SR-IOV に対応していなかったら？
A. 仮想 NIC をソフトウェアで実装する

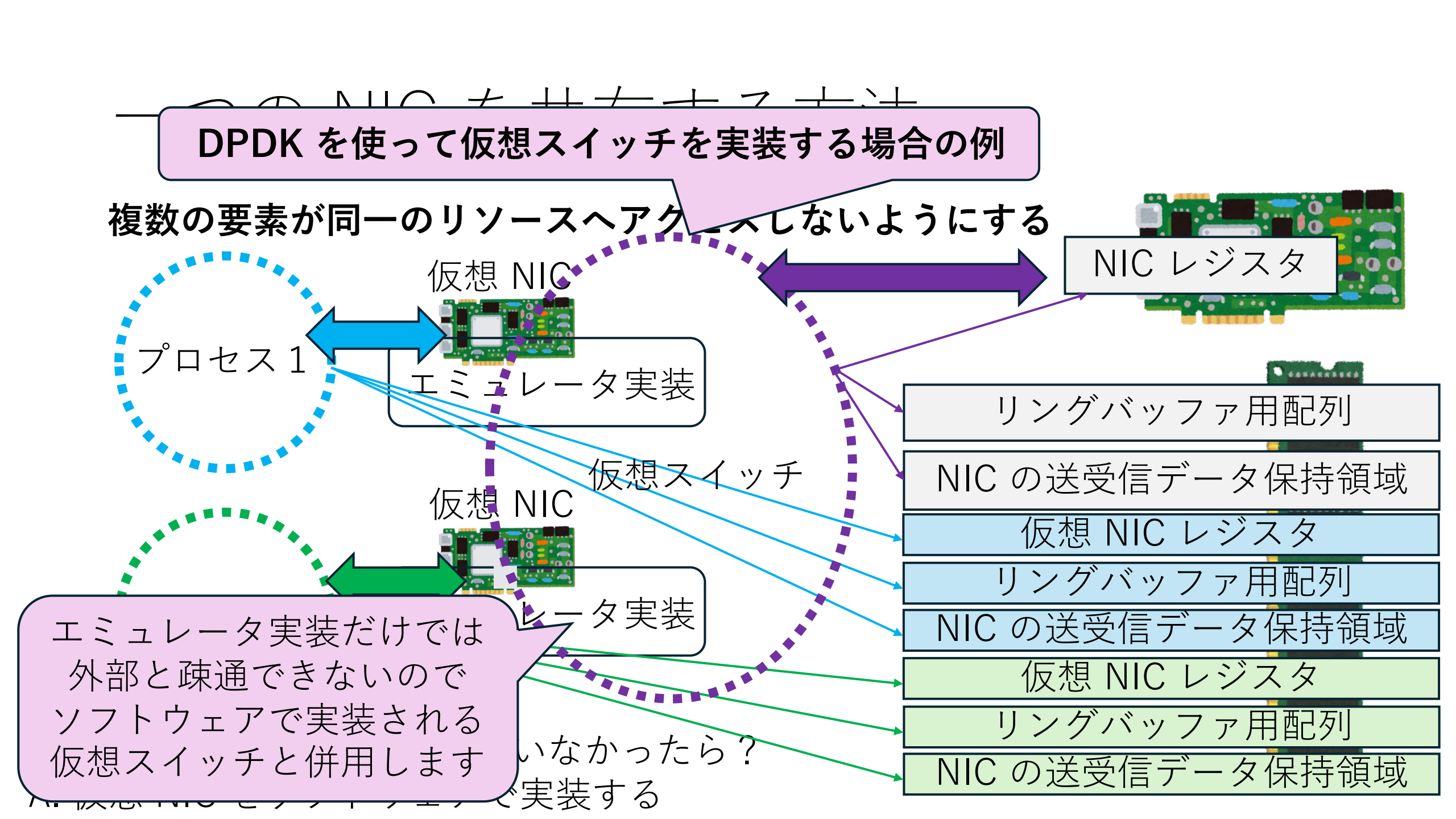
一つの NIC を共有する方法

複数の要素が同一のリソースへアクセスしないようにする



DPDK を使って仮想スイッチを実装する場合の例

複数の要素が同一のリソースへアクセスしないようにする



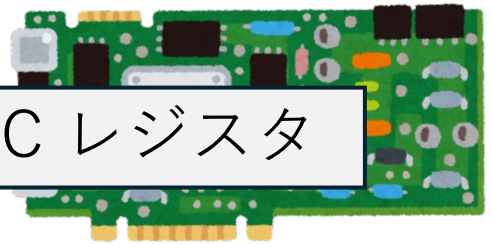
エミュレータ実装だけでは
外部と疎通できないので
ソフトウェアで実装される
仮想スイッチと併用します

いなかったら？

で実装する

DPDK を使って仮想スイッチを実装する場合の例

複数の要素が同一のリソースへアクセスしないようにする



NIC レジスタ

プロセス 1

仮想 NIC

エミュレータ実装

vhost-user という実装

仮想スイッチ

仮想 NIC

エミュレータ実装

エミュレータ実装だけでは
外部と疎通できないので
ソフトウェアで実装される
仮想スイッチと併用します

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

いなかったら？

で実装する

DPDK を使って仮想スイッチを実装する場合の例

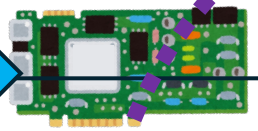
複数の要素が同一のリソースへアクセスしないようにする

NIC レジスタ

エミュレータ実装を動かす
仮想スイッチプロセスは
仮想 NIC 関連の DRAM 上領域に
アクセスできる必要があります

プロセス 1

仮想 NIC

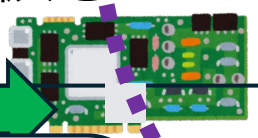


エミュレータ実装

vhost-user という実装

仮想スイッチ

仮想 NIC



エミュレータ実装

エミュレータ実装だけでは
外部と疎通できないので
ソフトウェアで実装される
仮想スイッチと併用します

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

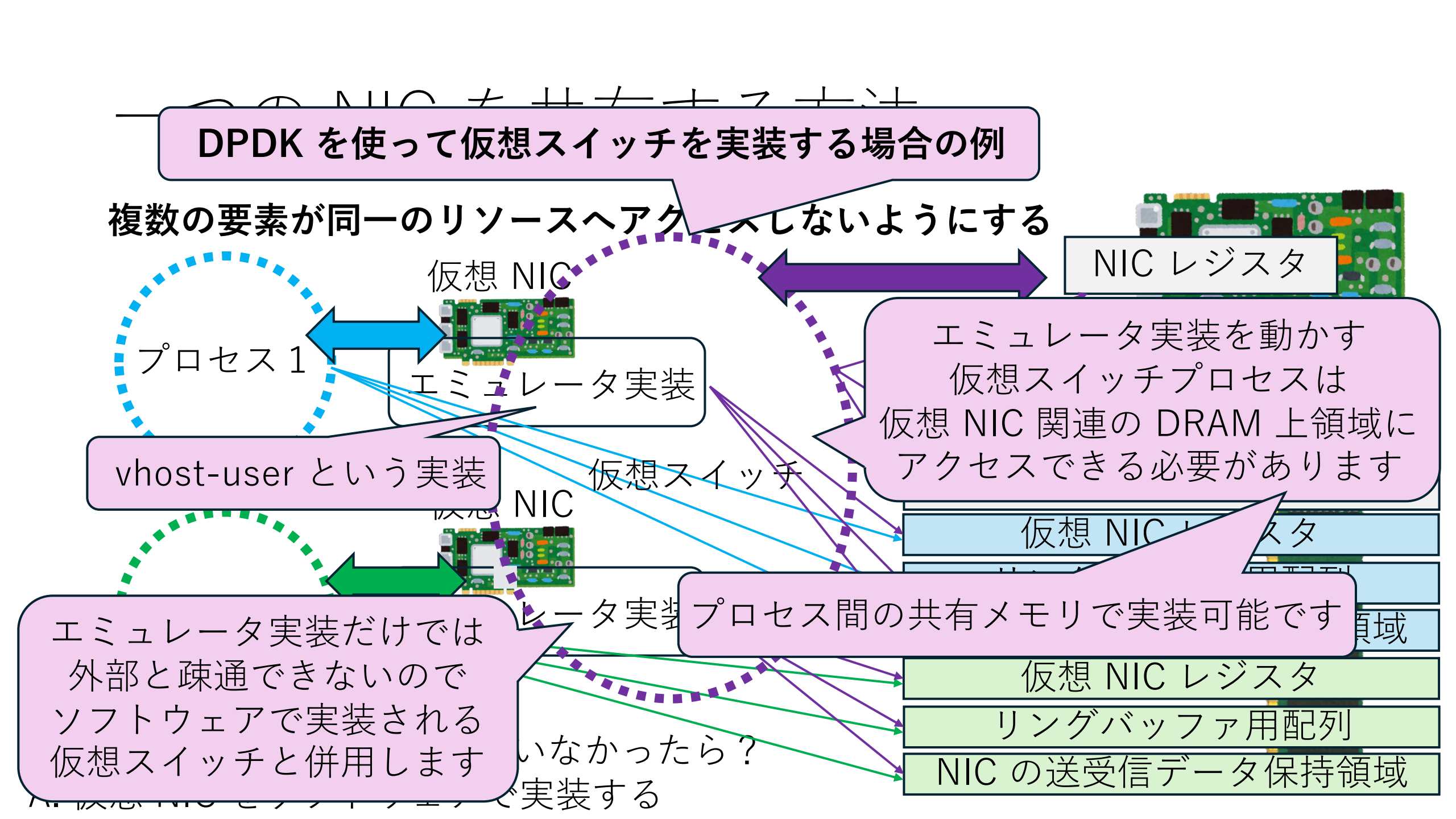
仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

いなかったら？

で実装する


DPDK を使って仮想スイッチを実装する場合の例
複数の要素が同一のリソースへアクセスしないようにする
仮想 NIC
プロセス 1
エミュレータ実装
NIC レジスタ
エミュレータ実装を動かす
仮想スイッチプロセスは
仮想 NIC 関連の DRAM 上領域に
アクセスできる必要があります
vhost-user という実装
仮想スイッチ
仮想 NIC
プロセス間の共有メモリで実装可能です
エミュレータ実装だけでは
外部と疎通できないので
ソフトウェアで実装される
仮想スイッチと併用します
仮想 NIC レジスタ
リングバッファ用配列
NIC の送受信データ保持領域

一般的な OS での運用

ちなみに、敢えて異なるプロセスが
同じ物理アドレスを参照できるように
設定するのがプロセス間の**共有メモリ**です

仮想	物理
0x0000	**0x4000**
0x1000	0x3000
…	
…	

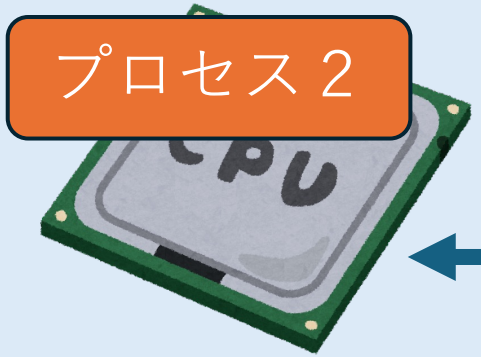
プロセス２用ページテーブル

仮想アドレスはプロセス１とプロセス２で
一致する必要はありません

仮想	物理
0x0000	
0x1000	0x2000
0x2000	**0x4000**
…	

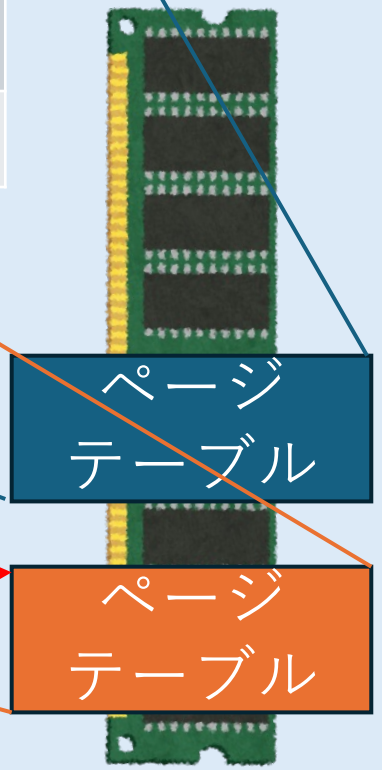
プロセス１用ページテーブル

プロセス２



MMU

この場合
物理メモリアドレス 0x4000 ~ 0x4fff が
プロセス１とプロセス２で共有されます



cr3:

仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

一般的な OS での運用

ちなみに、敢えて異なるプロセスが
同じ物理アドレスを参照できるように
設定するのがプロセス間の**共有メモリ**です

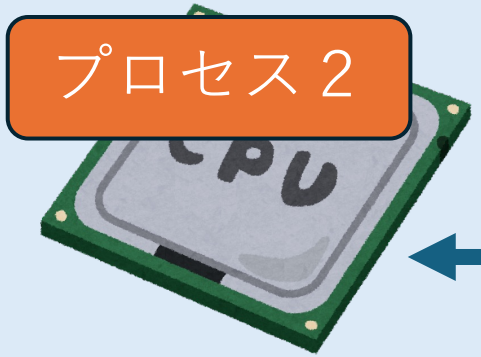
仮想	物理
0x0000	**0x4000**
0x1000	0x3000
…	
…	

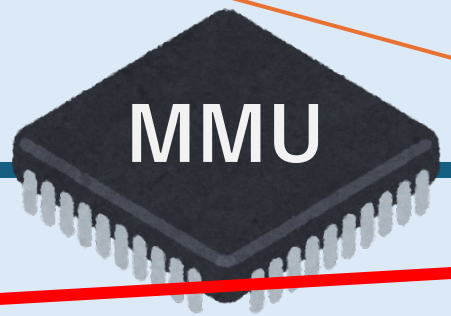
プロセス２用ページテーブル

仮想アドレスはプロセス１とプロセス２で
一致する必要はありません

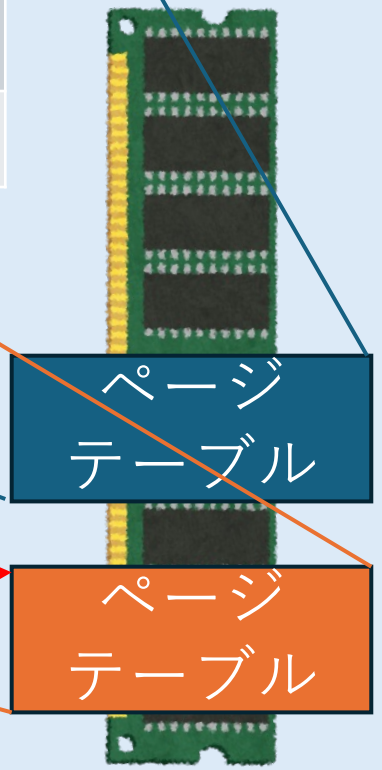
仮想	物理
0x0000	
0x1000	0x2000
0x2000	**0x4000**
…	

プロセス１用ページテーブル





この場合
物理メモリアドレス 0x4000 ~ 0x4fff が
プロセス１とプロセス２で共有されます

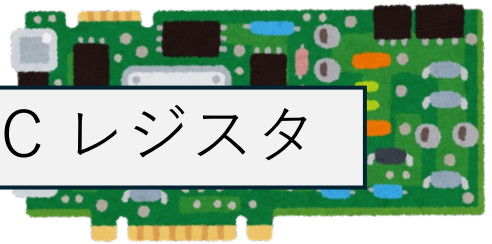


cr3:
仮想アドレス 0x1000 へアクセス

カーネルは
各プログラム（プロセス）ごとに
ページテーブルを用意する

DPDK を使って仮想スイッチを実装する場合の例

複数の要素が同一のリソースへアクセスしないようにする



NIC レジスタ

プロセス 1

仮想 NIC

エミュレータ実装

仮想スイッチ

仮想 NIC

レータ実装

エミュレータ実装だけでは
外部と疎通できないので
ソフトウェアで実装される
仮想スイッチと併用します

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

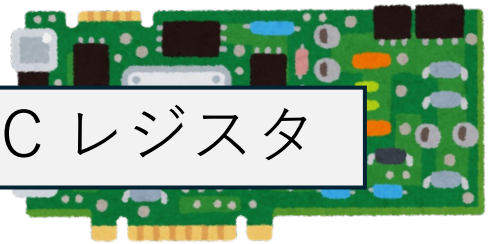
いなかったら？

で実装する

DPDK を使って仮想スイッチを実装する場合の例

複数の要素が同一のリソースへアクセスしないようにする

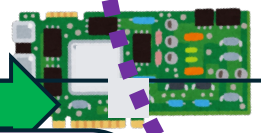
ポイント
物理 NIC へアクセスできるのは
仮想スイッチを実行するプロセスのみ



NIC レジスタ

仮想スイッチ

仮想 NIC



エミュレータ実装

エミュレータ実装だけでは
外部と疎通できないので
ソフトウェアで実装される
仮想スイッチと併用します

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

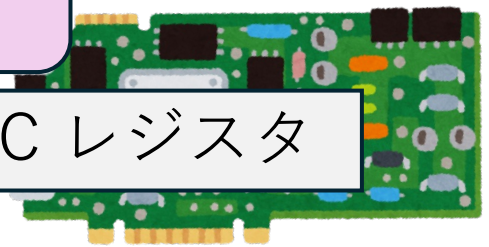
いなかったら？

で実装する

一つの NIC を共有する方法

このような構成になっていれば仮想スイッチはカーネルに実装されていても大丈夫です（例： netmap / VALE ）

複数の要素が同時にアクセスしないようにする

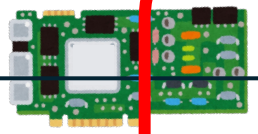


カーネル

プロセス 1

プロセス 2

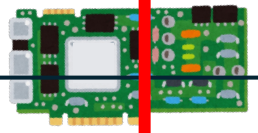
仮想 NIC



エミュレータ実装

仮想スイッチ

仮想 NIC



エミュレータ実装

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

Q. NIC が SR-IOV に対応していなかったら？

A. 仮想 NIC をソフトウェアで実装する

DPDK のインストール方法

- コマンド：
  https://github.com/yasukata/jumpstart-on-docker/tree/master#dpdk-installation

- おすすめしたいポイント
  - 実行するマシンごとにソースコードからコンパイルすること
    - 何故？：コンパイル時に、利用可能な CPU 命令などに合わせた最適化が適用されるため
      - 別のマシンでコンパイルしたライブラリファイルを利用する場合に対応していない CPU 命令が含まれているとプログラムが停止したりします
  - インストール先のディレクトリを指定してコンパイルすること
    - 何故？：複数の異なる DPDK ライブラリを保持できるようにするため
      - インストール先が一般ユーザーのディレクトリであればインストール自体に root 権限が不要なのも良いです

DPDK の使い方の基本ポイント

- ポイント１：hugepages を設定する
  - DPDK 付属のツールで設定できます
    - 例：./dpdk-version/usertools/dpdk-hugepages.py -p 2M -r 2G
      - -p: huge page size
      - -r: huge pages として扱う合計のメモリサイズ
- ポイント２：利用するデバイスは PCI バス・デバイス・ファンクション番号で指定する
  - lspci コマンドで確認できます
  - 基本的にはカーネルのドライバと紐付けられているので、DPDK 付属のツールで DPDK から扱えるようにします
    - 例：./dpdk-version/usertools/dpdk-devbind.py -b vfio-pci XXXX:XX:XX.X
      - -b: 紐付けるデバイスドライバです
        - vfio-pci は DPDK が扱えるようにするもので、Linux のカーネルに返す場合は、対応する NIC のドライバの名前を指定します
        - 設定可能なドライバの名前は -s オプションで確認できます

DPDK を使ってアプリを作る場合のおすすめの想定

- DPDK のライブラリは実行環境を操作する権限のある人からの提供を想定すべき
  - DPDK を利用するアプリケーションを配布する人が DPDK も一緒に配布しないようにした方が良い

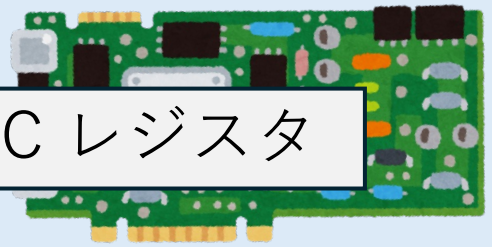
- 異なるバージョンや利用者固有の改変が含まれた DPDK ライブラリと併用されることを想定しておく方が良い

DPDK のインストール方法

- コマンド：
  https://github.com/yasukata/jumpstart-on-docker/tree/master#dpdk-installation
- おすすめしたいポイント
  - 実行するマシンごとにソースコードからコンパイルすること
    - 何故？：コンパイル時に、利用可能な CPU 命令などに合わせた最適化が適用されるため
      - 別のマシンでコンパイルしたライブラリファイルを利用する場合に対応していない CPU 命令が含まれているとプログラムが停止したりします
  - インストール先のディレクトリを指定してコンパイルすること
    - 何故？：複数の異なる DPDK ライブラリを保持できるようにするため
      - インストール先が一般ユーザーのディレクトリであればインストール自体に root 権限が不要なのも良いです

DPDK を使って仮想スイッチを実装する場合の例

複数の要素が同一のリソースへアクセスしないようにする



NIC レジスタ

プロセス 1

仮想 NIC

エミュレータ実装

仮想スイッチ

仮想 NIC

レータ実装

- リングバッファ用配列
- NIC の送受信データ保持領域
- 仮想 NIC レジスタ
- リングバッファ用配列
- NIC の送受信データ保持領域
- 仮想 NIC レジスタ
- リングバッファ用配列
- NIC の送受信データ保持領域

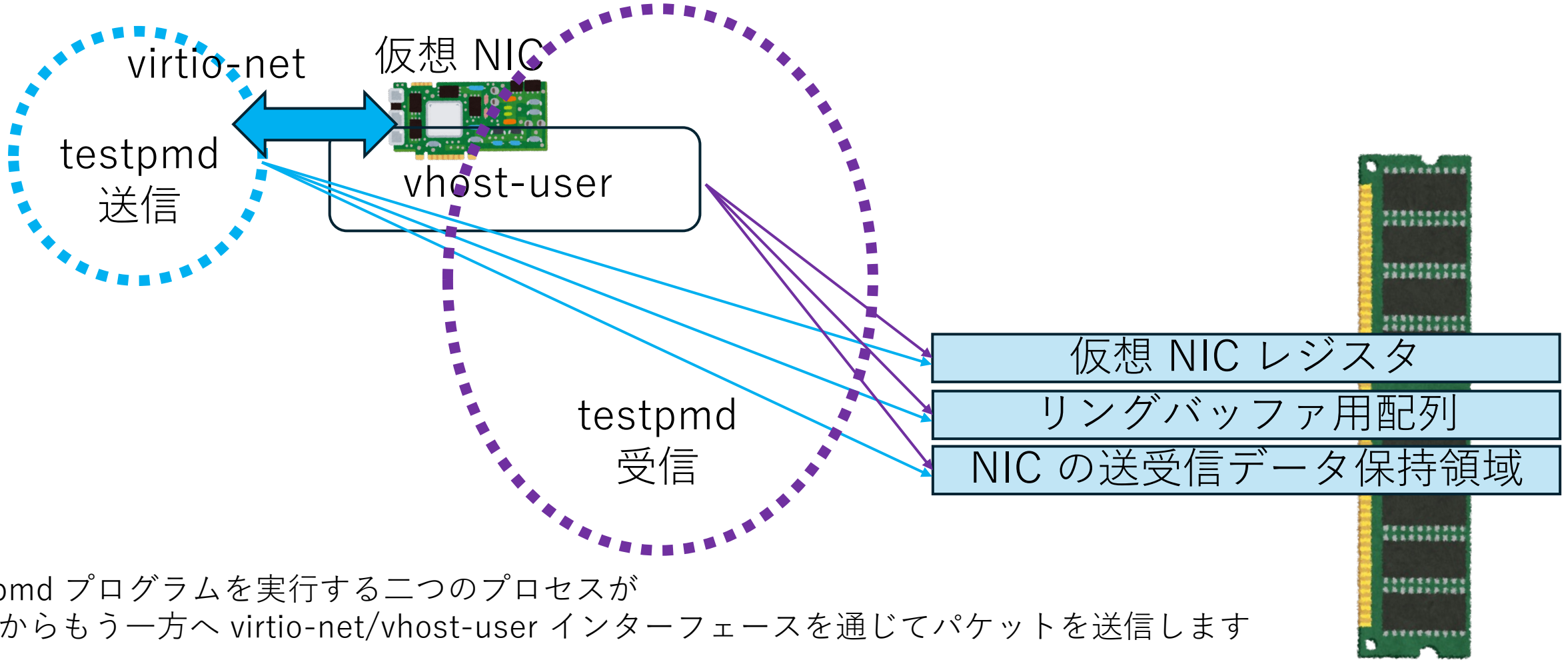
エミュレータ実装だけでは
外部と疎通できないので
ソフトウェアで実装される
仮想スイッチと併用します

いなかったら？

で実装する

DPDK のインストール方法

https://github.com/yasukata/jumpstart-on-docker/tree/master#dpdk-installation



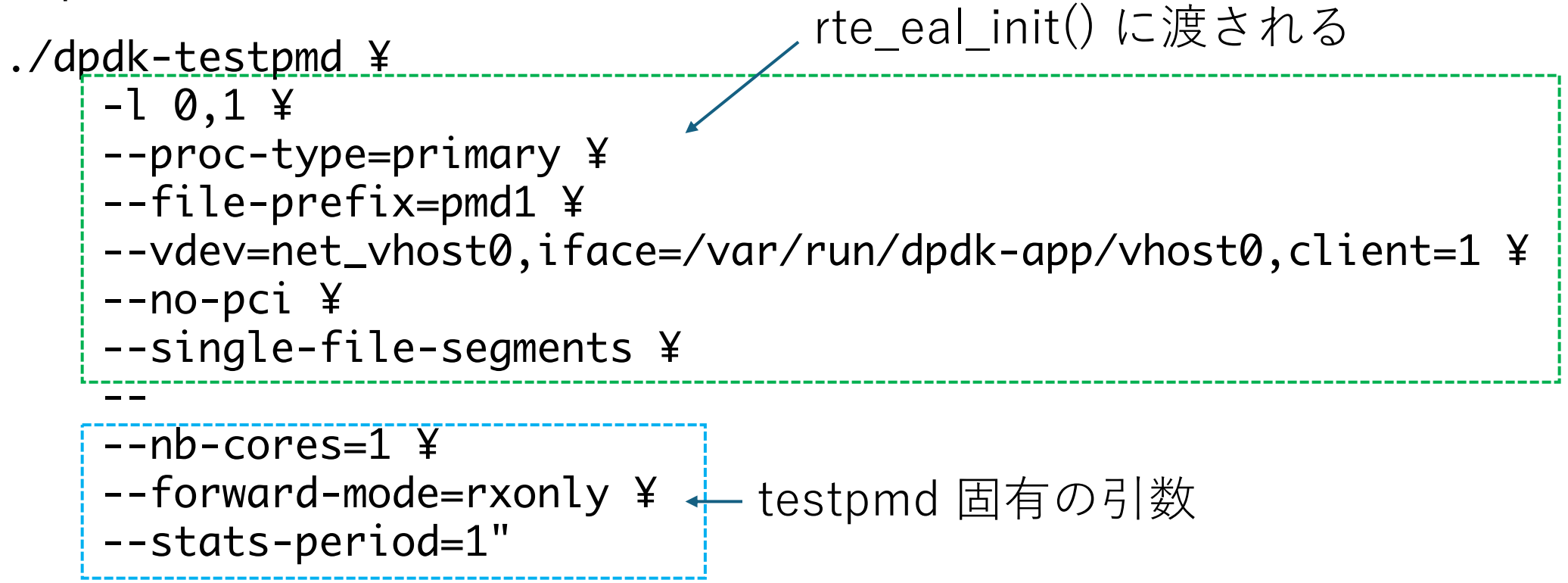
testpmd プログラムを実行する二つのプロセスが
片方からもう一方へ virtio-net/vhost-user インターフェースを通じてパケットを送信します

DPDK の使い方のポイント

- DPDK の API である rte_eal_init() の引数の渡し方だけ覚える
- よく使う引数
  - -l：実行に利用する CPU コアのリスト
  - --proc-type：だいたい primary を指定しておけば大丈夫
  - --file-prefix：起動するプロセスごとに変えることで、複数の DPDK を扱うアプリが実行できる
  - PCI デバイス関連
    - --allow：lspci コマンドで確認できる PCI アドレスを指定（この指定がないと全てのアクセス可能な PCI デバイスを探索する）
    - --no-pci：PCI デバイスを探索しない
  - 仮想デバイス設定
    - --vdev=net_デバイスの種類,デバイスドライバ依存情報,…
      - tap デバイスの例：--vdev=net_tap,iface=tap001

DPDK の使い方のポイント

- DPDK の API である rte_eal_init() の引数の渡し方だけ覚える
- testpmd の例

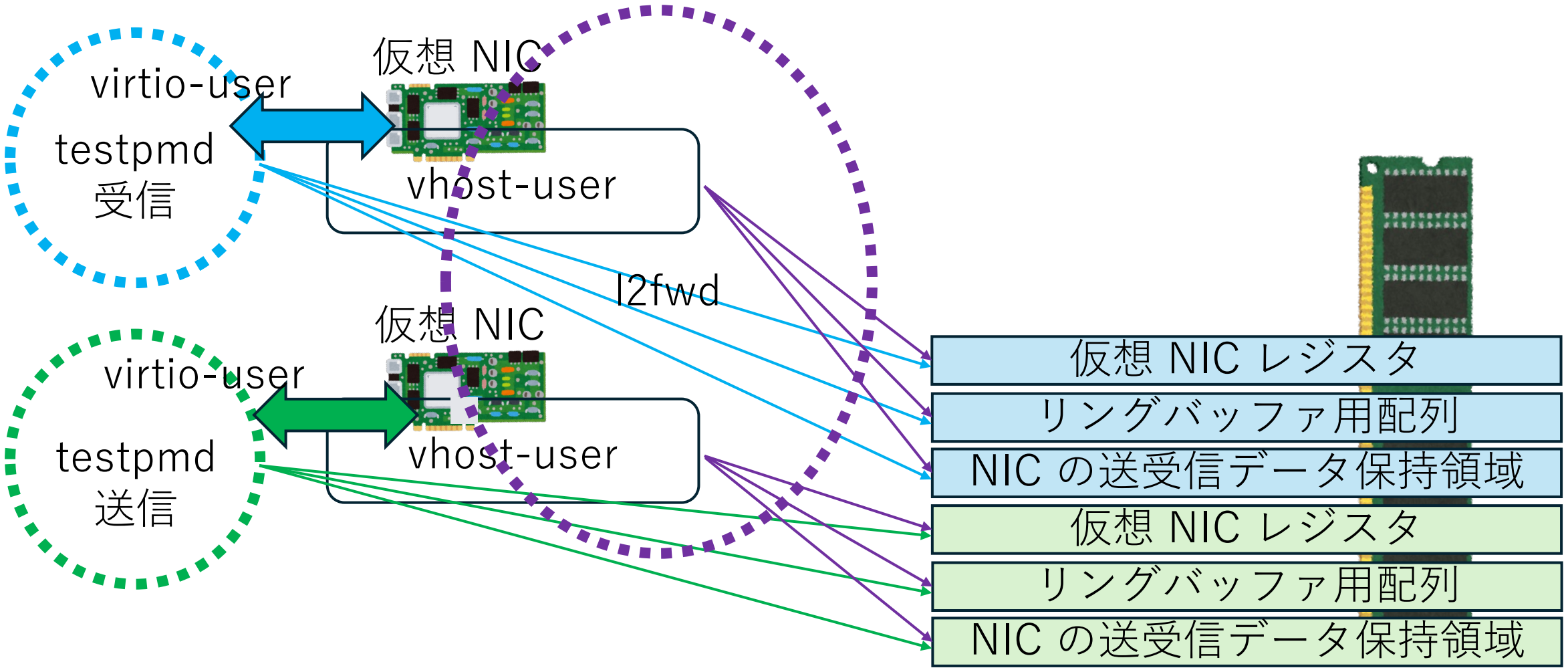


簡単な仮想スイッチ l2fwd を使う

- コマンド：
  https://github.com/yasukata/jumpstart-on-docker#dpdk-l2fwd-bridging-dpdk-testpmd-containers
- DPDK に付属している l2fwd という簡易的な仮想スイッチアプリで testpmd 間を接続することができます

簡単な仮想スイッチ l2fwd を使う

https://github.com/yasukata/jumpstart-on-docker#dpdk-l2fwd-bridging-dpdk-testpmd-containers

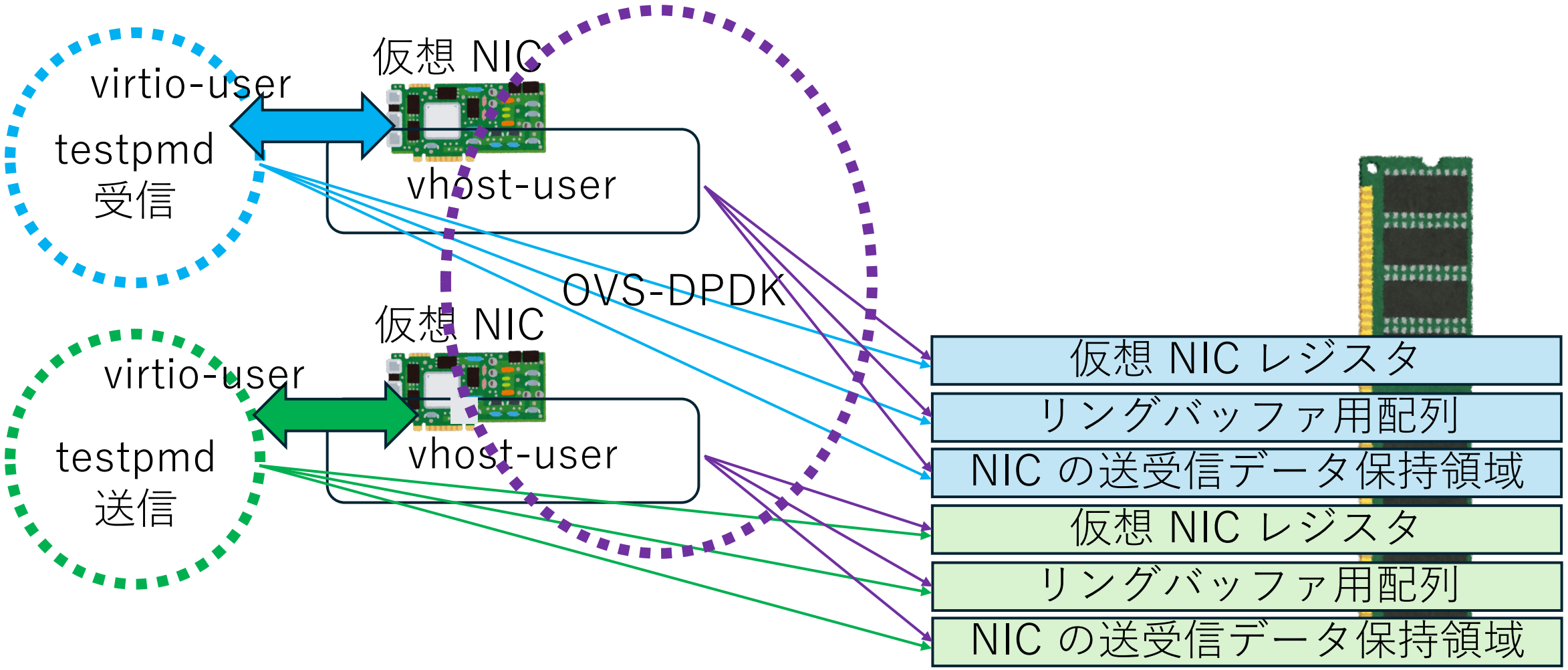


OVS-DPDK のインストール方法

- コマンド：
  https://github.com/yasukata/jumpstart-on-docker#ovs-dpdk-installation
- Open vSwitch で DPDK アプリ間を接続することができます

OVS-DPDK のインストール方法

https://github.com/yasukata/jumpstart-on-docker#ovs-dpdk-installation

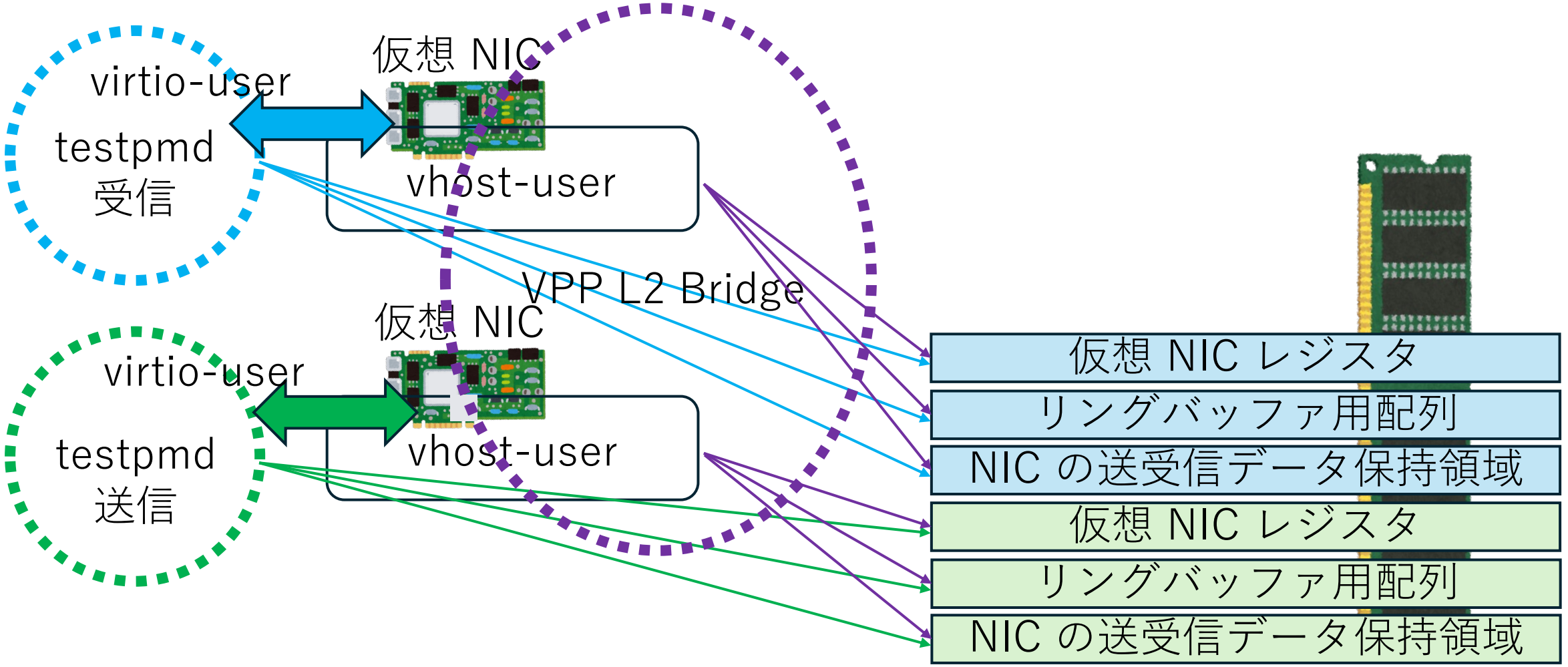


VPP のインストール方法

- コマンド：
  https://github.com/yasukata/jumpstart-on-docker#vpp-installation
- VPP の L2 Bridge 機能 で DPDK アプリ間を接続します

VPP のインストール方法

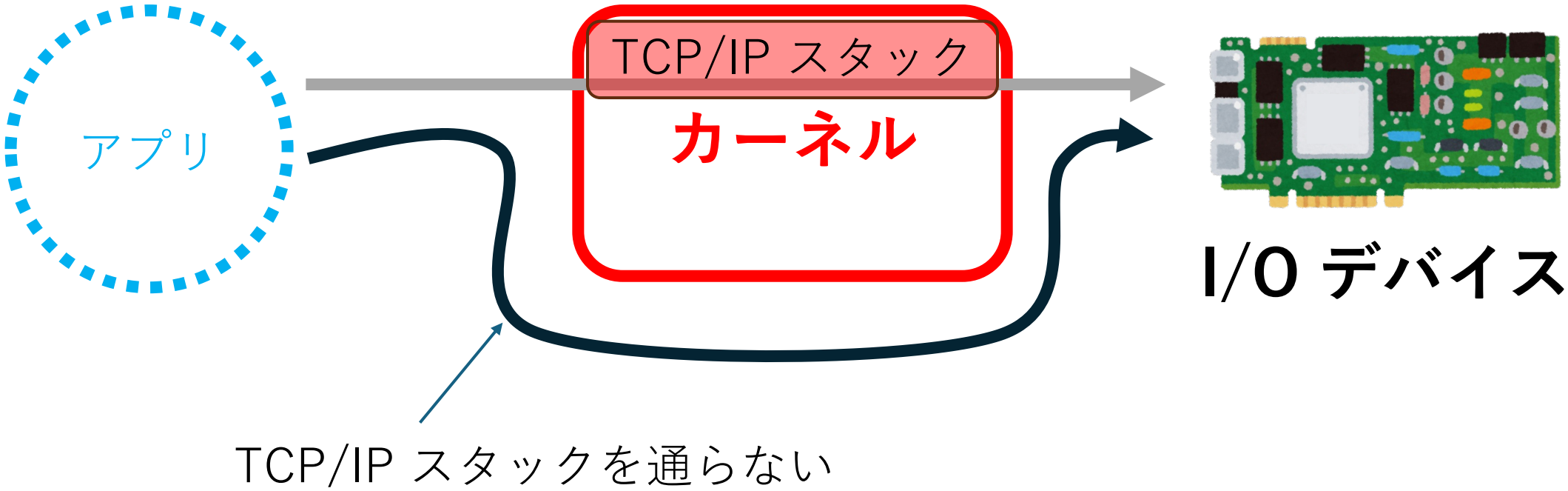
https://github.com/yasukata/jumpstart-on-docker#ovs-dpdk-installation



TCP/IP 通信を行う方法

TCP/IP 通信を行う方法

- 通常 TCP/IP スタックはカーネルに実装されているので、カーネルをバイパスすると TCP/IP 通信ができない



TCP/IP 通信を行う方法

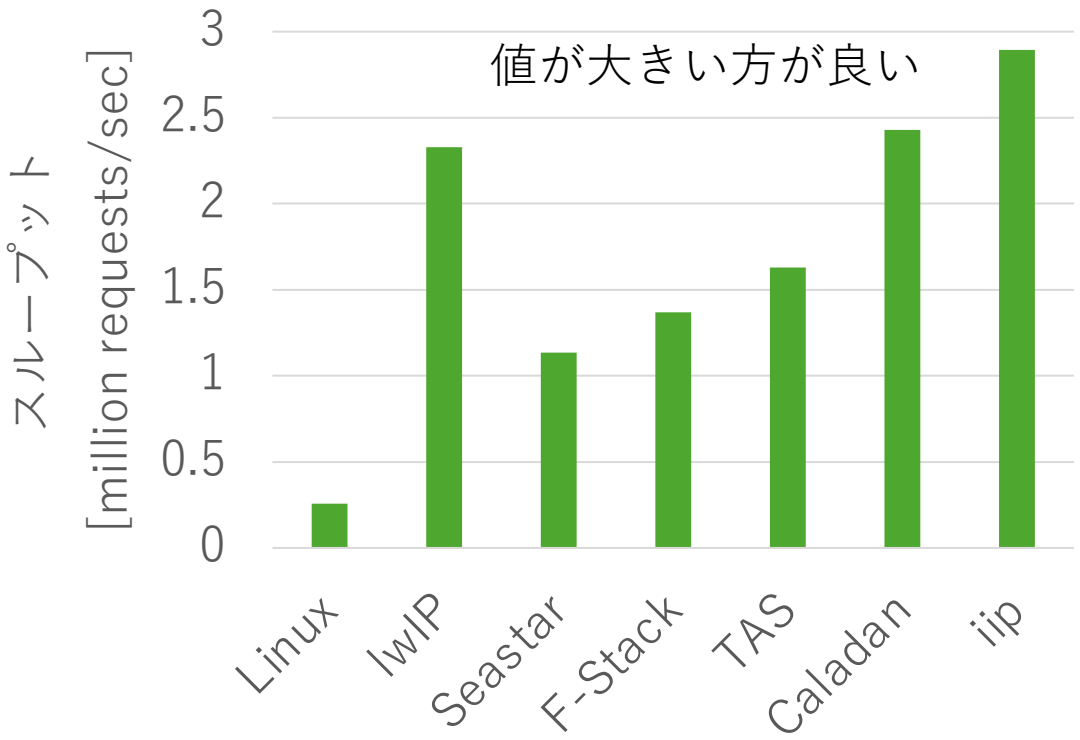
- 通常 TCP/IP スタックはカーネルに実装されているので、カーネルをバイパスすると TCP/IP 通信ができない

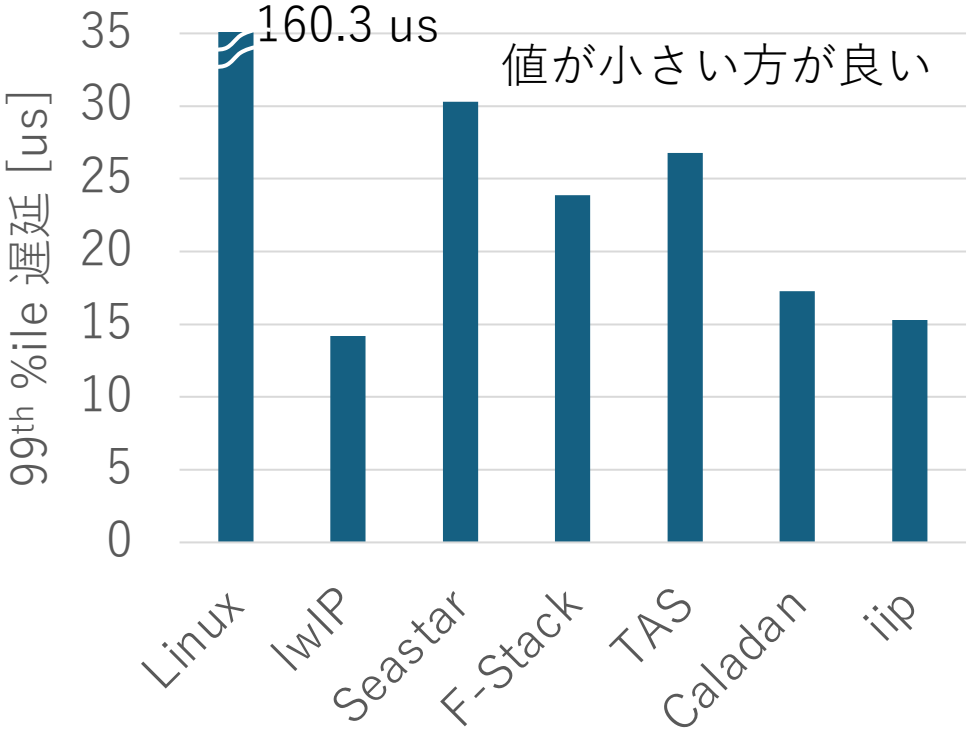
- アプリの一部として TCP/IP スタックを動かせばカーネルをバイパスしながら TCP/IP 通信ができます

TCP/IP スタック実装の選択肢

アプリケーションが 1 CPU コアを利用して 32 並列接続を通して 1 バイトのメッセージを往復させる場合の性能

注意：それぞれの実装の機能が違うため公平な比較ではありません。目安としてご覧ください



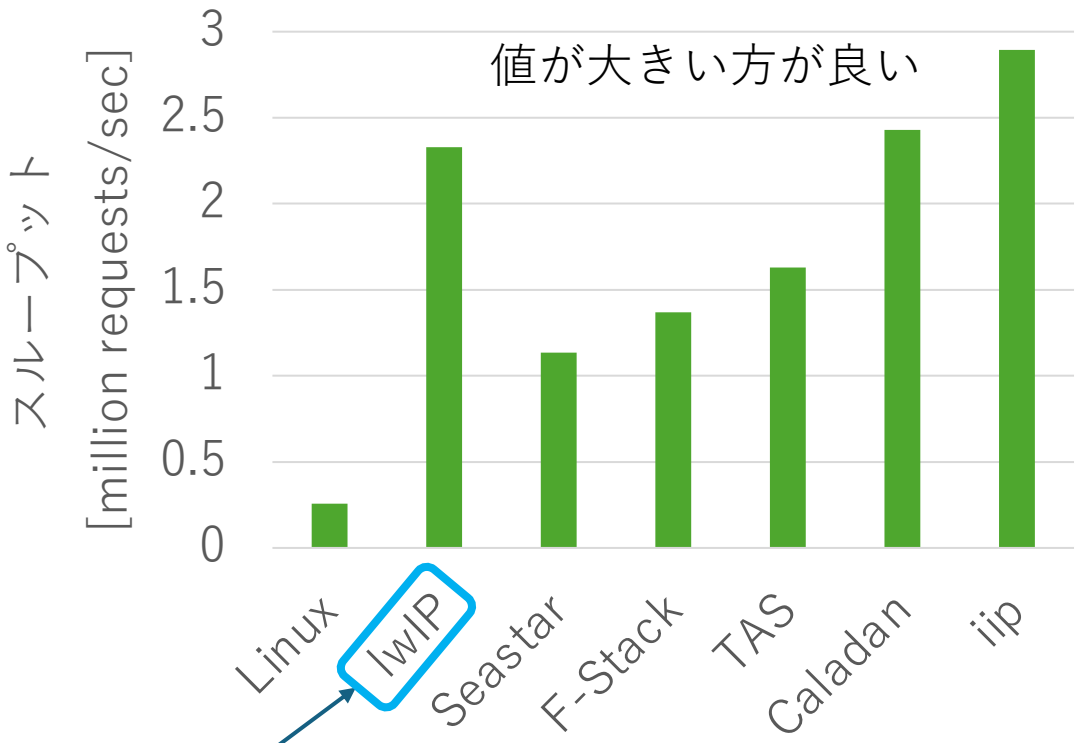


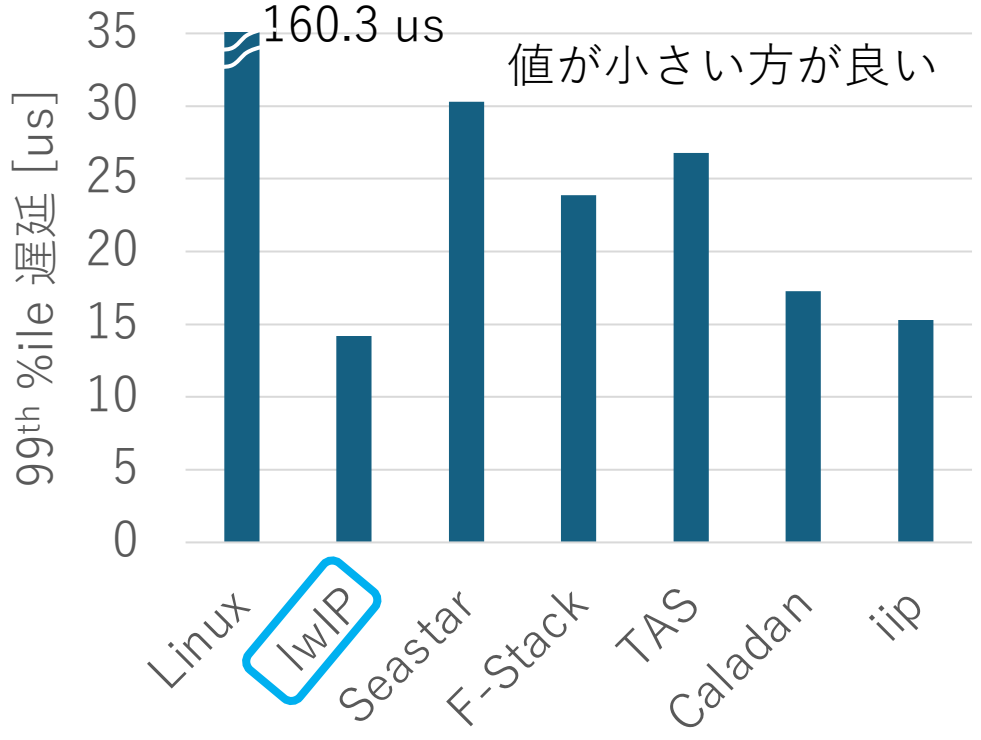
実験の設定とコマンド： https://github.com/yasukata/bench-iip#performance-numbers-of-other-tcpip-stacks

TCP/IP スタック実装の選択肢

アプリケーションが 1 CPU コアを利用して 32 並列接続を通して 1 バイトのメッセージを往復させる場合の性能

注意：それぞれの実装の機能が違うため公平な比較ではありません。目安としてご覧ください





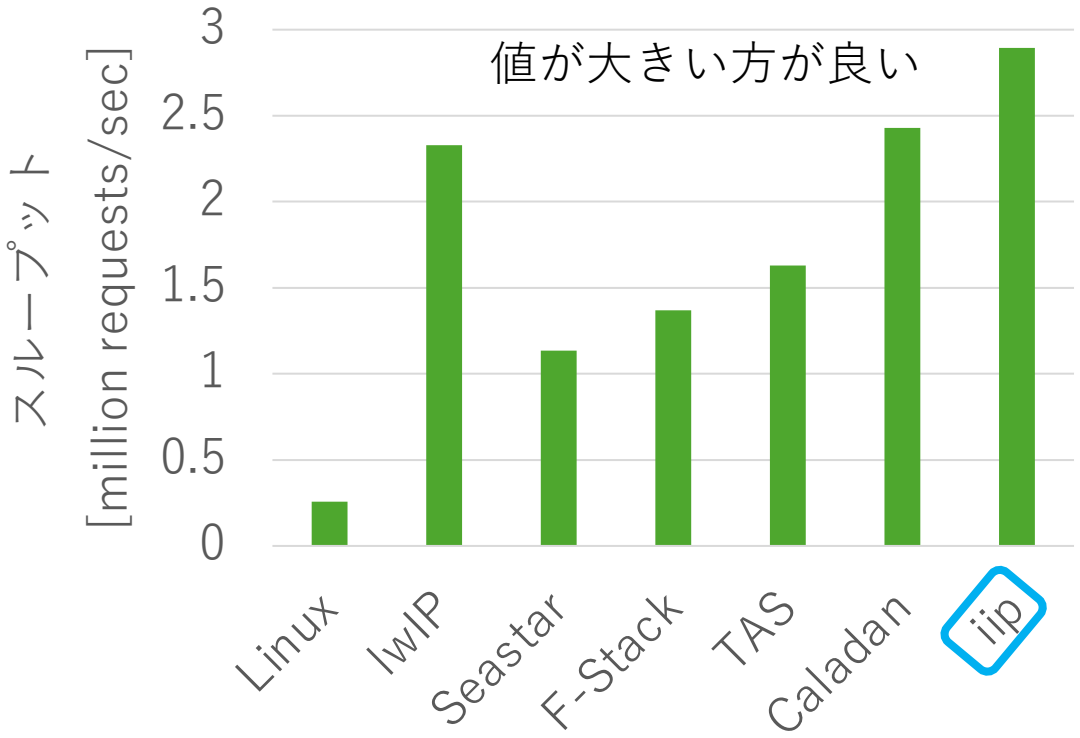
個人的なおすすめ：lwIP は組み込み用途で長らく広く利用され、最高速を目指す研究でも採用されています
（ですが、lwIP は 1 CPU コアでしか利用できず NIC のオフロード機能にも対応していないという制約があります）

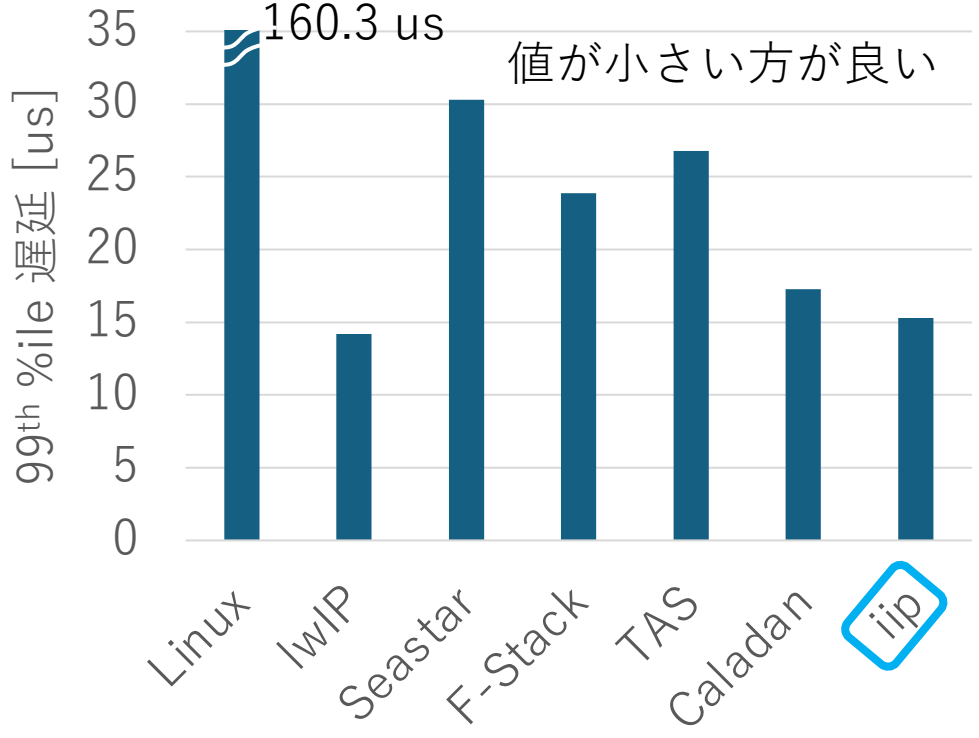
実験の設定とコマンド： https://github.com/yasukata/bench-iip#performance-numbers-of-other-tcpip-stacks

TCP/IP スタック実装の選択肢

アプリケーションが 1 CPU コアを利用して 32 並列接続を通して 1 バイトのメッセージを往復させる場合の性能

注意：それぞれの実装の機能が違うため公平な比較ではありません。目安としてご覧ください



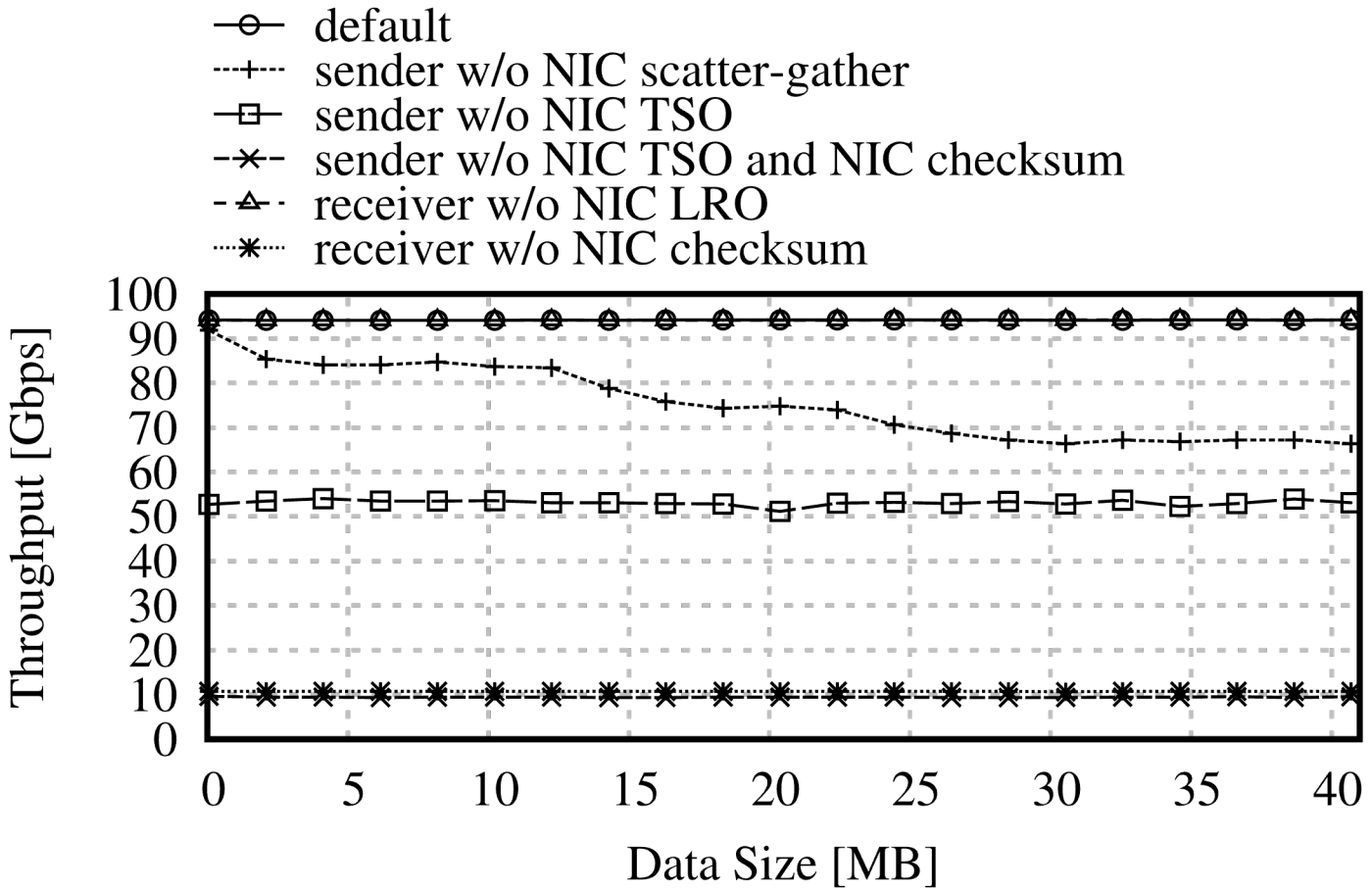


iip はこれら制約がないように作られているので、lwIP で物足りなくなったら試してみてください
（ですが、lwIP は 1 CPU コアでしか利用できず NIC のオフロード機能にも対応していないという制約があります）

実験の設定とコマンド： https://github.com/yasukata/bench-iip#performance-numbers-of-other-tcpip-stacks

NIC のオフロード機能の効果

- 100 Gbps NIC を通じて大きなデータを片方のマシンからもう一方のマシンへ送る場合



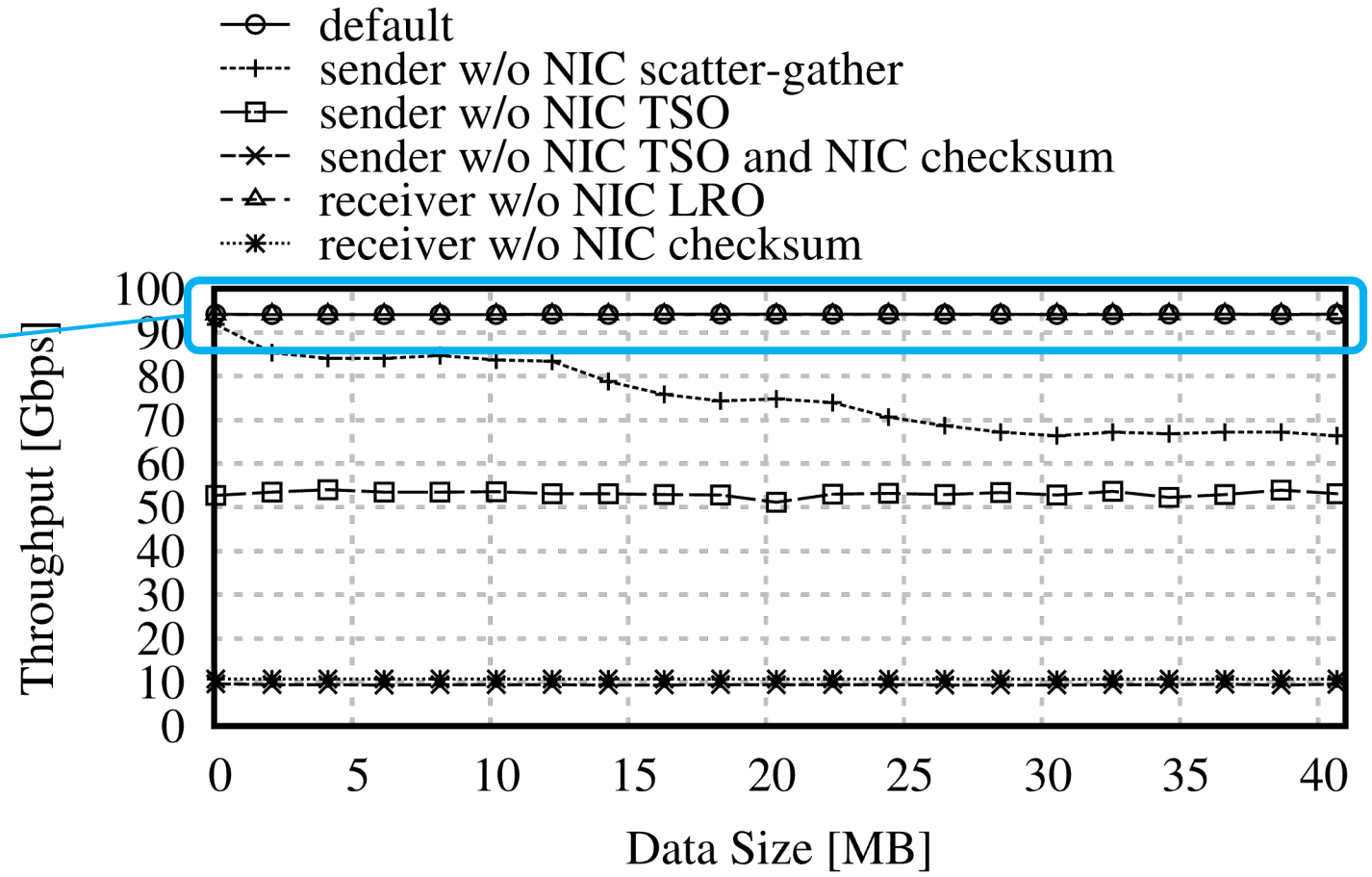
https://doi.org/10.1145/3687230.3687233

NIC のオフロード機能の効果

- 100 Gbps NIC を通じて大きなデータを片方のマシンからもう一方のマシンへ送る場合

全部のオフロードが
有効だと約 100 Gbps

NIC の速度が 10 Gbps くらいまでであれば
オフロード機能はなくても大丈夫かもしれません



https://doi.org/10.1145/3687230.3687233

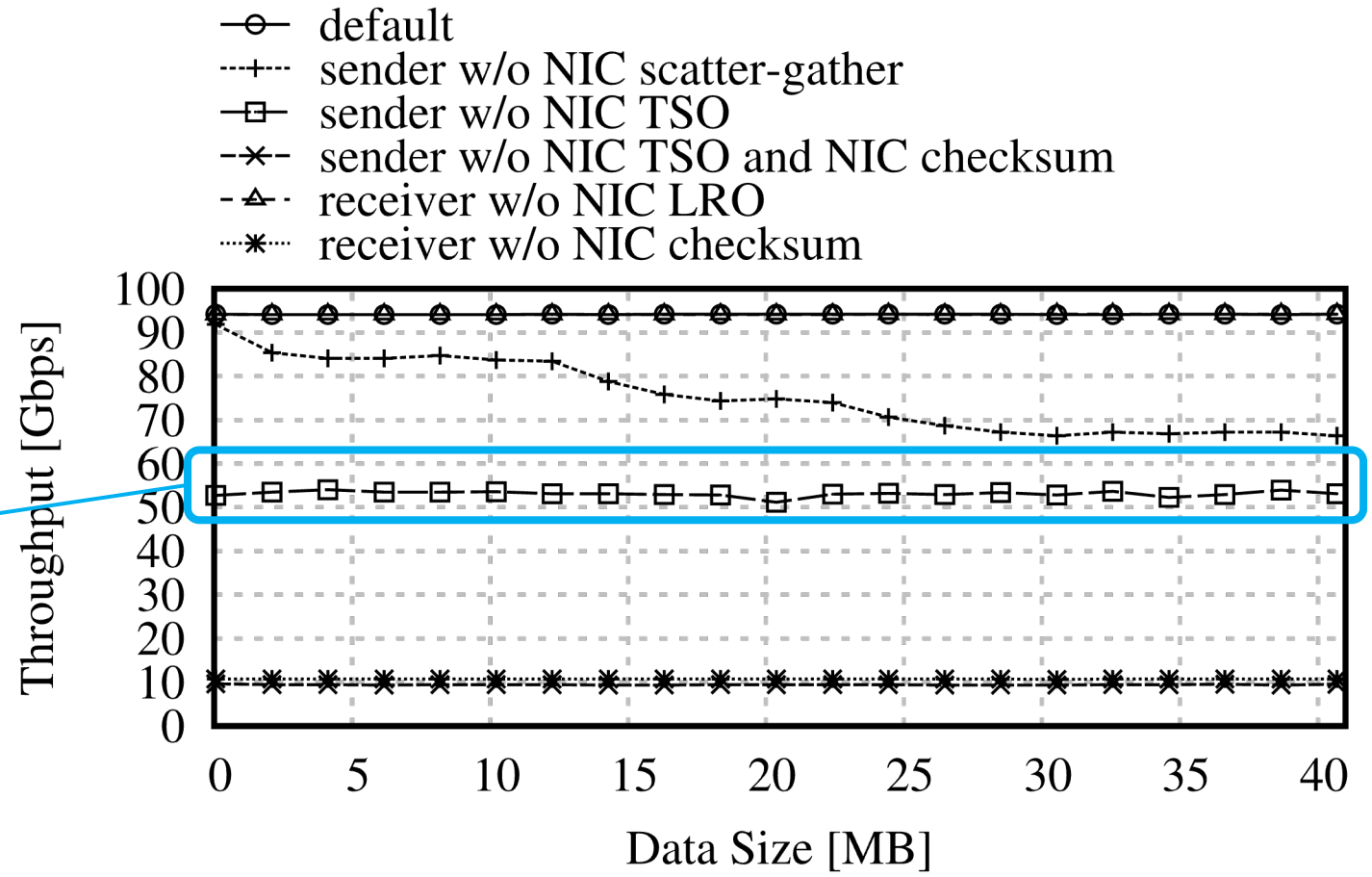
NIC のオフロード機能の効果

- 100 Gbps NIC を通じて大きなデータを片方のマシンからもう一方のマシンへ送る場合

全部のオフロードが
有効だと約 100 Gbps

TSO が無効だと約 50 Gbps

NIC の速度が 10 Gbps くらいまでであれば
オフロード機能はなくても大丈夫かもしれません



https://doi.org/10.1145/3687230.3687233

NIC のオフロード機能の効果

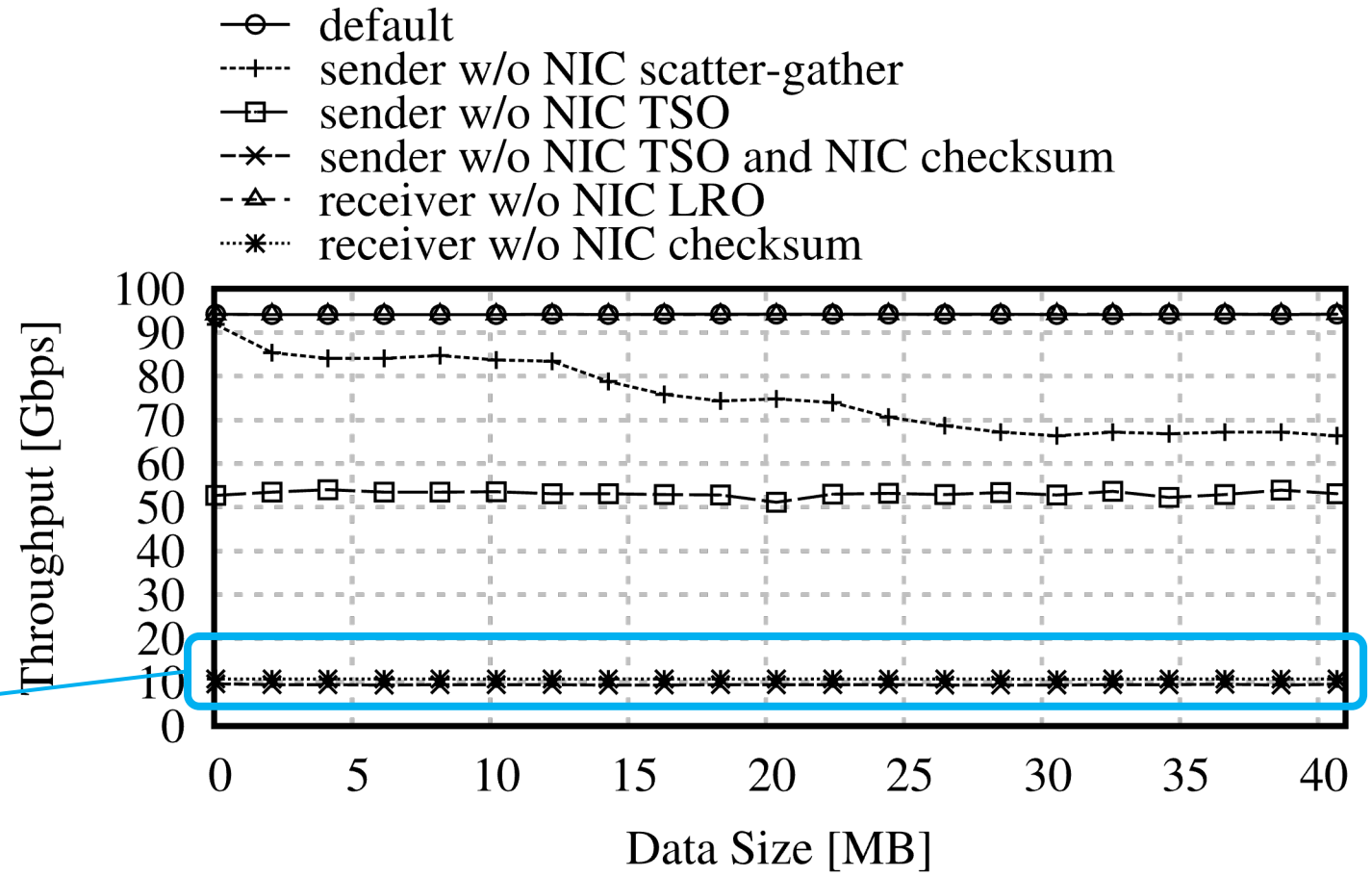
- 100 Gbps NIC を通じて大きなデータを片方のマシンからもう一方のマシンへ送る場合

全部のオフロードが有効だと約 100 Gbps

TSO が無効だと約 50 Gbps

チェックサムオフロードが無効だと約 10 Gpbs

NIC の速度が 10 Gbps くらいまでであれば
オフロード機能はなくても大丈夫かもしれません



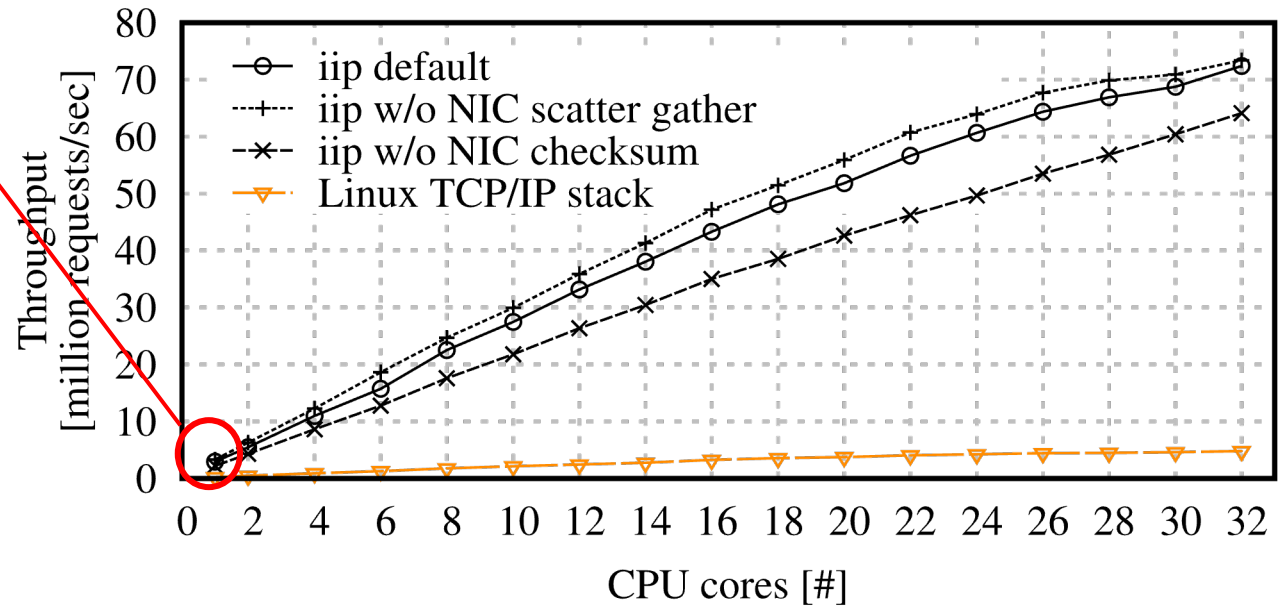
https://doi.org/10.1145/3687230.3687233

マルチコア環境での性能

- 100 Gbps NIC を経由して２つのマシン間で TCP 接続を通じて１バイトのメッセージを往復させる場合のスループット
  - １CPUコアが 32 並列接続 TCP を処理するように並列数を調整

1 CPU コアしか利用できない場合は
2~3 million requests / sec くらい

複数コアをうまく使えると
もっとスループットを伸ばせます



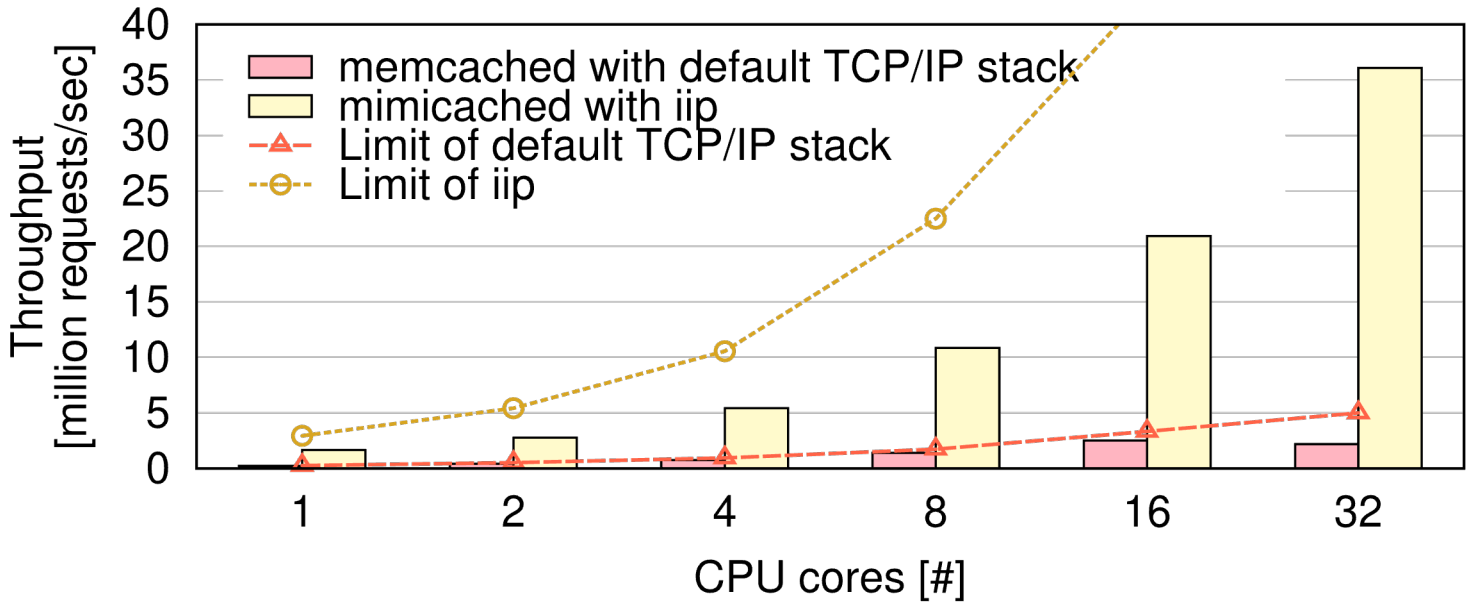
https://github.com/yasukata/bench-iip#multi-core-server-performance

マルチコア環境での性能

- DPDK と iip の上で memcached 互換サーバーを動かすと
  公式の memcached 実装よりも大幅に高い性能を発揮できます
  - https://github.com/yasukata/mimicached

赤い線が Linux の TCP/IP スタックの
最高性能を表しており
インメモリストレージ実装が
速くてもこれ以上高速化できない
要因になっています

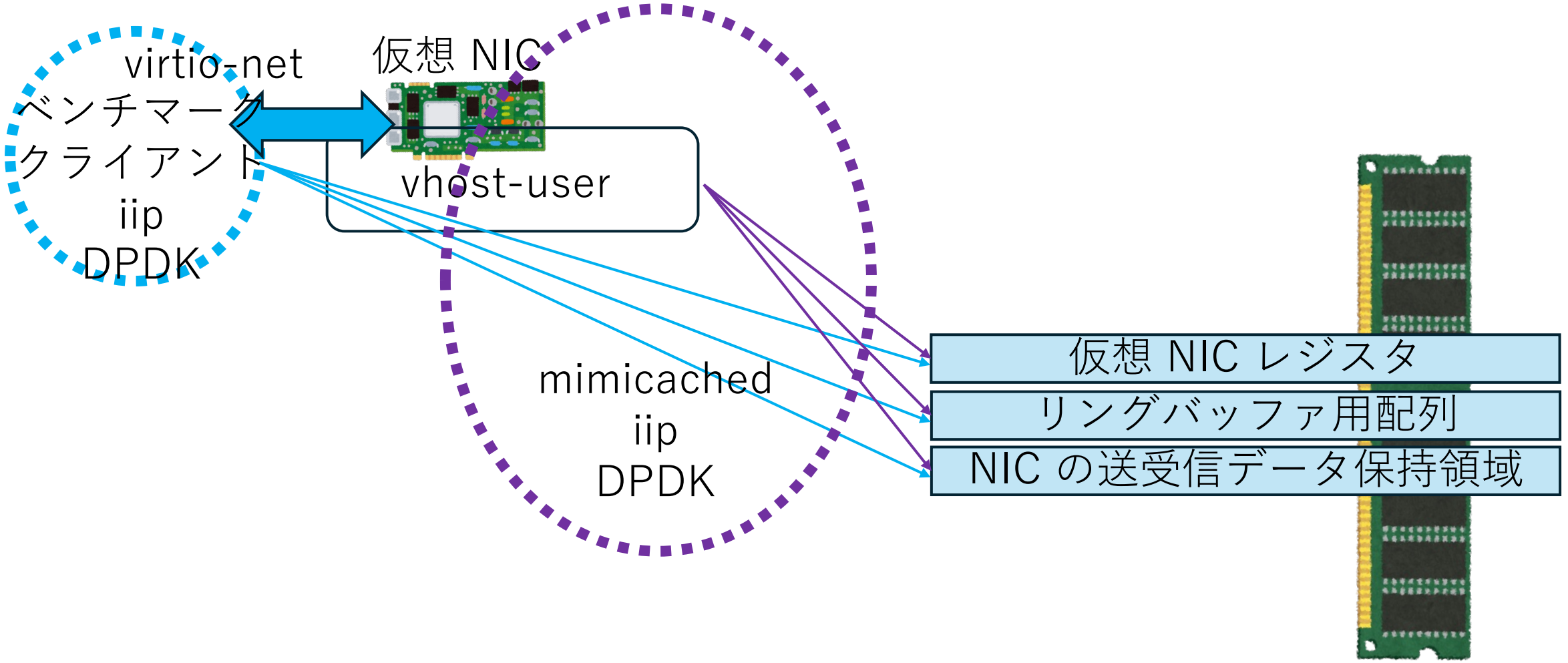
DPDK と iip を使うと
この要因を大幅に緩和できます



https://speakerdeck.com/yasukata/shi-jie-zui-su-ji-memcached-hu-huan-sabazuo-tuta

TCP/IP アプリの使い方

コマンド：https://github.com/yasukata/jumpstart-on-docker#mimicached-on-dpdk



AF_XDP のインストール方法

- コマンド：`apt install libxdp-dev`
  - AF_XDP のインストールというよりは、Linux に実装されている AF_XDP 機能を利用するためのライブラリのインストール

- AF_XDP の基本的な機能は Linux カーネルに含まれているものであるのでコンパイルが必要な DPDK よりも用意自体は手軽

AF_XDP 初期化時の設定ポイント

- 送受信リングサイズ：十分に大きい値が設定されている方が良いかも？
- XDP_FLAGS_DRV_MODE：できたら有効の方が速い
- XDP_USE_NEED_WAKEUP
  - 有効にしないとカーネルスレッドが busy loop して 100% CPU を使い続けたりするので基本は有効で良さそう、だが busy loop のおかげで遅延が減る場合もある（次ページ rx-usecs との兼ね合いもありそう）
- XDP_ZEROCOPY：できたら有効の方が速い
- SO_PREFER_BUSY_POLL

NIC のドライバのサポートの不足により設定に失敗してエラーが返ってくる場合は
オプションを無効にして再度設定を試みるように実装するのが良いようです

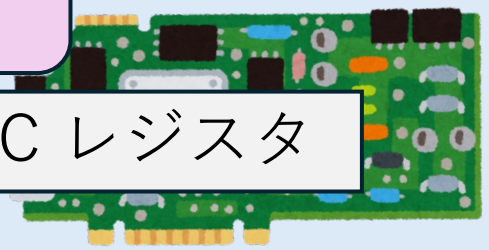
AF_XDP の設定のポイント

- 物理 NIC 利用時に ethtool で事前に設定すると良さそうな項目
  - NIC のキューの数をプログラムが使う数と揃える (-L)
    - 複数のキューが有効で NIC が受信したパケットを分散して振り分けている場合には、プログラムは全てのキューを確認しないと全てのパケットを受け取れない
  - 割り込み頻度 (-C)
    - rx-usecs 0 にしないと遅い場合がある？
    - カーネルの更新で改善される可能性もあると思います
  - 送受信リングサイズ (-G)
    - 十分に大きい値が設定されている方が良いかも？
- 仮想 NIC を利用したい場合はサポートの充実度の観点から veth を利用するのが良さそうです

一つの NIC を共有する方法

このような構成になっていれば仮想スイッチはカーネルに
実装されていても大丈夫です（例： netmap / VALE ）

複数の要素が同一のリソースを共有する



プロセス 1

プロセス 2

仮想 NIC

エミュレータ実装

仮想 NIC

エミュレータ実装

カーネル

仮想スイッチ

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

NIC の送受信データ保持領域

仮想 NIC レジスタ

リングバッファ用配列

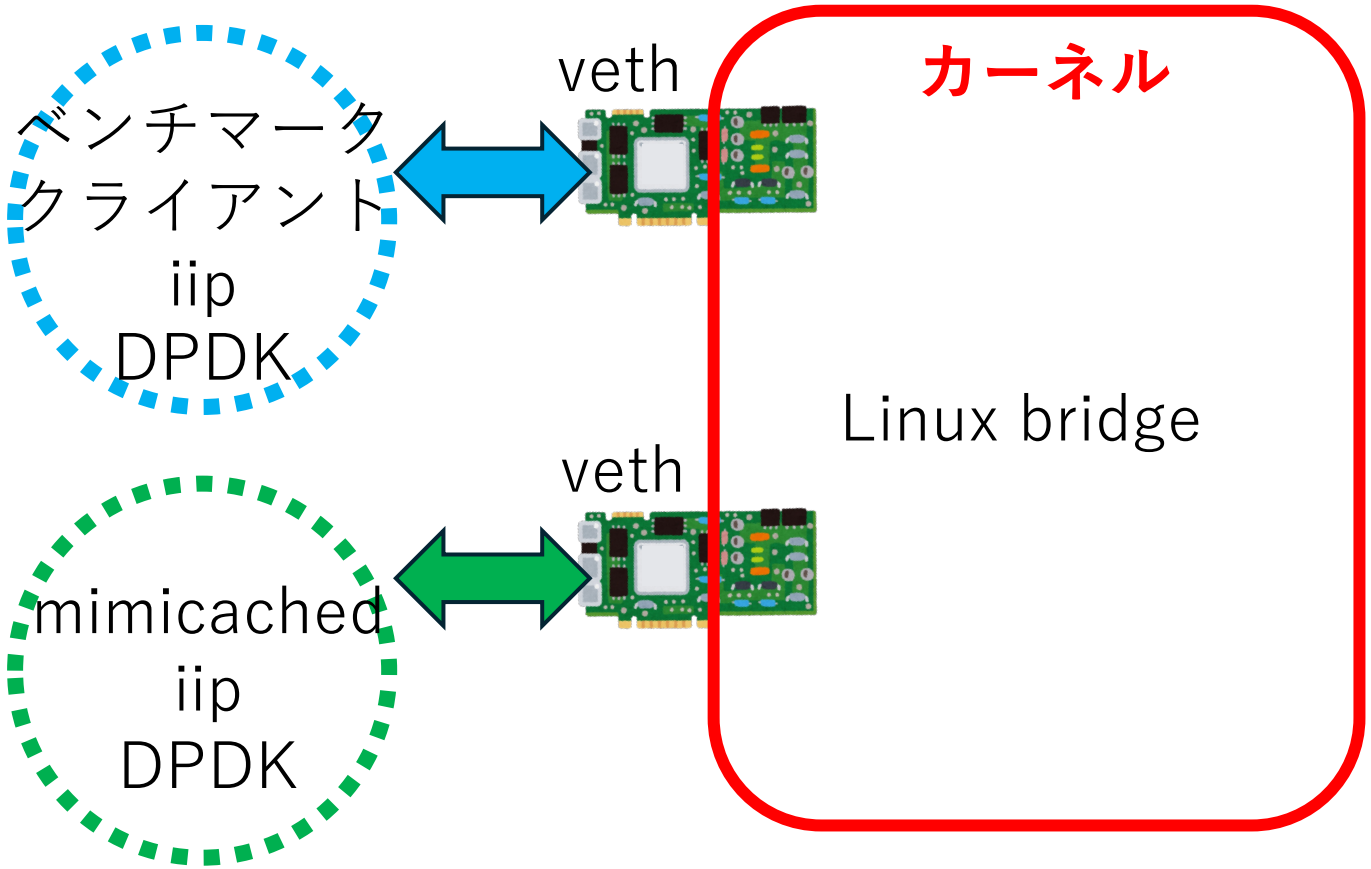
NIC の送受信データ保持領域

Q. NIC が SR-IOV に対応していなかったら？

A. 仮想 NIC をソフトウェアで実装する

TCP/IP アプリの使い方

コマンド：https://github.com/yasukata/jumpstart-on-docker?tab=readme-ov-file#mimicached-on-af_xdp



その他

- 高い性能を発揮するためには、頻繁に実行される箇所が適切に実装されている必要があります
- 特にデバイスドライバは重要で性能に大きく影響するとともに NIC ごとに DPDK 付属の実装と AF_XDP が扱う実装で精度にばらつきがあります
  - 必ずしも全てのデバイスドライバが最高の性能を発揮できるように実装されていなさそうです
- なので、単一のパケット I/O 機構に依存せず、複数のパケット I/O 機構の利用をサポートしておいて、環境に合わせて高い性能が発揮できる設定を選べるようにすることがおすすめです

まとめ

- カーネルバイパス構成の理解に重要と思われるポイント
  1. プログラム（CPU 命令）によるメモリアクセス方法
  2. ページテーブルを通したメモリアクセスの制限方法
  3. NIC のリングバッファの構成

  上記３点を基本とするとイメージがしやすくなると考えます
- セキュリティのポイント
  - 複数の要素でリソースを共有しないこと
  - 信頼しない実装が NIC のレジスタへアクセスにする場合には IOMMU などを利用して NIC がアクセス可能な DRAM 領域を制限すること
- 一つの NIC を共有する場合のポイント
  - 仮想 NIC を使う（SR-IOV かソフトウェアでのエミュレーション）
  - 仮想スイッチで仮想 NIC に外部との疎通性を提供する