❖ アドレス空間の大きさに制限されないスレッド移動を実現するPGAS処理系 ❖

東京大学　原健太朗，中島潤，田浦健次朗

2010.8.5

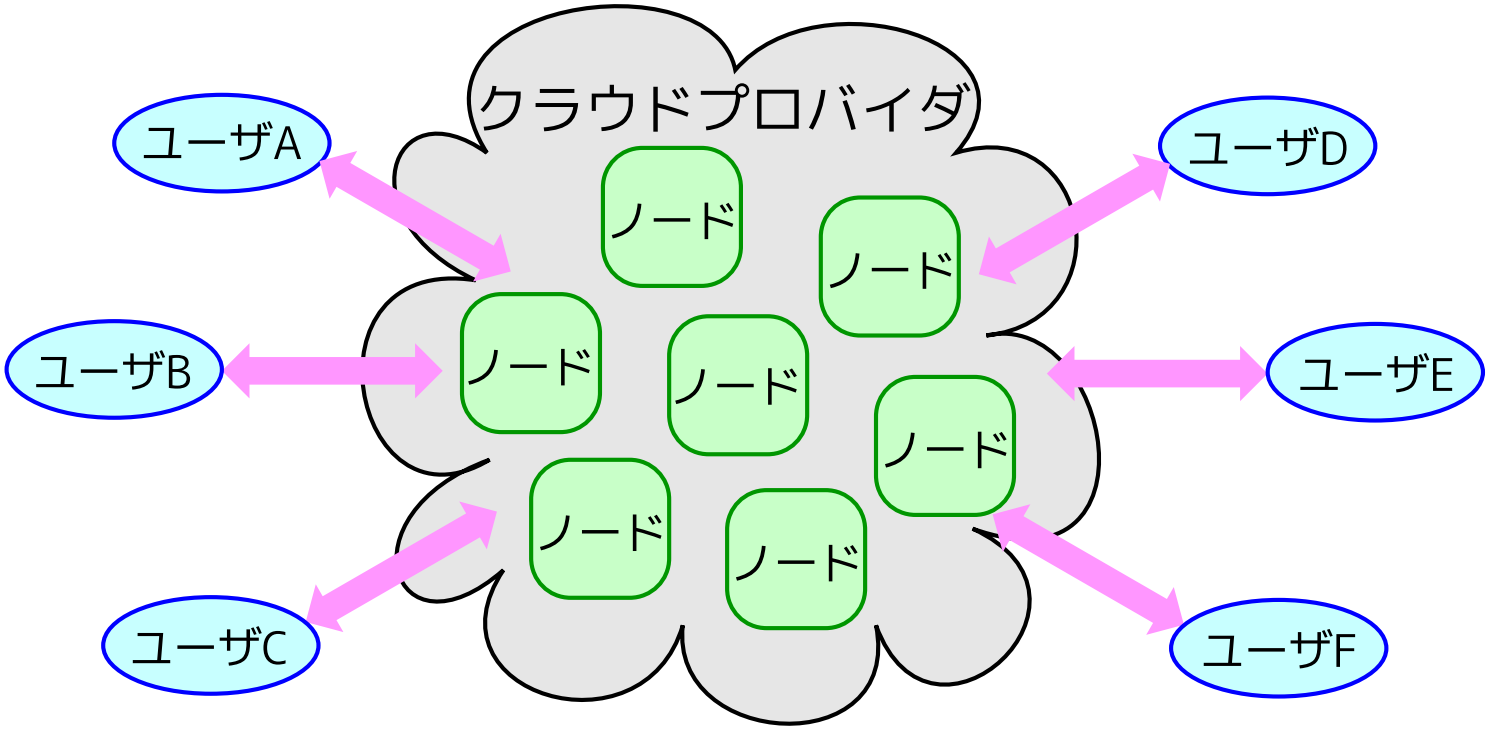
❖ 序論

背景と目的

- 背景：大規模な並列科学技術計算
  - → 応力解析
  - → 流体解析
  - → 地震シミュレーション
  - → …
- 目的：大規模な並列科学技術計算をクラウド上で効率的に実行できる並列分散プログラミング処理系を作る

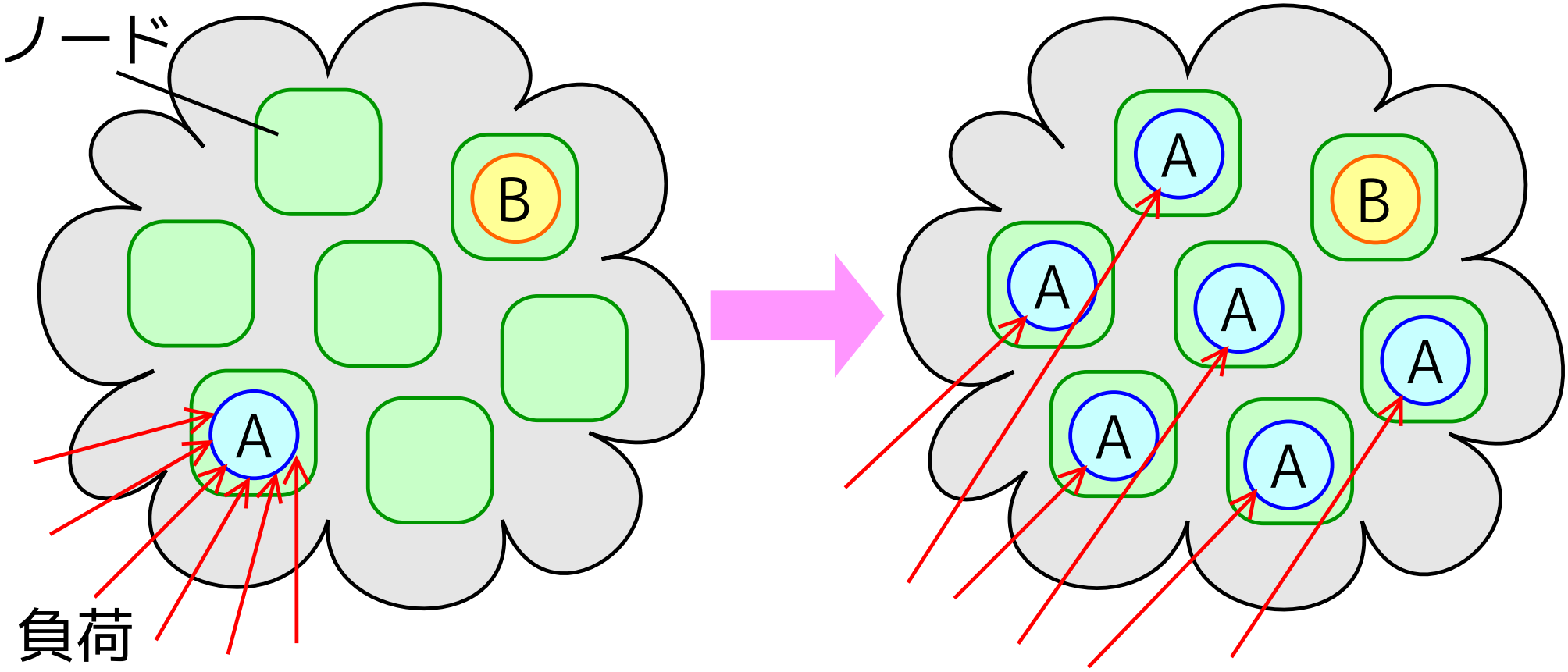
クラウドとは何か?

- クラウドのしくみ：
  - → プロバイダがデータセンタを構築・管理して，それらの計算資源をサービスとして提供
  - → ユーザは，それらの計算資源を従量性課金のもとで必要なときに必要なだけ利用可能
- モデル：**多数の計算資源を多数のユーザで利用**
  - → 計算資源はどうスケジューリングされるのか?



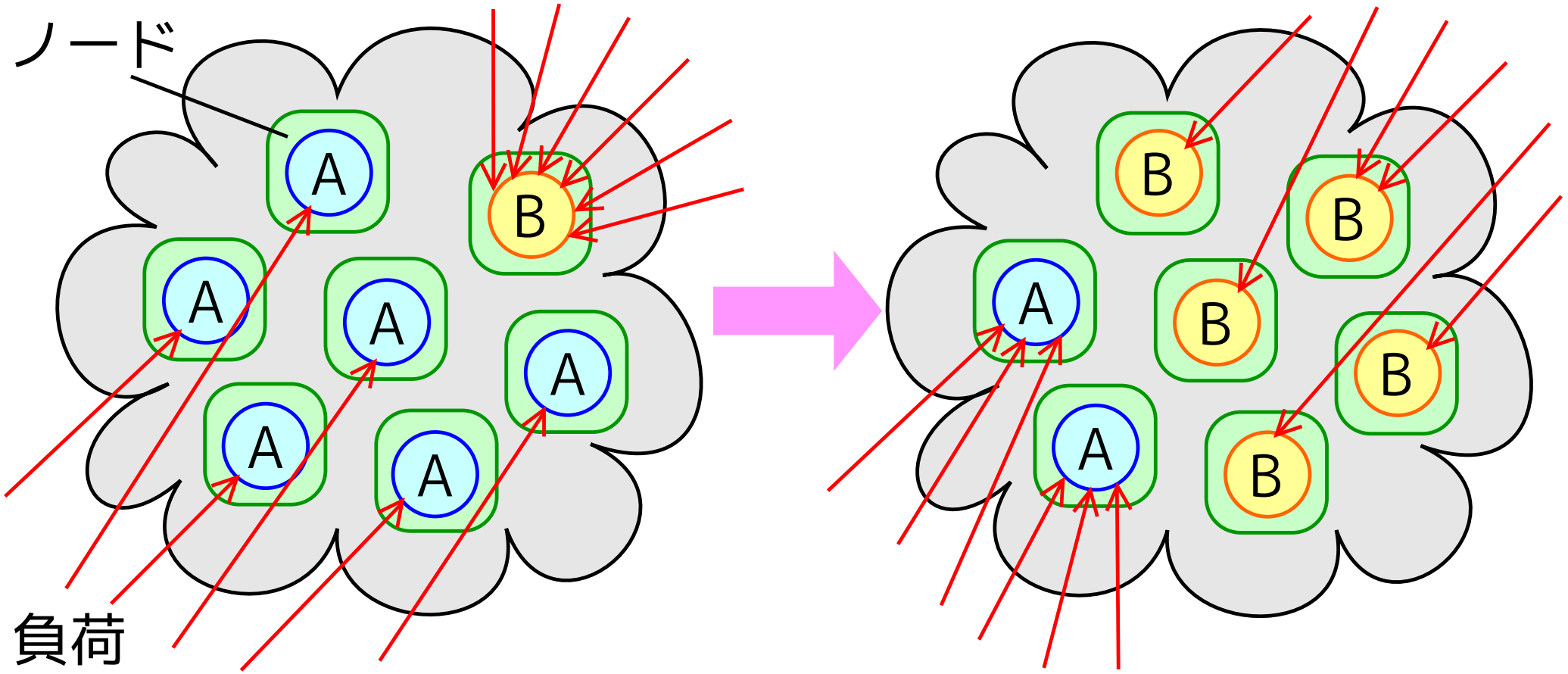
クラウドにおける計算資源のスケジューリング (1)

- ユーザ A の負荷が増大したら，ユーザ A の計算規模が拡張



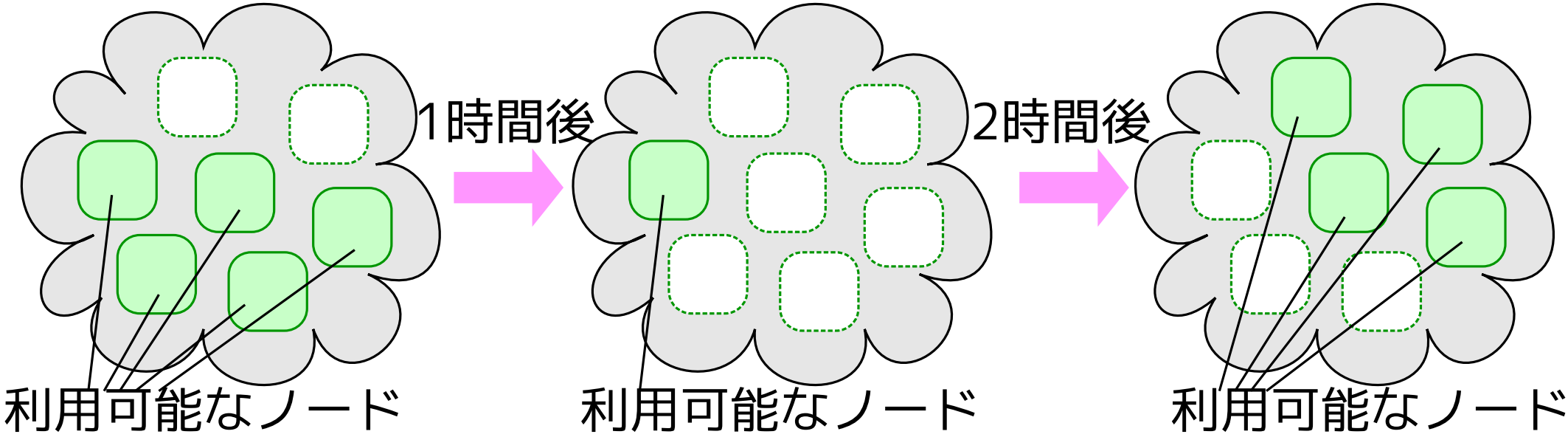
クラウドにおける計算資源のスケジューリング (2)

- やがてユーザ B の負荷が増大したら，(何らかのスケジューリングポリシーに基づいて) ユーザ A の計算規模を縮小する代わりにユーザ B の計算規模を拡張



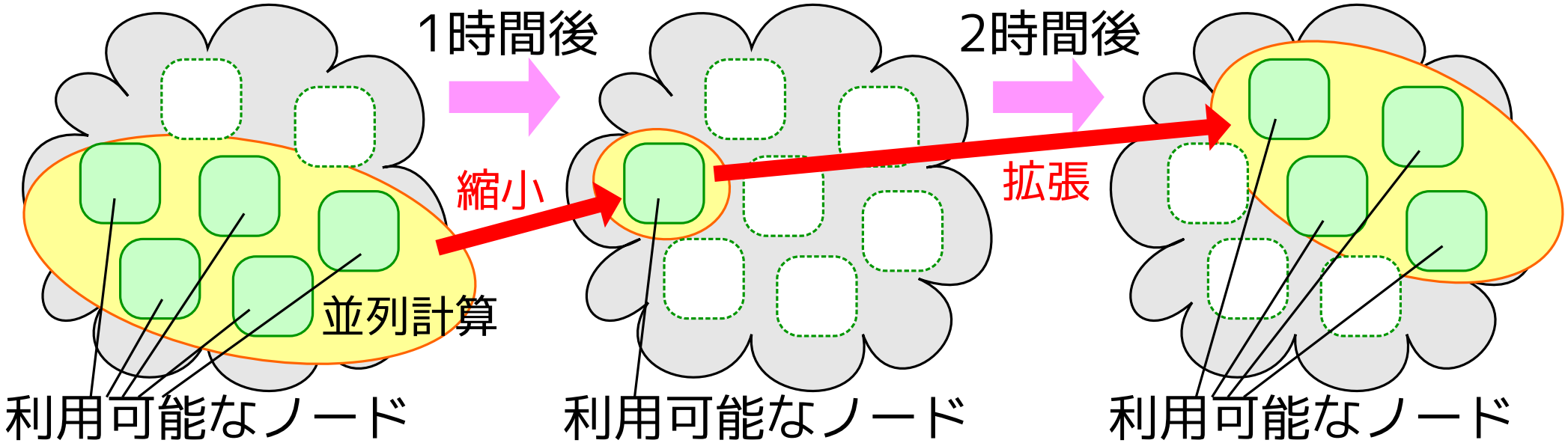
クラウドの本質

- モデル：多数の計算資源を多数のユーザで利用
- 結果：各ユーザが利用可能な計算資源が全体の負荷状況に応じて動的に増減
  - → Amazon EC2 Spot：マシンを「競り落して」使う



では，クラウド上での並列計算はどう動くべきか?

- 対象アプリ：**長時間を要する**大規模な並列科学技術計算
- 計算資源が動的に増減しうるクラウド上では，これらのアプリは，**そのとき利用可能な計算資源に対応して，アプリの計算規模 (＝並列度) を動的に拡張・縮小**して動かなければならない [Chaudhart et al,2006]
  - → しかし，そのような並列アプリを書くのは困難
  - → 処理系による強力なサポートが必須!



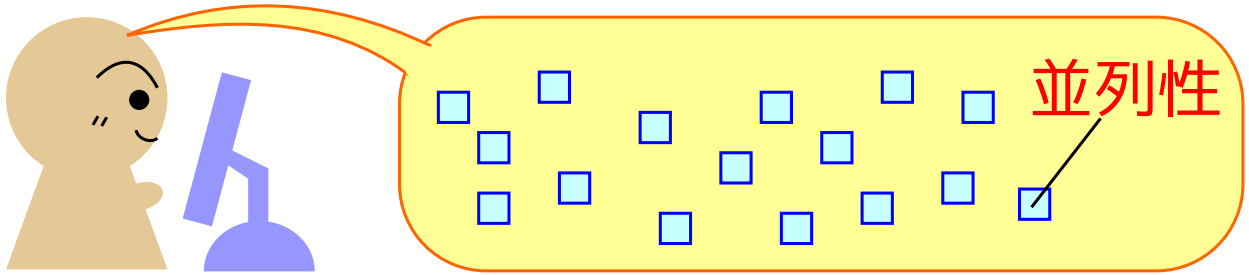
要請される並列分散プログラミングモデル

(1) プログラマは，「単に」アプリの十分な並列性を書けば良い

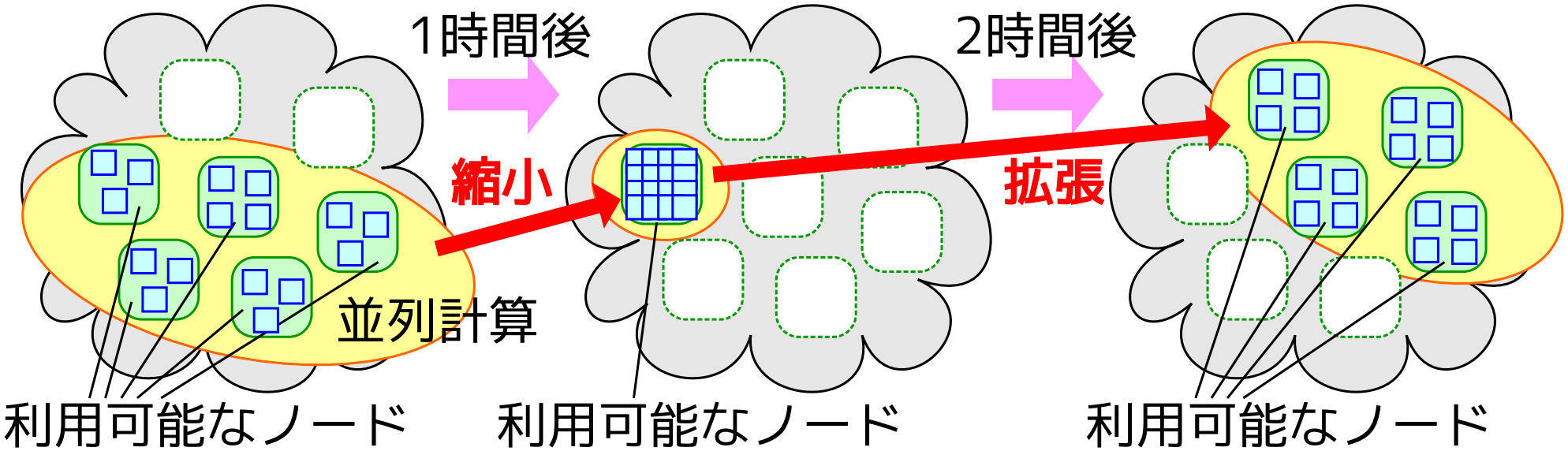
(2) あとは処理系が，それら大量の並列性を利用可能な計算資源に動的にマップして，「透過的に」計算規模を拡張・縮小してくれる

(3) プログラマは，並列インスタンス間のデータ共有を簡単に表現できる

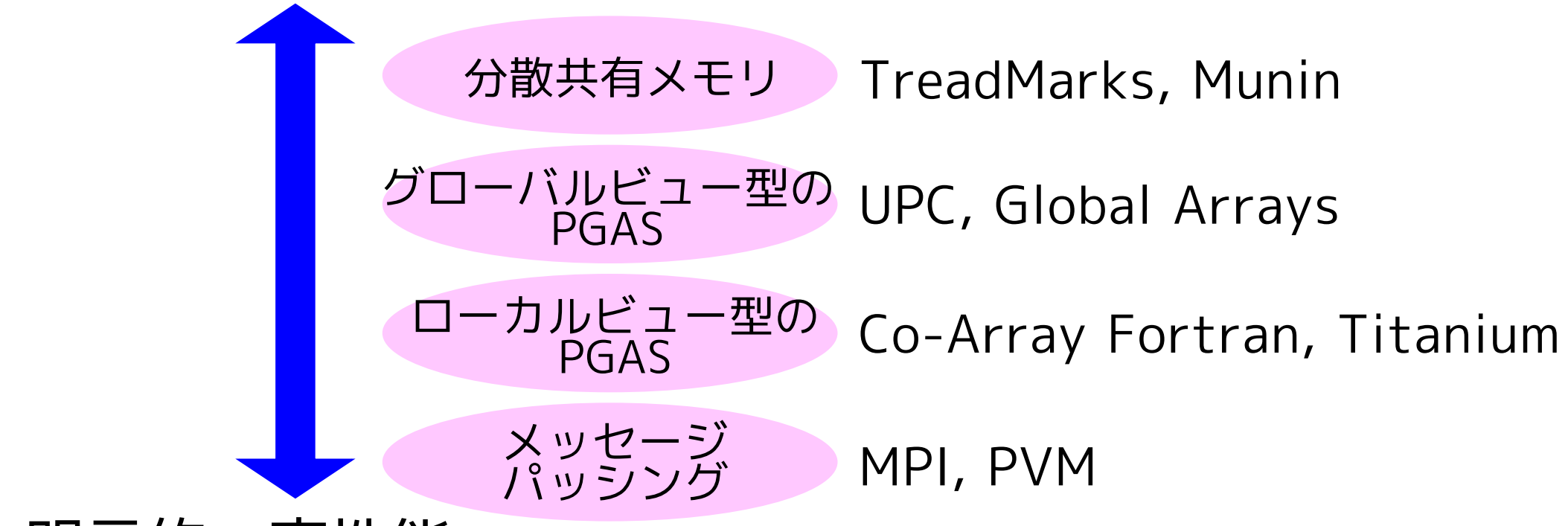
プログラマ：



処理系：



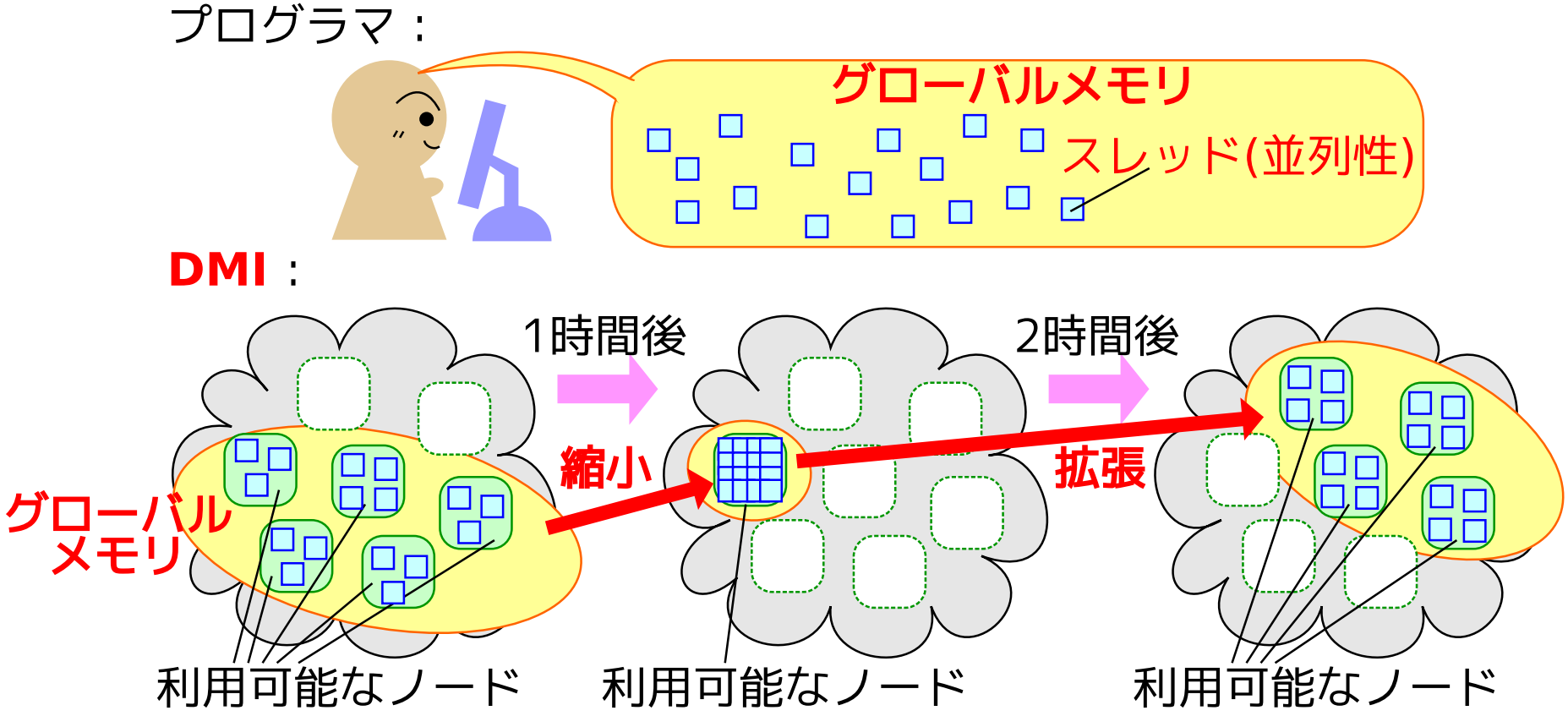
データ共有を簡単に表現させるためには?



- **結論：グローバルビュー型の PGAS モデル**
  - → **簡単：グローバルメモリがあるから**
  - → **高性能：リモートとローカルを明確に区別できるから**

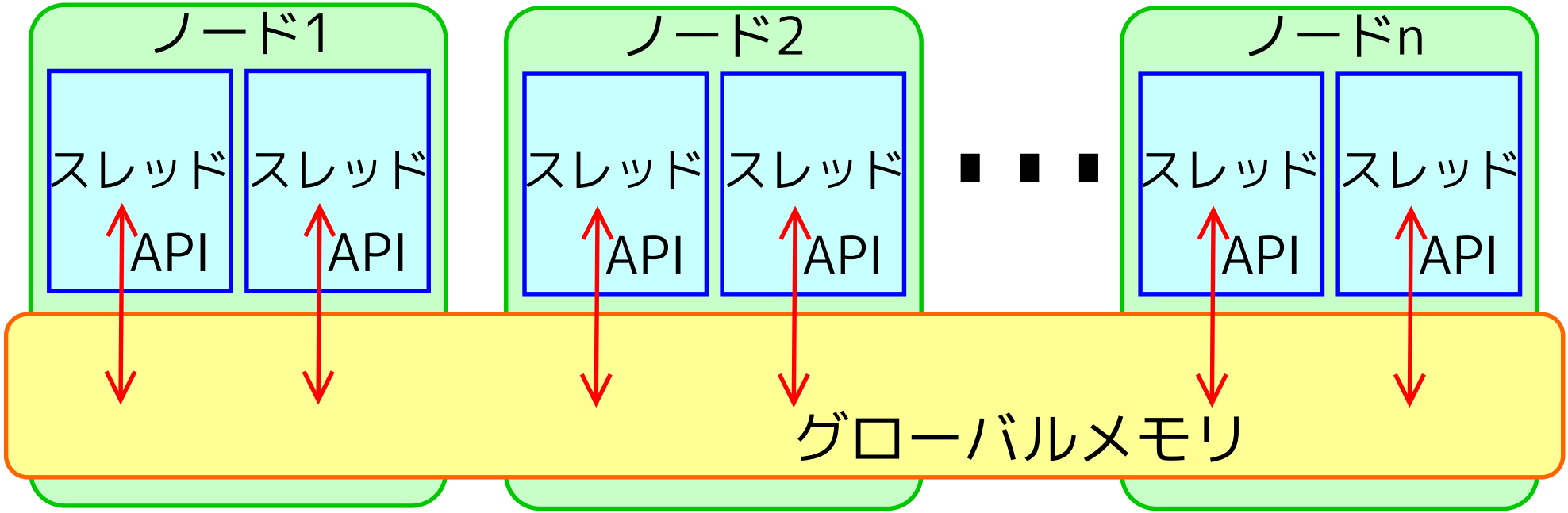
提案：DMI(Distributed Memory Interface)

- **DMI：並列計算の規模を「透過的に」拡張・縮小可能な PGAS 処理系**
  - (1) **プログラマは，「単に」十分な数のスレッドを生成するだけで良い**
  - (2) **あとは DMI が，それら大量のスレッドを利用可能な計算資源に動的にマップして，「透過的に」計算規模を拡張・縮小してくれる**
  - (3) **スレッド間のデータ共有レイヤーとして，高性能なグローバルメモリが提供される**



DMI のプログラミングインタフェース

- C 言語の静的ライブラリ
- **概念的には普通の共有メモリ環境のスレッドプログラミングと同じ：簡単!**
  - → mmap/munmap
  - → read/write
  - → **同期**
  - → **スレッドの** create/join/detach
  - → ... **その他** 86 **個の** API



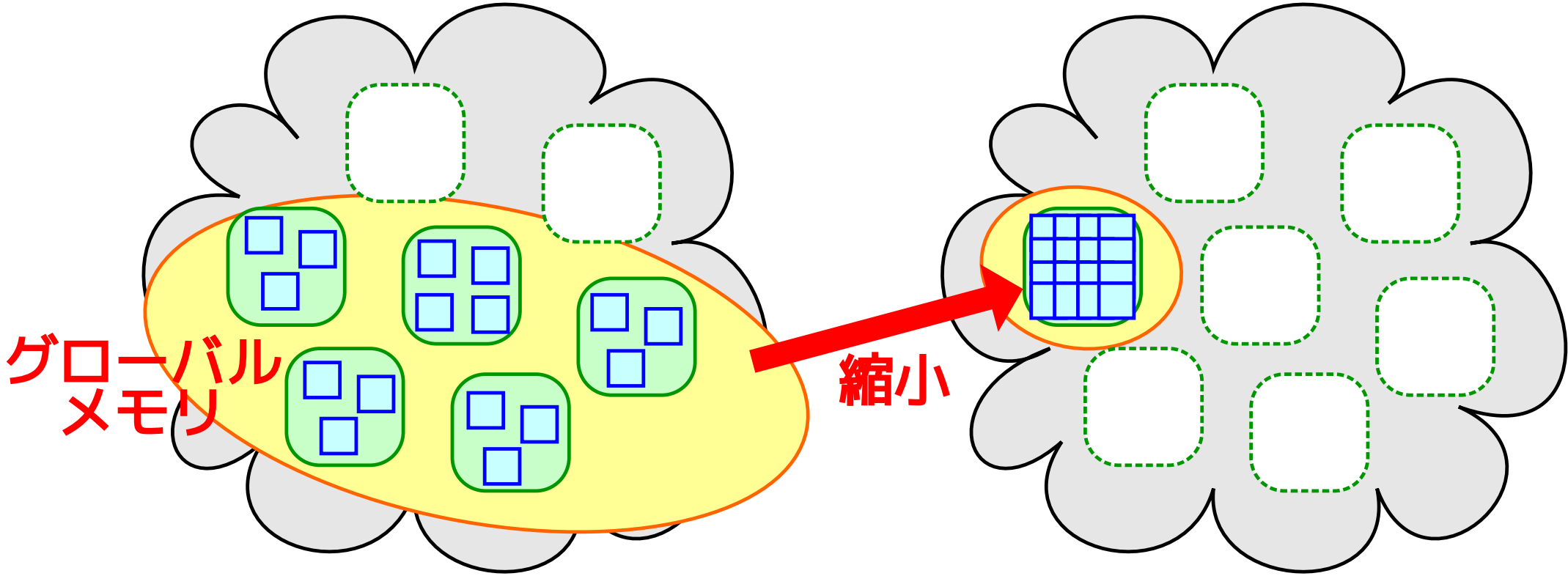
DMI の主要技術

(1) グローバルメモリへの read/write をいかに高性能化するか [PRO 論文誌 2010][HPDC2010]

(2) ノードの動的な参加・脱退を越えて，グローバルメモリのコンシステンシをいかに維持するか [PRO 論文誌 2010]

(3) 生きたスレッドをどう移動させるか?[SWoPP2010]

→ random-address：スレッド移動のための新アドレス管理手法



❖ 関連研究

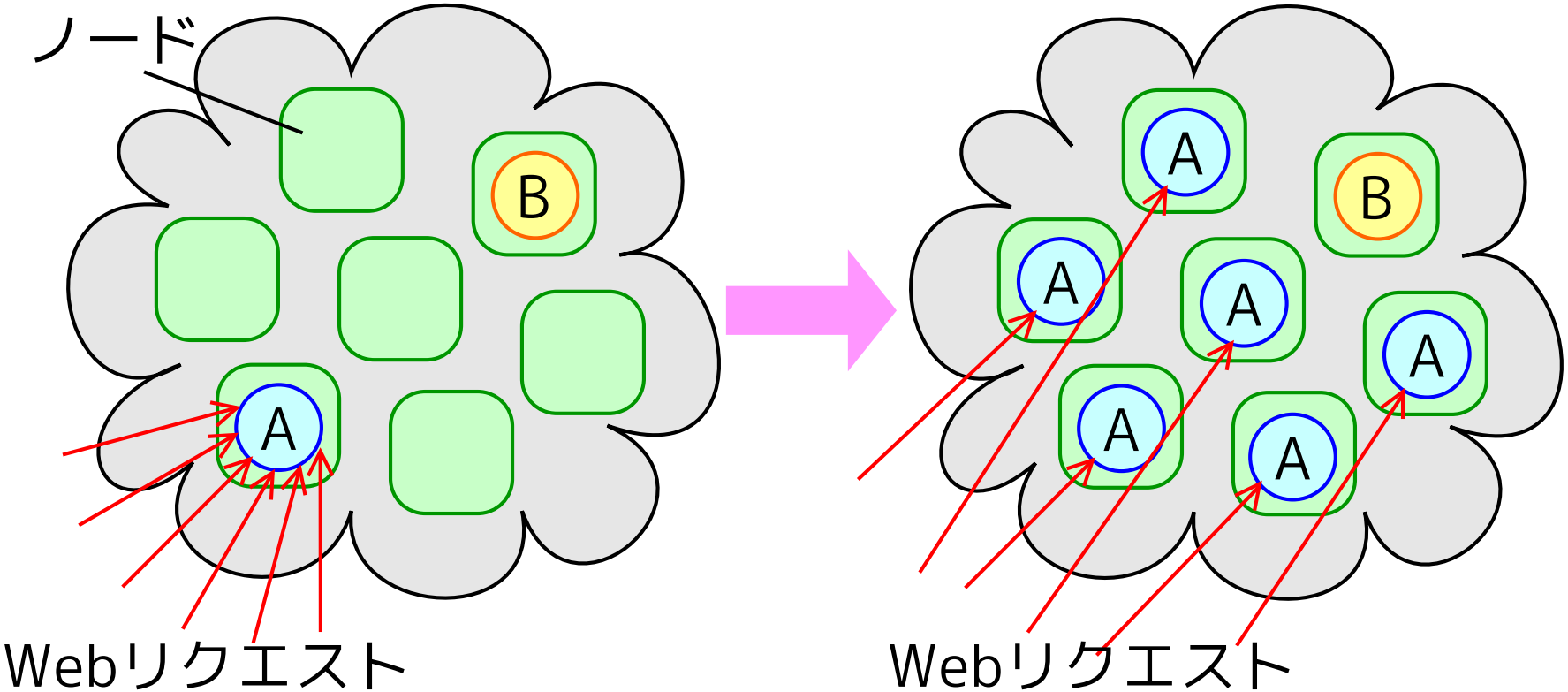
計算を移動する際の粒度

- ポイント：「何」を粒度として計算規模の拡張・縮小 (=計算移動) を実現すべきか?
  - → 候補：仮想マシン，プロセス，スレッド
- 観察：
  - → 計算移動のコストは，移動対象の資源消費量に比例
  - → 資源消費量は，VM＞プロセス＞スレッド
  - → たいていの並列科学技術計算は，プロセス or スレッドのレベルで処理が完結しているので，OS レベルでの機能の移動までは要求されない [Chaudhart et al,2006]
    - ♦ 仮想マシンの移動は「必要以上に重すぎる」
      - 例：Amazon EC2，Windows Azure
- 結論：プロセス or スレッドを粒度とした計算移動が適切

関連研究 1：Google App Engine(GAE)

- Google の効率的なインフラ上で Web アプリを実行可能なサービス
- Web リクエスト数の増減に応じて，自動的かつ高速に処理プロセス数が拡張・縮小される
- 30 秒ルール：各 Web リクエストは短時間 (30 秒以内) で終了されなければならない

→ 欠点：長時間を要する並列科学技術計算をサポートできない



関連研究 1：GAE にはなぜ 30 秒ルールが必要なのか?

- そもそもの要請：ユーザ間で高速に (=応答性良く) 計算資源をスケジューリングしたい
- GAE にはなぜ 30 秒ルールが必要なのか?
  - → システム：各 Web リクエスト単位でしかスケジューリングできない
  - → 結果：各 Web リクエストの実行時間を制限する必要が出てくる
- DMI にはなぜ 30 秒ルールがいらないのか?
  - → システム：任意のタイミングでスケジューリングできる
    - ♦ スレッド実行中の「(ほぼ) いつでも」スレッドを移動できる
  - → 結果：各スレッドの実行時間を制限しなくて OK
    - ♦ 長時間を要する並列科学技術計算をサポート可能!

関連研究 2：MPI のプロセス移動

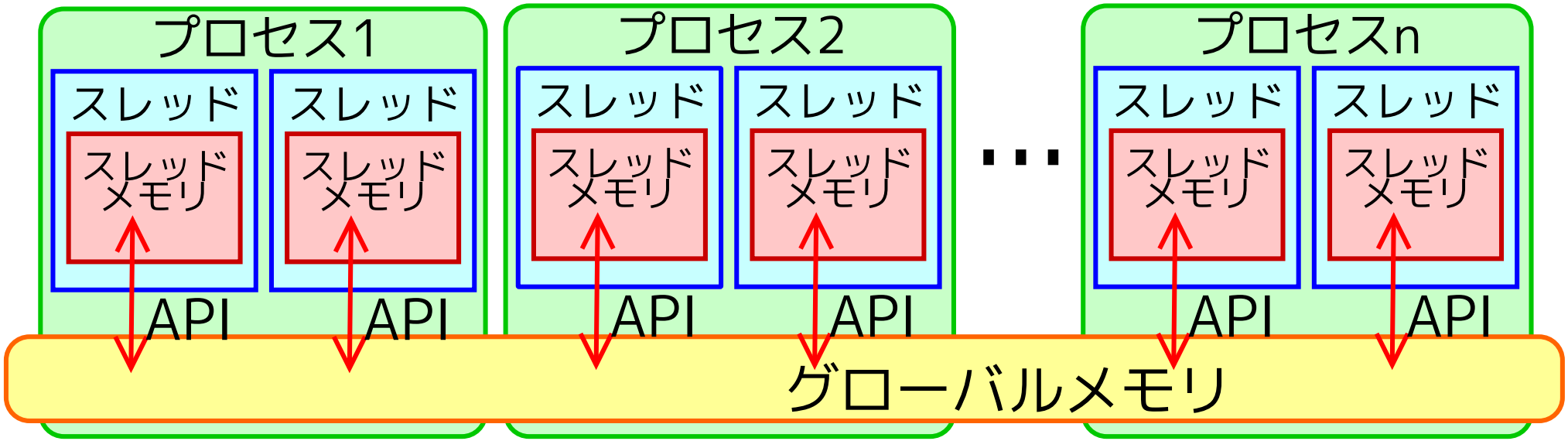
- Adaptive MPI[Huang et al,2003]，MPI Process Swapping[Sievert et al,2004]
  - → 目標：計算環境の負荷状況に応じて，処理系が MPI プロセス (or スレッド) を動的に移動させることで，クラウドのような動的な計算環境における負荷分散を透過的に実現
- DMI との比較：
  - → 目標は同じ
  - → しかしプログラミングの簡単さは，(DMI における)PGAS モデル ≫(MPI における) メッセージパッシングモデル
- DMI の新規性：

(1) 「(同じ) 目標を (より簡単な)PGAS モデルでいかに実現するか」

(2) その要素技術としての，スレッド移動のための新アドレス管理手法：random-address

❖ 提案手法：random-address

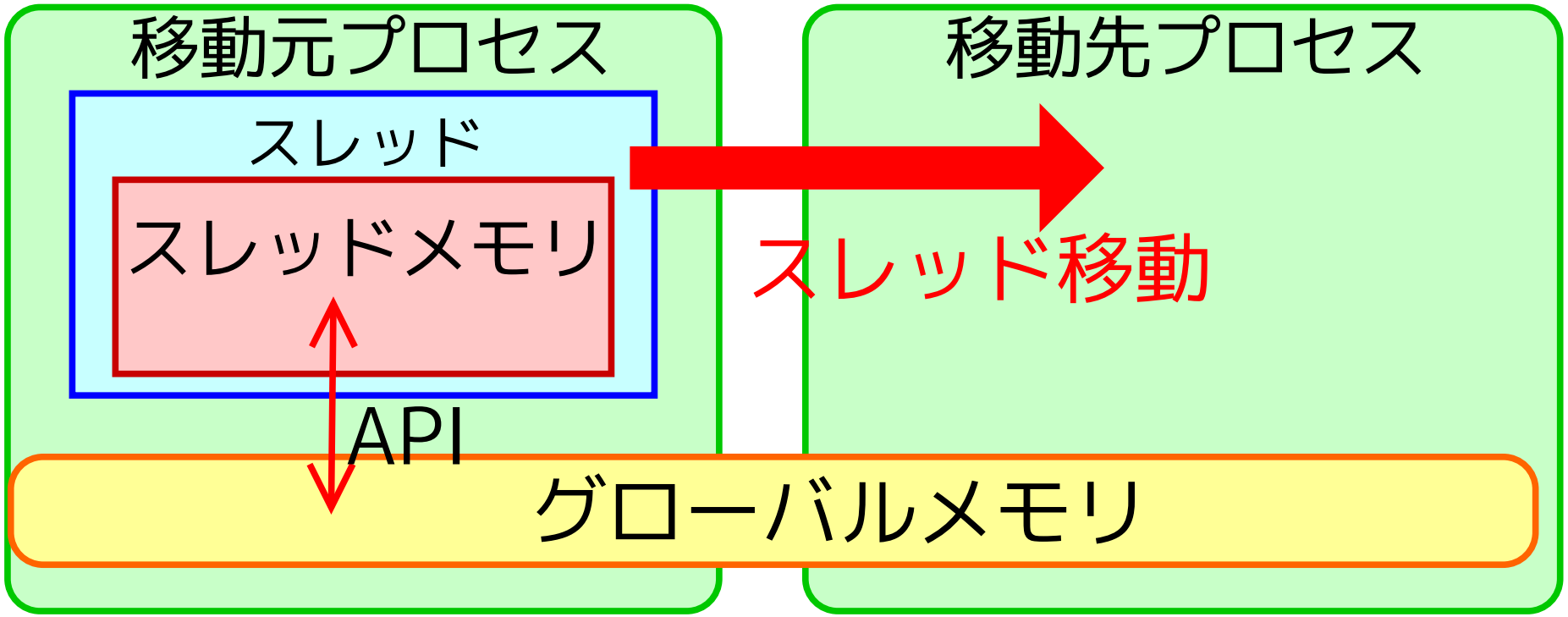
DMI の概要



- **マルチスレッド型のユーザレベル PGAS 処理系**
  - → SPMD **ではない**
  - → DMI_create()/DMI_join() **でスレッドの生成・回収**
- **スレッドメモリとグローバルメモリを明確に分離**
  - → **スレッドメモリ：通常の** malloc()/free() + **通常の** read/write
  - → **グローバルメモリ：**DMI_mmap()/DMI_munmap() + DMI_read()/DMI_write()
- **プロセスの動的な参加・脱退をサポート**

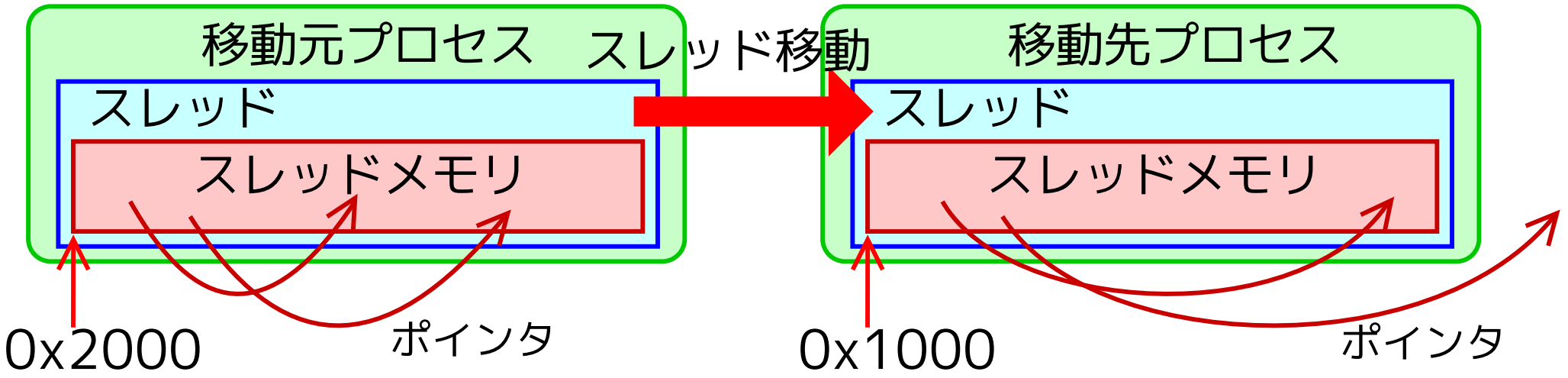
DMI におけるスレッド移動

- スレッドを別プロセスに移動



スレッド移動における「一般的な」問題点

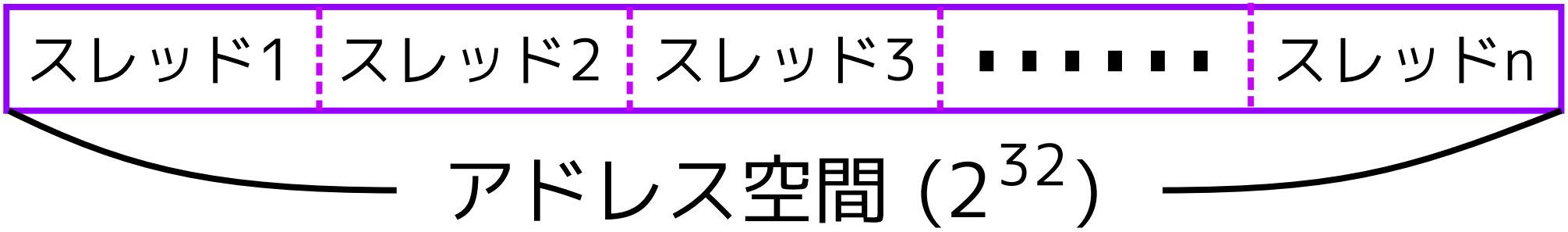
- スレッドが使っているアドレス領域を，移動元プロセスと移動先プロセスとで完全に一致させないと，ポインタが無効化してしまう
- 何の対策も取らなければ，移動元プロセスでスレッドが使っているアドレス領域が移動先プロセスで空いている保証などない



既存研究における解決策

- 解決策 1：移動直後に，移動先プロセスのアドレス領域に合わせて全ポインタの値を正しく更新する [Cronk et al,1997]
  - → C 言語は型安全ではないので全ポインタを正しく追跡するのは困難
- 解決策 2(iso-address)：アドレス領域全体 (例：2^{32}) を静的に分割して，各スレッドが使えるアドレス領域を予め決め打っておく [Weissman et al,1998]
  - → 各スレッドが使うアドレスのグローバルな一意性を保証
  - → 移動先プロセスでは必ず同一アドレス領域に配置できるので，ポインタ無効化の問題が起きない
  - → 従来の見解：「32bit 環境では非現実的だが，64bit 環境なら大丈夫」

[Itzkovitz et al,1998][Weissman et al,1998][Thitikamol et al,1999]



本当に 64bit 環境ならば大丈夫なのか?

- 多くの 64bit 環境で使用可能なアドレス空間は 2^{47} バイト

スレッド数　　各スレッドが使用可能なメモリ量

2^{47}バイト ＝ 8192 × 64GB

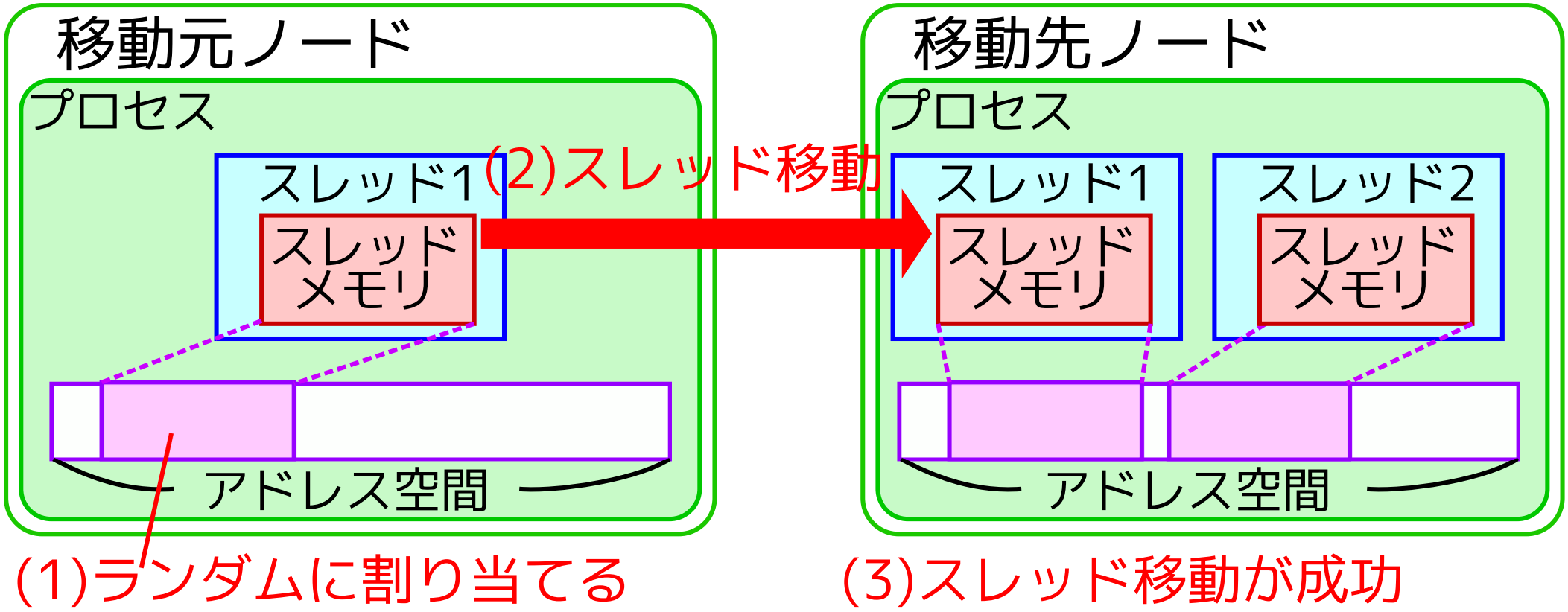
2^{47}バイト ＝ 1024 × 512GB

- これらの数字は現在の超並列環境では十分現実的
  - → 「限界が近い」
- アドレス空間の大きさに制限されないスレッド移動手法が必須
  - → たとえハードウェアの進化に伴って 2^{47} という数字が増えるとしても必要

基本アイディア：random-address (1)

(1) 各スレッドが使うアドレスをランダムに決定

(2)「運が良ければ」スレッド移動時にアドレスが衝突しない



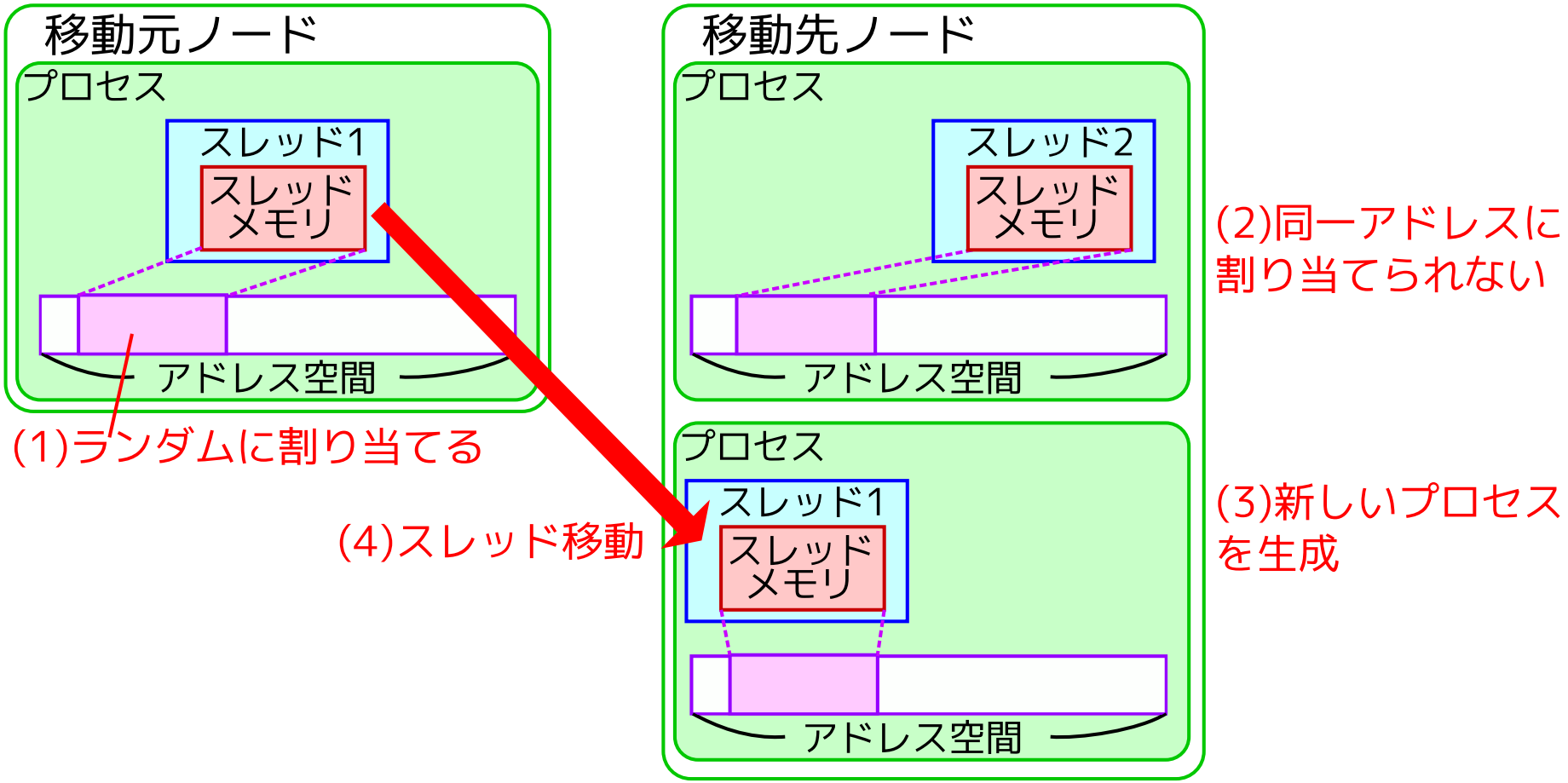
基本アイディア：random-address (2)

(1)「運が悪ければ」スレッド移動時にアドレスが衝突

(a) 移動先ノードに新しいプロセス (=新しいアドレス空間) を生成

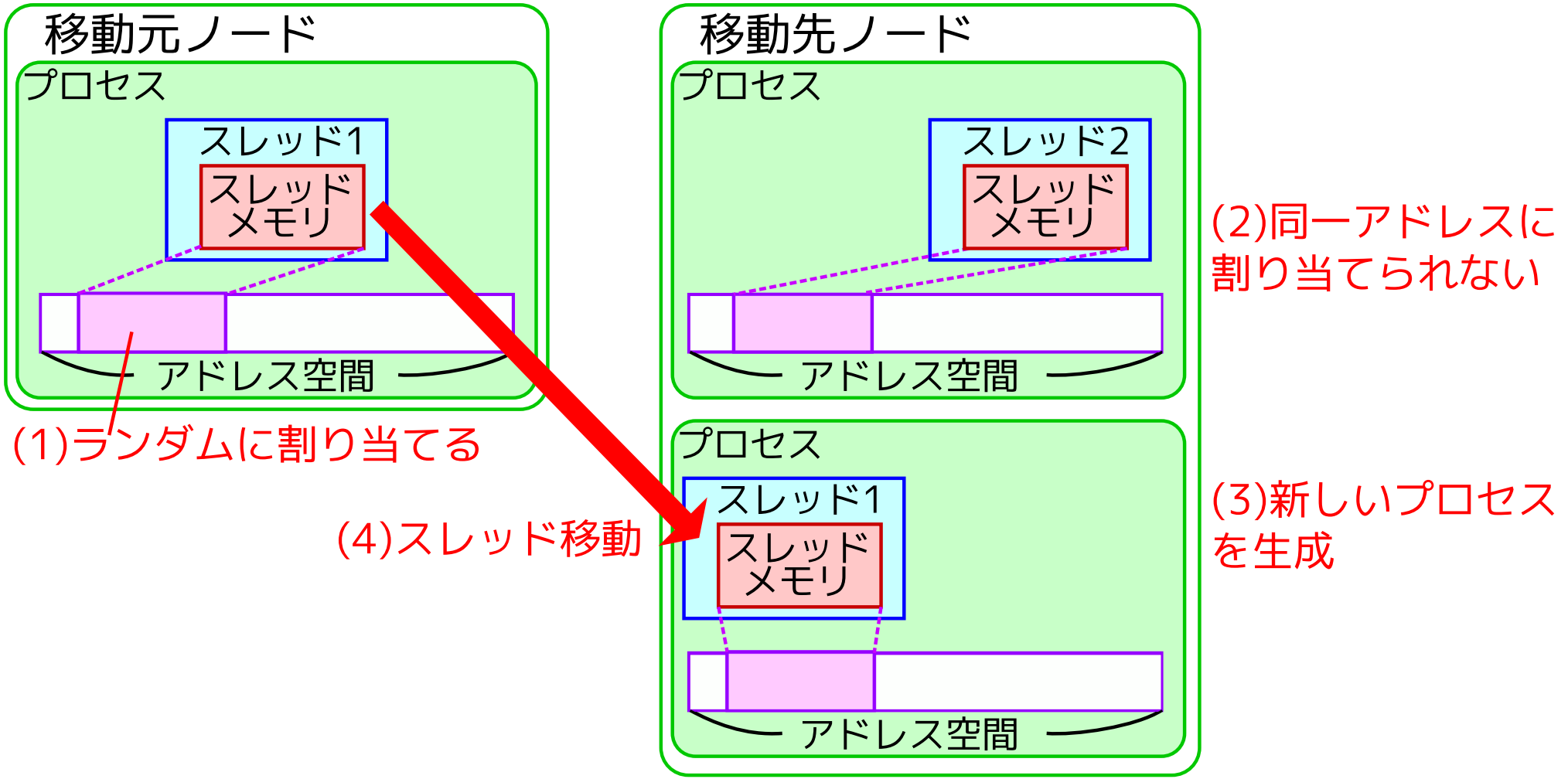
(b) その新しいプロセスの中へスレッドを移動

➤ DMI がプロセスの動的な参加・脱退に対応しているからこそ実現できる手法



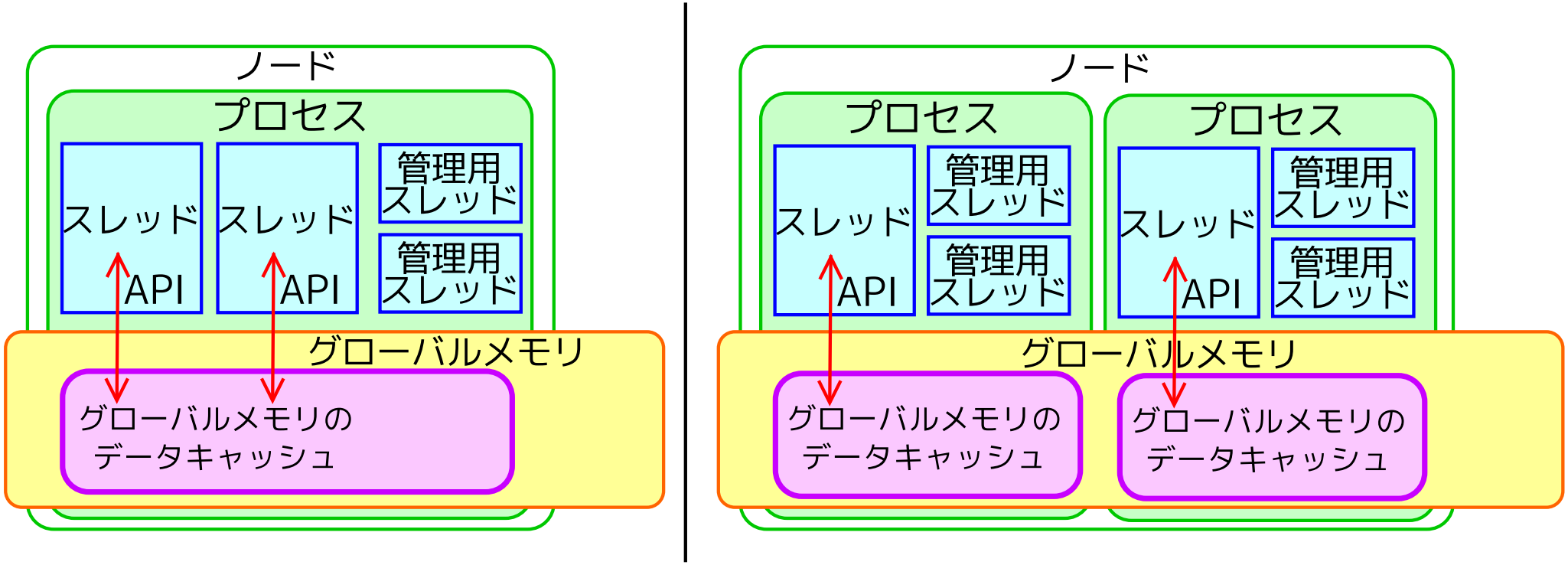
アドレスが衝突すると?

➤ **アドレスが衝突するとノード内プロセス数が増える**



ノード内プロセス数が増えると?

- **ノード内プロセス数が増えると性能が劣化**
  - → **一般論：スレッドよりプロセスの方が重いから**
  - → **理由 1：管理用スレッドが無駄に増えてしまうから**
  - → **理由 2：グローバルメモリのデータキャッシュをスレッド間で共有できなくなるから**

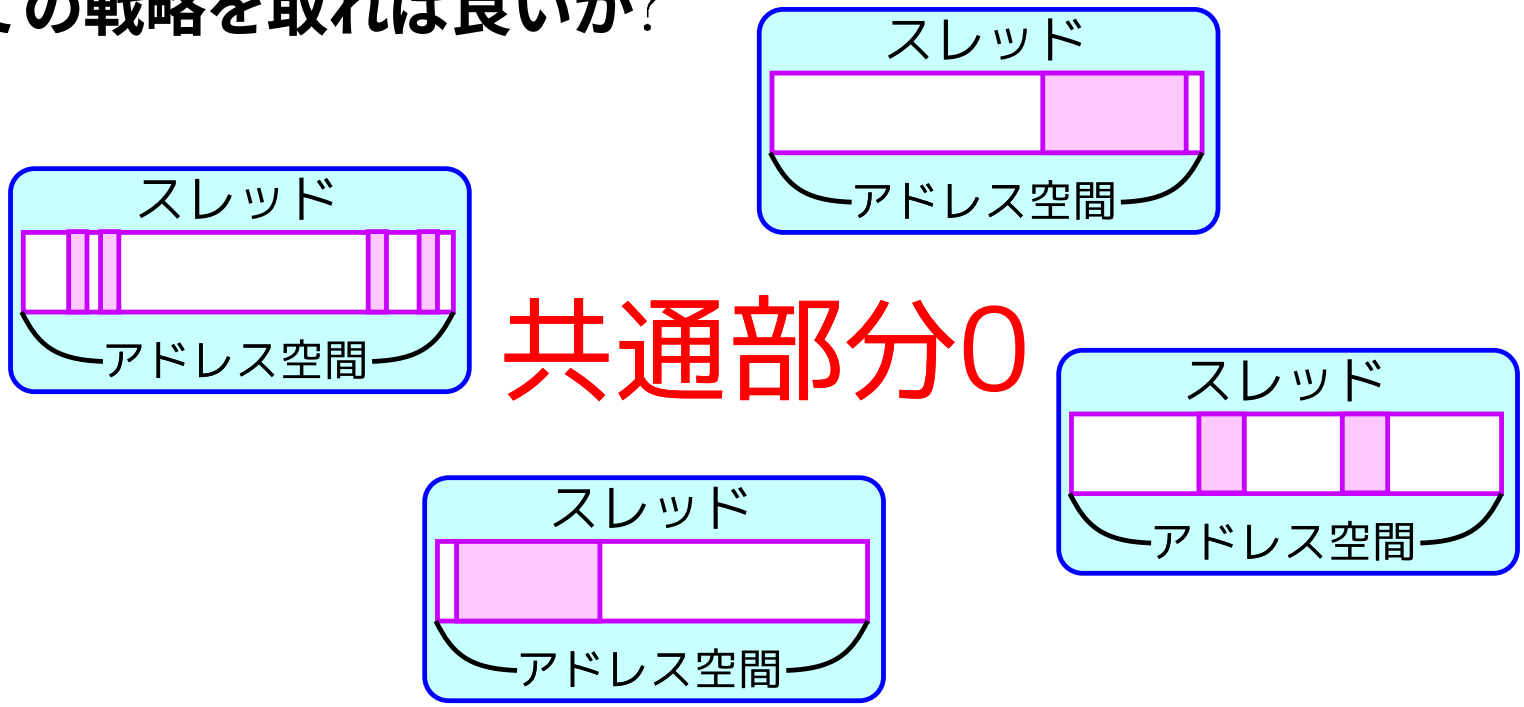


random-address におけるポイント

- 事情の整理：
  - → アドレスが衝突するとノード内プロセス数が増える
  - → ノード内プロセス数が増えると性能が劣化する
- 結論：アドレス衝突確率を最小化するための工夫が必須
  - → 各スレッドが使うアドレスをランダムに決めるとはいえ「デタラメ」ではダメ

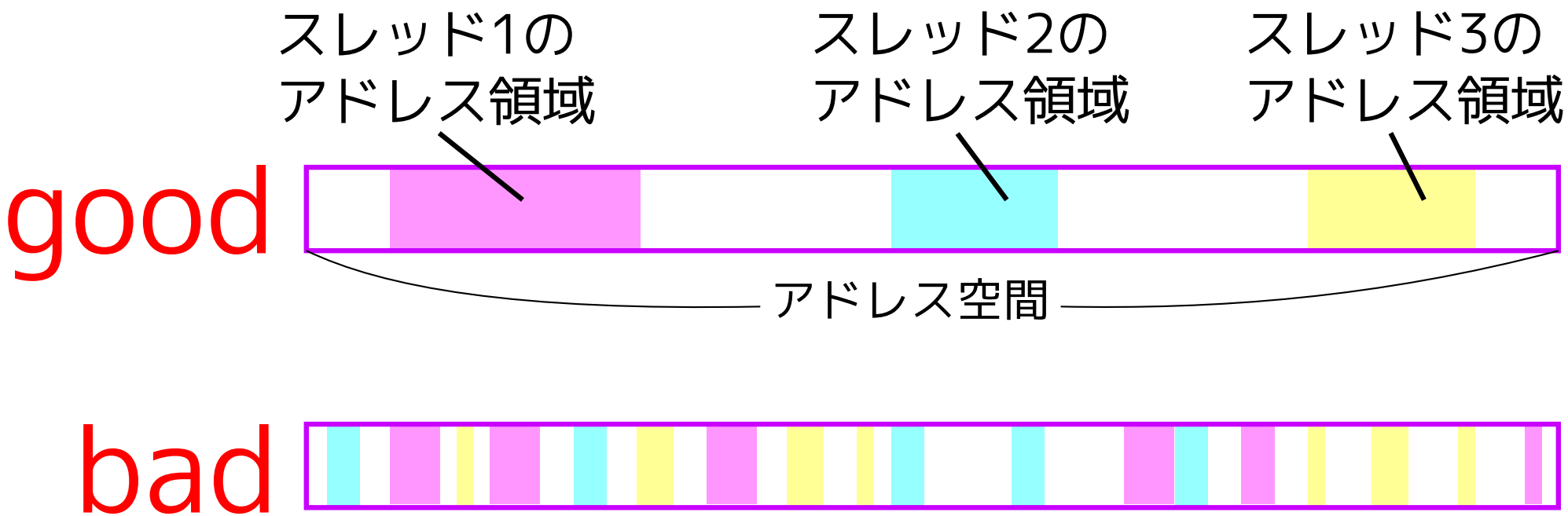
アドレス衝突確率の最小化：考えるべき問題

- 状況：
  - → たくさんのスレッドがある
  - → 各スレッドは，他のスレッドがどのようなアドレス集合を使っているかの知識を持たない
- 問題：「どの 2 本のスレッド i とスレッド j に対しても，スレッド i が使っているアドレス集合とスレッド j が使っているアドレス集合が共通部分を持たない確率」を最大化するためには，各スレッドはどのようなアドレス割り当ての戦略を取れば良いか?



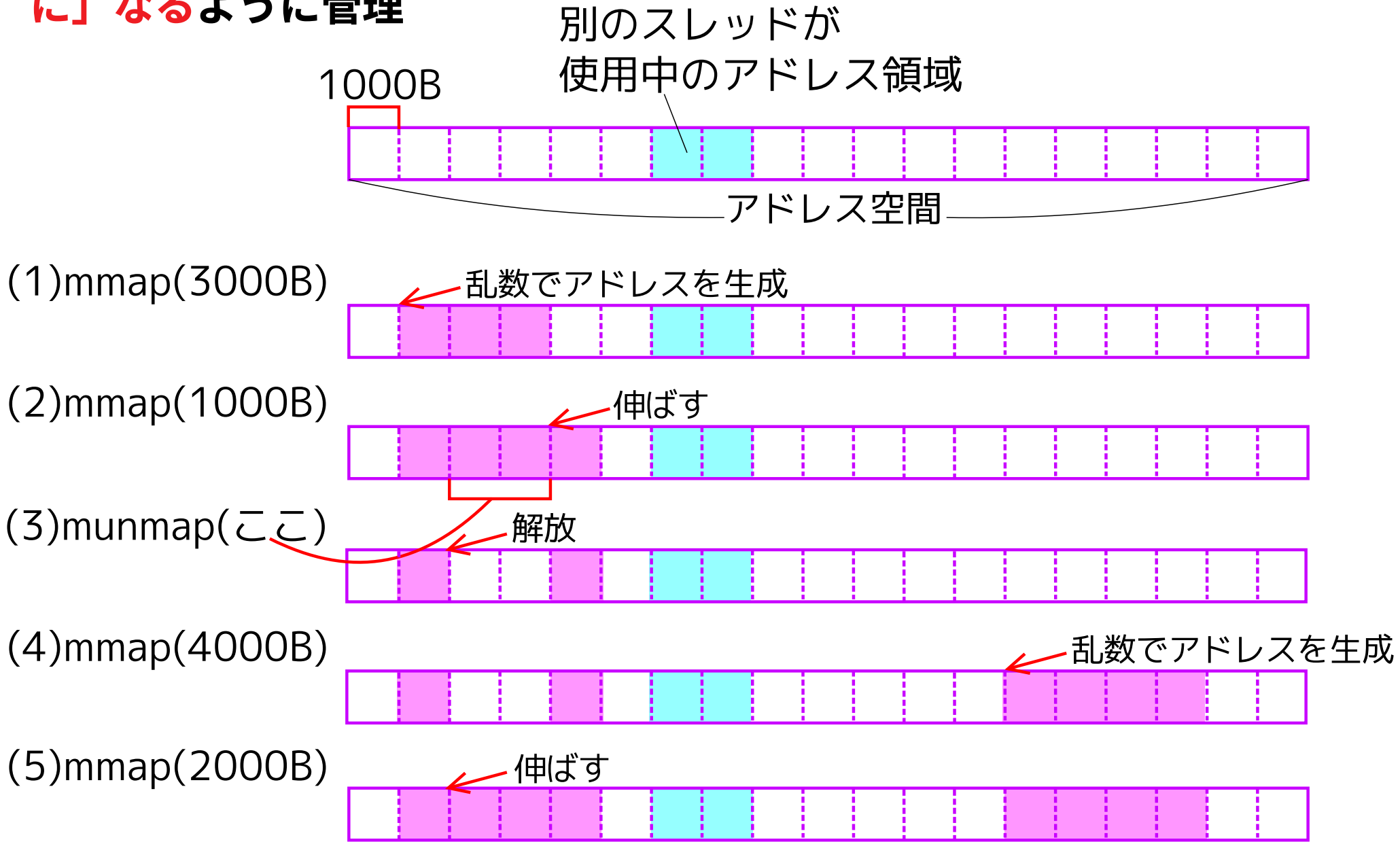
アドレス衝突確率の最小化：最適な戦略

- 最適な戦略 (の 1 つ)：全スレッドがアドレスを「連続的に」使う
  - → 問題の厳密な定義，数学的な証明は論文を参照



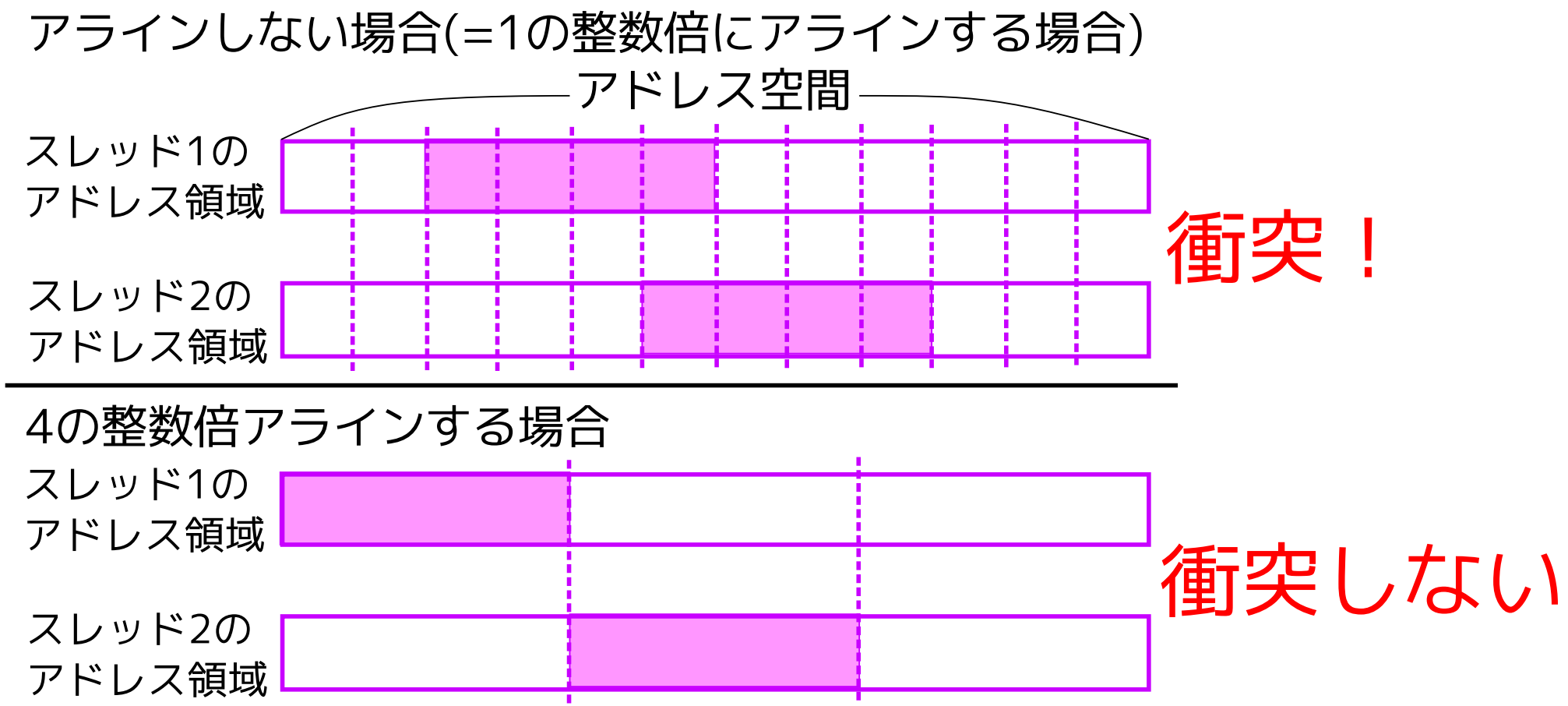
アドレスアロケータのアルゴリズム

- 各プロセス内で，各スレッドが使うアドレス領域ができるかぎり「連続的に」なるように管理



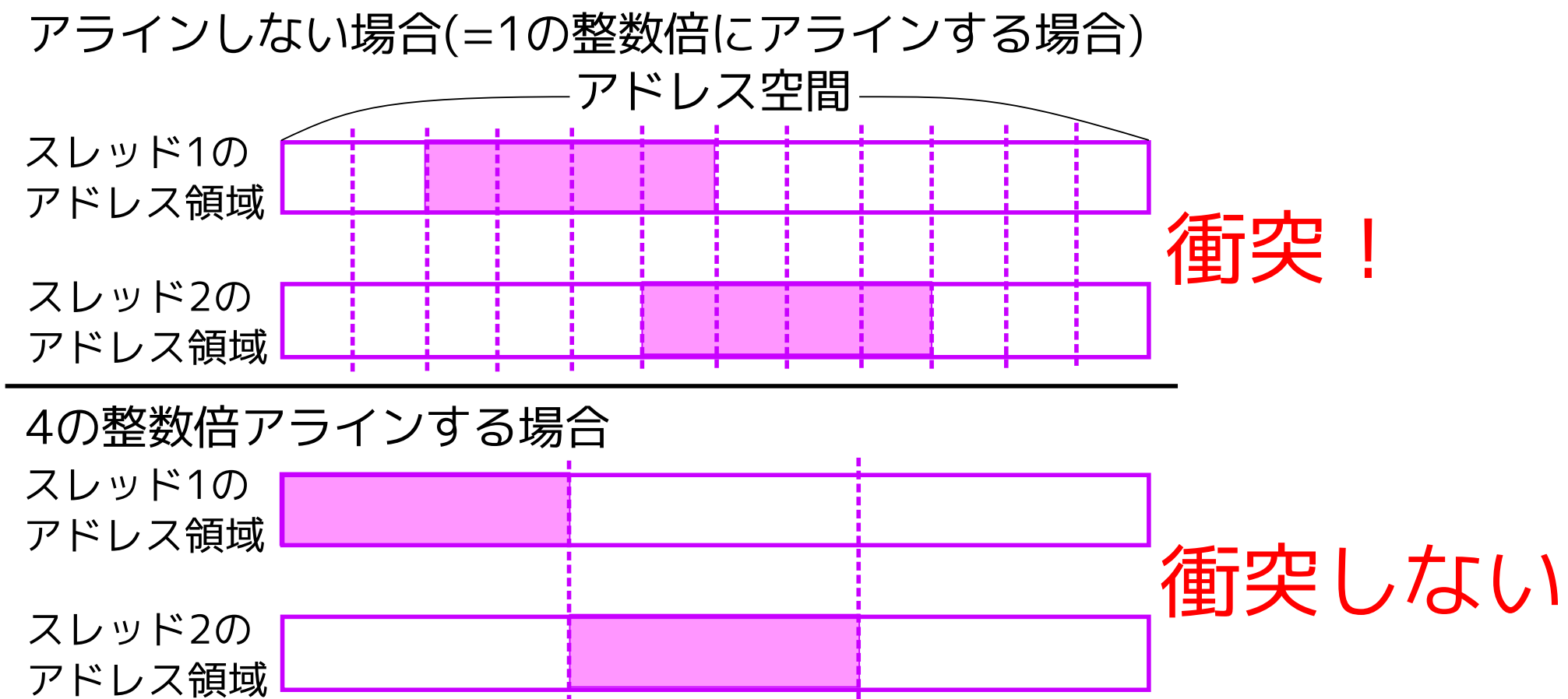
(特定の知識のもとで) さらに最適な戦略 (1)

- **整数 x を「うまく」選んで，各スレッドが使うアドレス領域の開始アドレスを x の整数倍にアラインさせる**
  - → **2 本のスレッドのアドレス集合がごく一部だけ $(=$「惜しく」$)$ 衝突することに起因するアドレス衝突が起きにくくなるため，全体としてのアドレス衝突確率が下がる**



(特定の知識のもとで) さらに最適な戦略 (2)

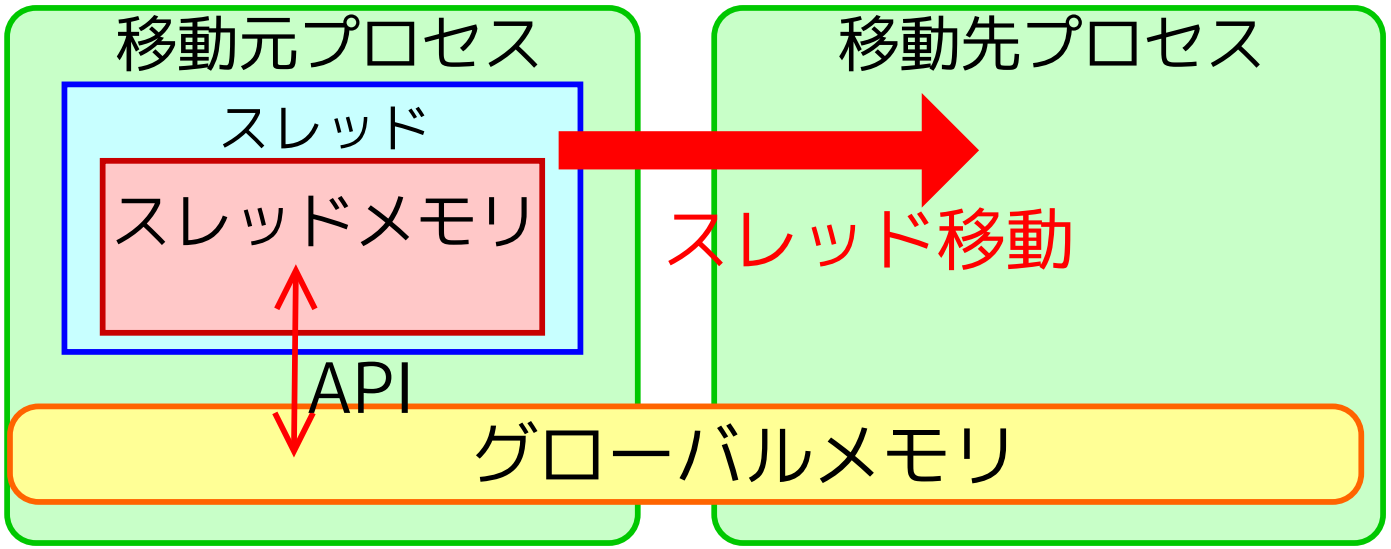
- アライン x の最適値は?
  - → 各スレッドの使用メモリ量に依存
  - → 自明な例：全スレッドが常に m バイトを使うとわかっているならば $x = m$ が最適
  - → 実際には，各スレッドの使用メモリ量の予測値に基づいて x を決定



プログラミングインタフェース

プログラミングインタフェース (1)

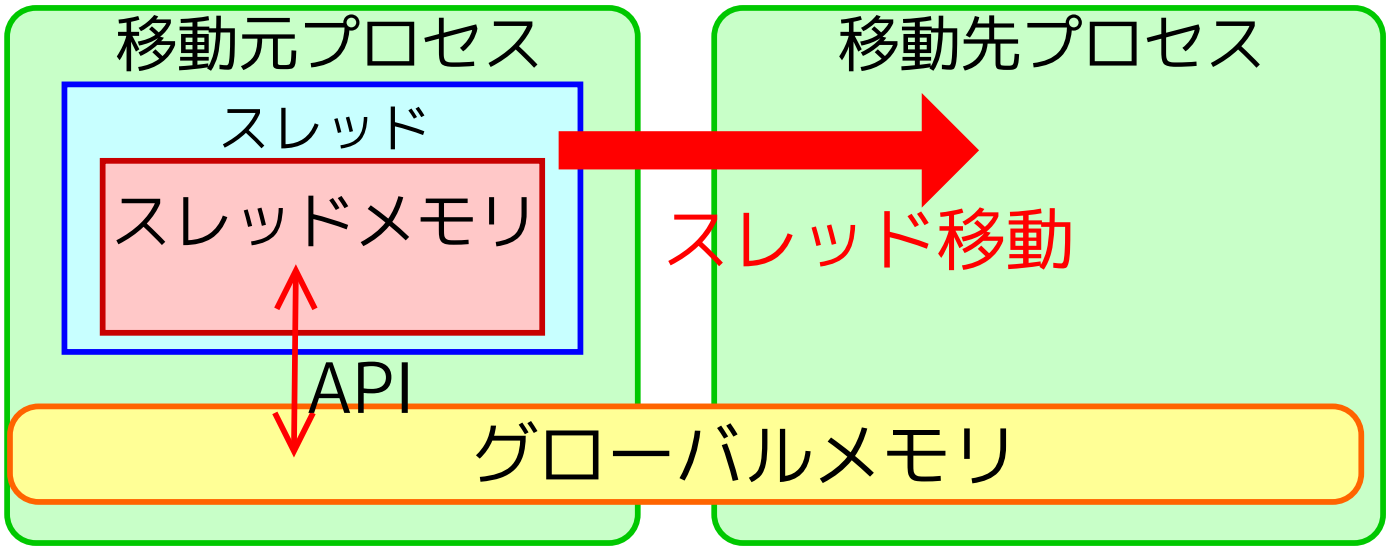
```
DMI_thread() {
  for(iter=0; ; iter++) {
    ...;
    DMI_yield();
    ...;
  }
}
```



- **協調的なスレッド移動**
  - → DMI_yield() **が呼ばれたとき，スレッド移動の必要があれば** DMI_yield() **の「中」でスレッド移動**

プログラミングインタフェース (2)

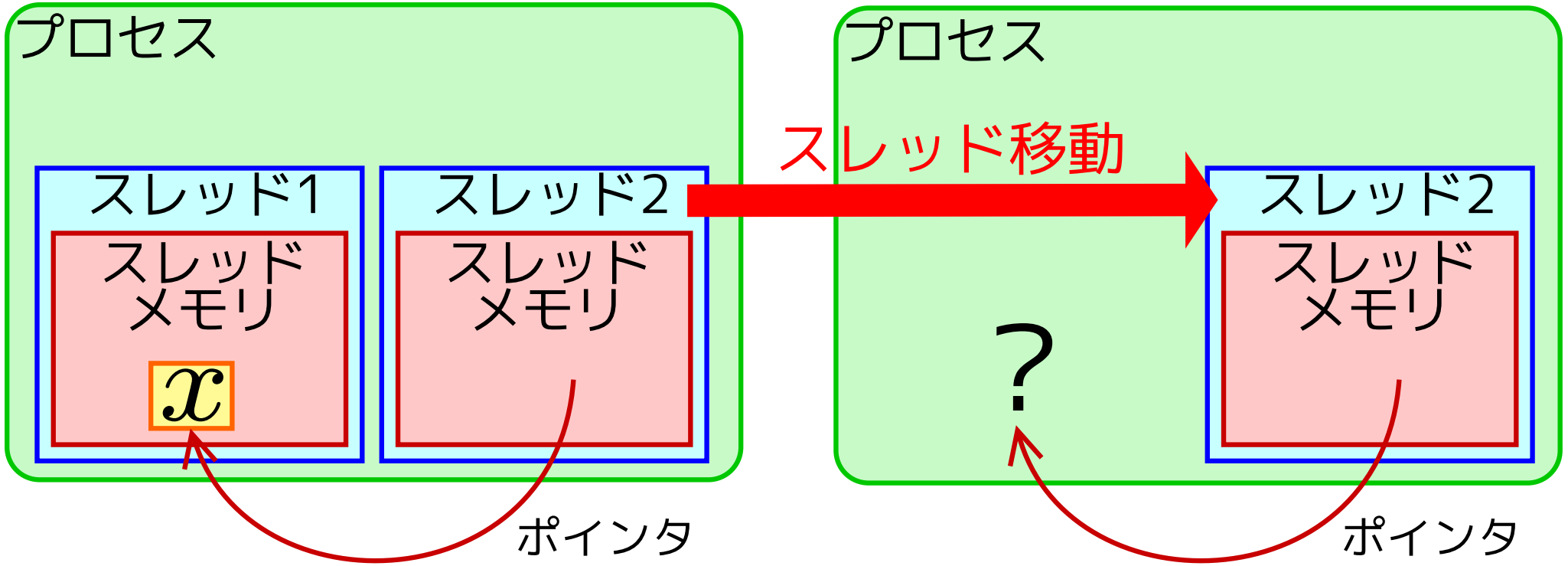
```
DMI_thread() {
  for(iter=0; ; iter++) {
    ...;
    DMI_yield();
    ...;
  }
}
```



- プログラム側では**必要十分に短い間隔で DMI_yield() を呼び出す「だけ」**
  - → (計算規模の拡張・縮小非対応の)DMI プログラムに対する**変更はわずか**
- …… ただし，実は「若干の制約」がある (なぜか?)

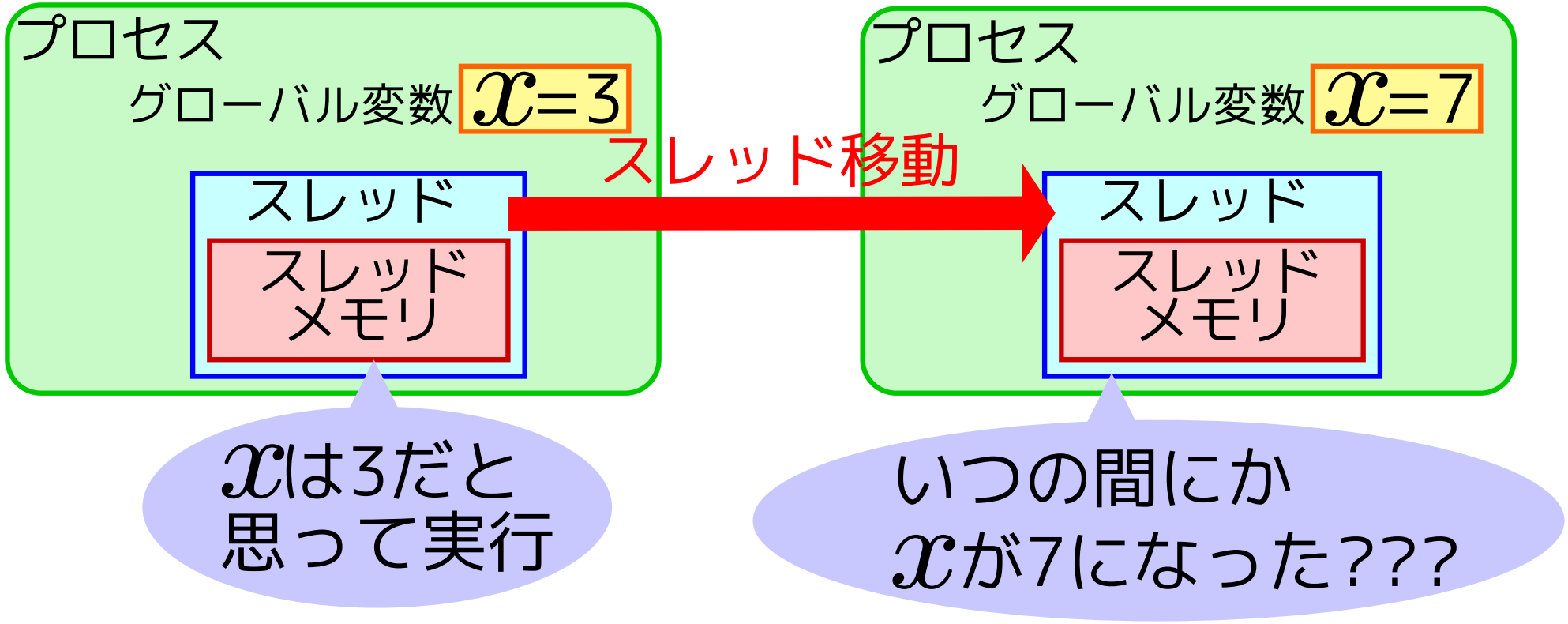
スレッド移動の失敗例 (1)

- 別のスレッドのスタック領域を使っているスレッドを別プロセスに移動させたら…
  - → 破綻!



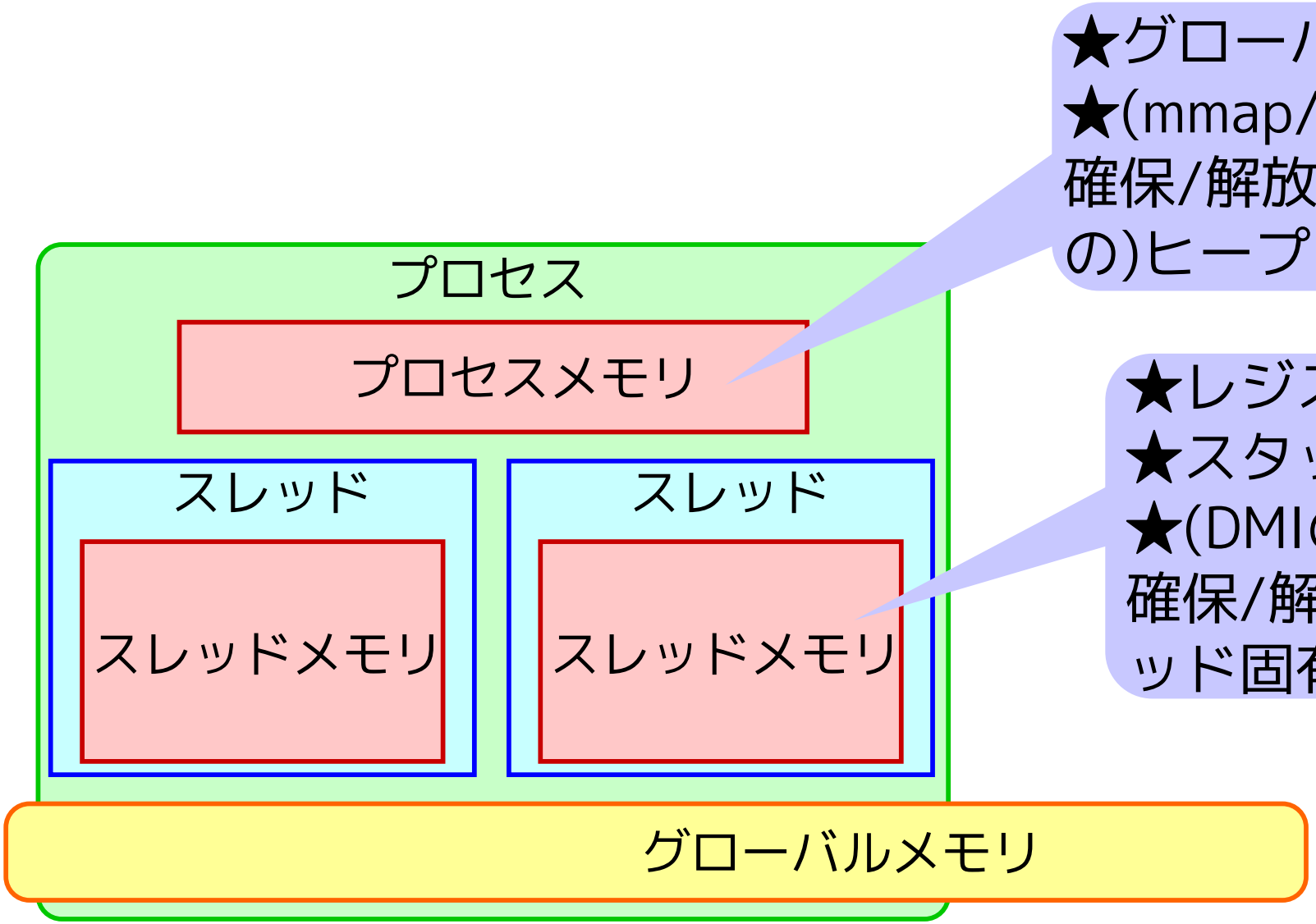
スレッド移動の失敗例 (2)

- グローバル変数を使っているスレッドを別プロセスに移動させたら…
  - → グローバル変数も一緒に移動させる?→ 移動先プロセスのグローバル変数の値を書き潰してしまうので破綻!
  - → グローバル変数を移動させない?→ 破綻!

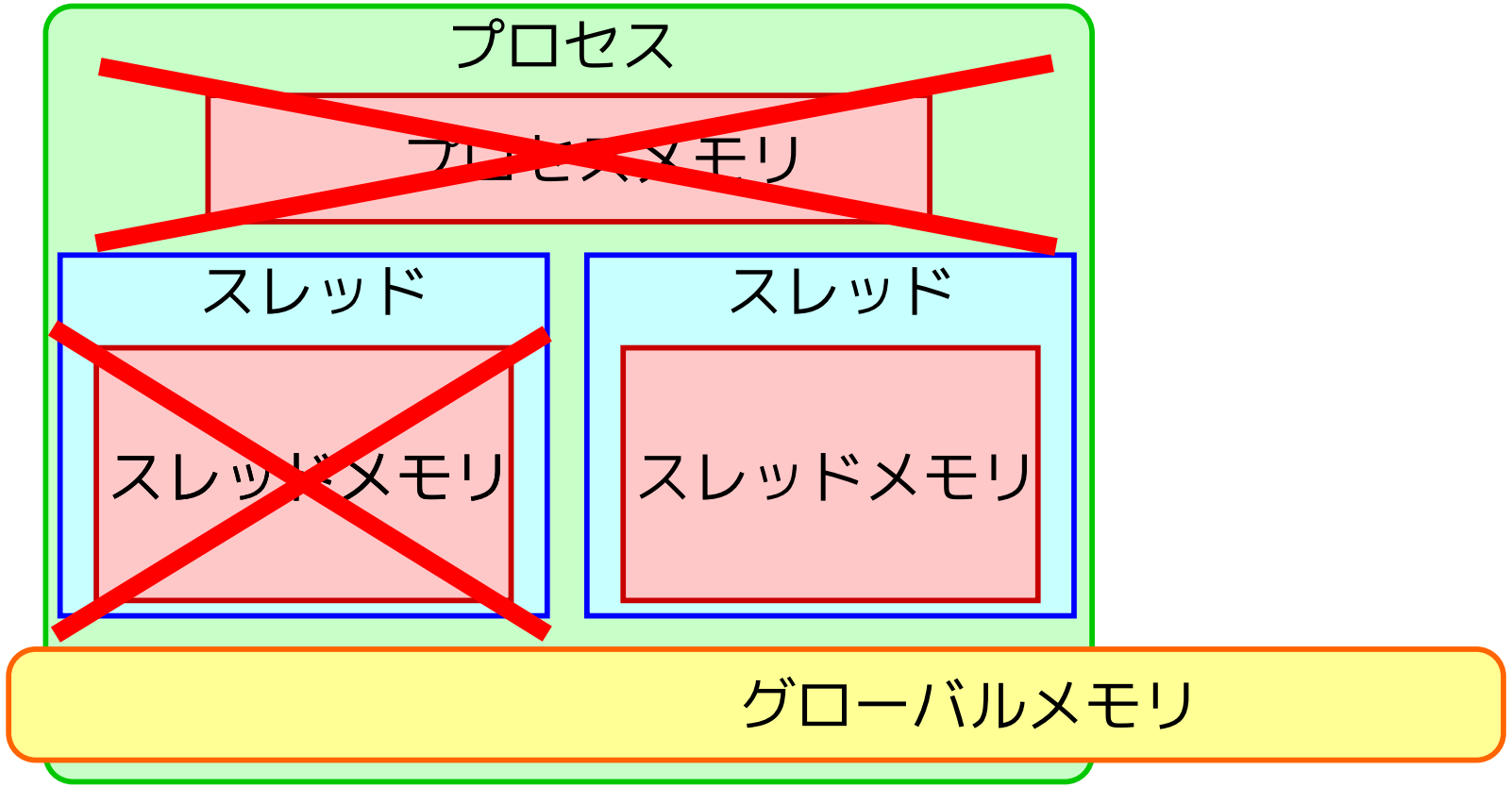


- 「C 言語で書けること全て」をサポートできるわけではない
  - → 何らかの合理的な「プログラミング制約」が必要

準備：メモリ領域のモデル化

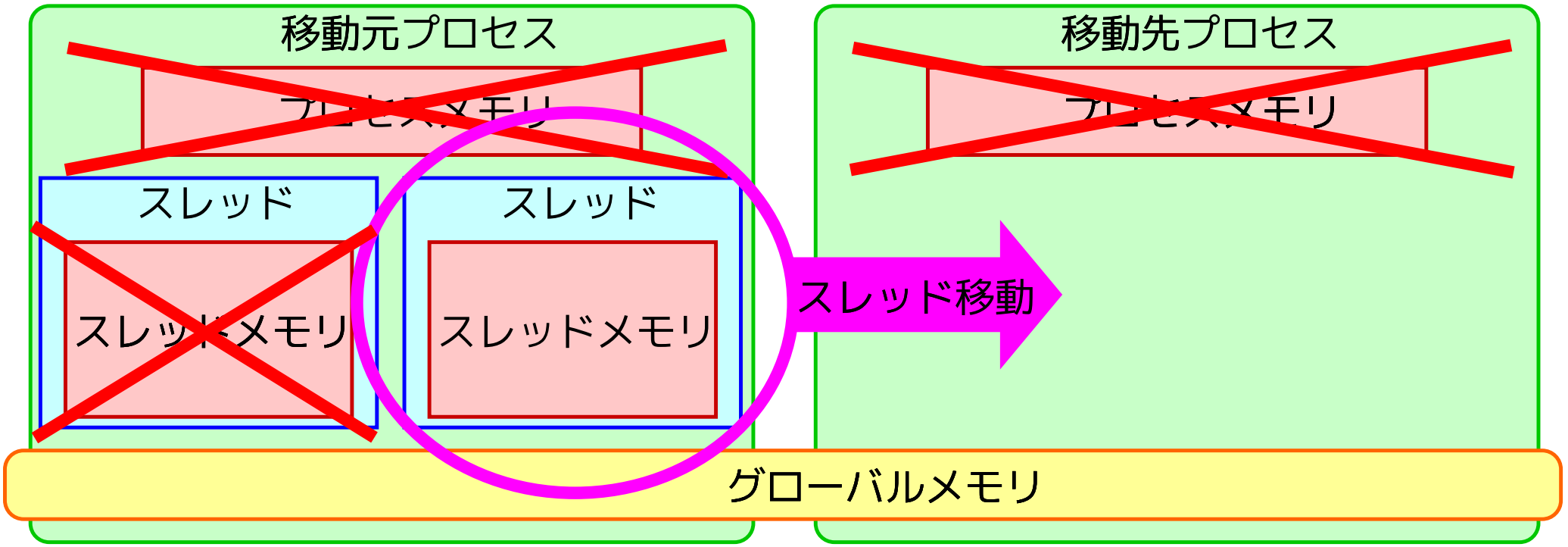


プログラミング制約



- **制約：DMI_yield() を呼び出す「瞬間」には，スレッドの実行状態が，そのスレッドのスレッドメモリまたはグローバルメモリに存在しなければならない**
  - → **つまり，他のスレッドメモリやプロセスメモリに実行状態が存在してはダメ**
- DMI_yield() **呼び出し時「以外」ではプロセスメモリを使っても** OK

この制約のもとでの「安全な」スレッド移動

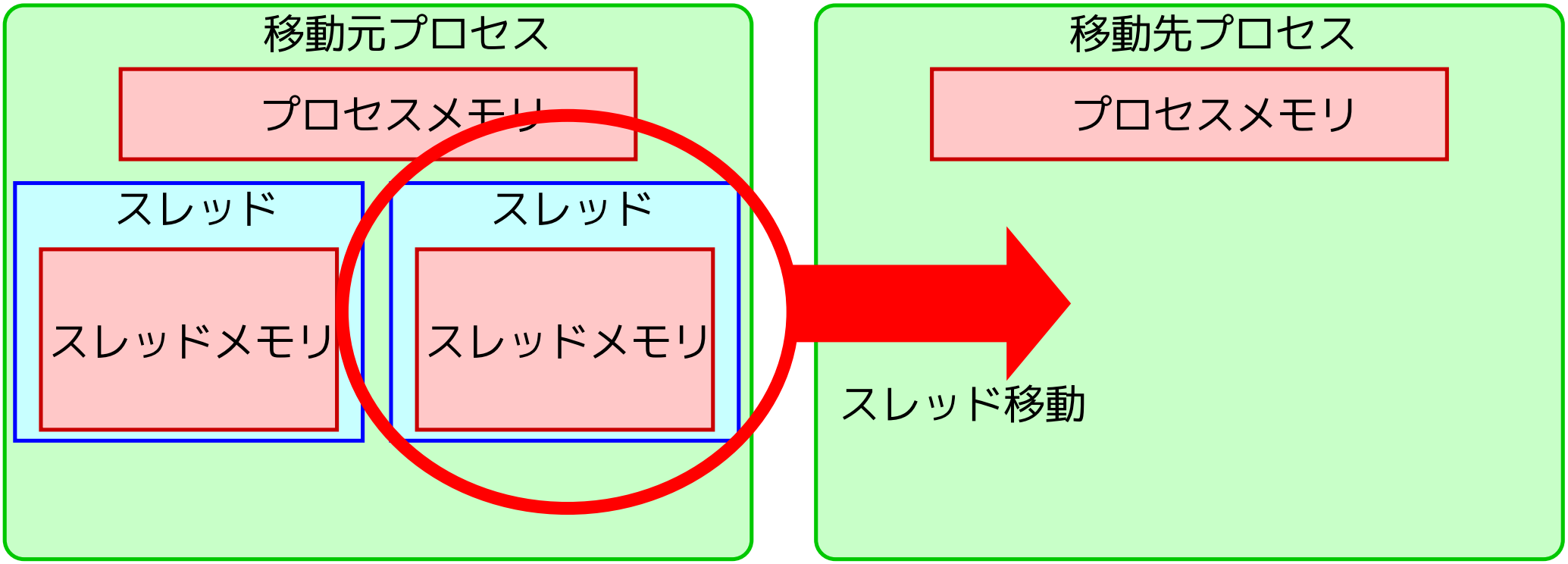


- **スレッドメモリだけを移動**して，移動元プロセスと移動先プロセスとで同一のアドレスに配置すれば OK
  - → 安全な継続実行を保証
- グローバルメモリの移動は不要

❖ スレッド移動の実装

DMI に限らない「一般的な」問題設定

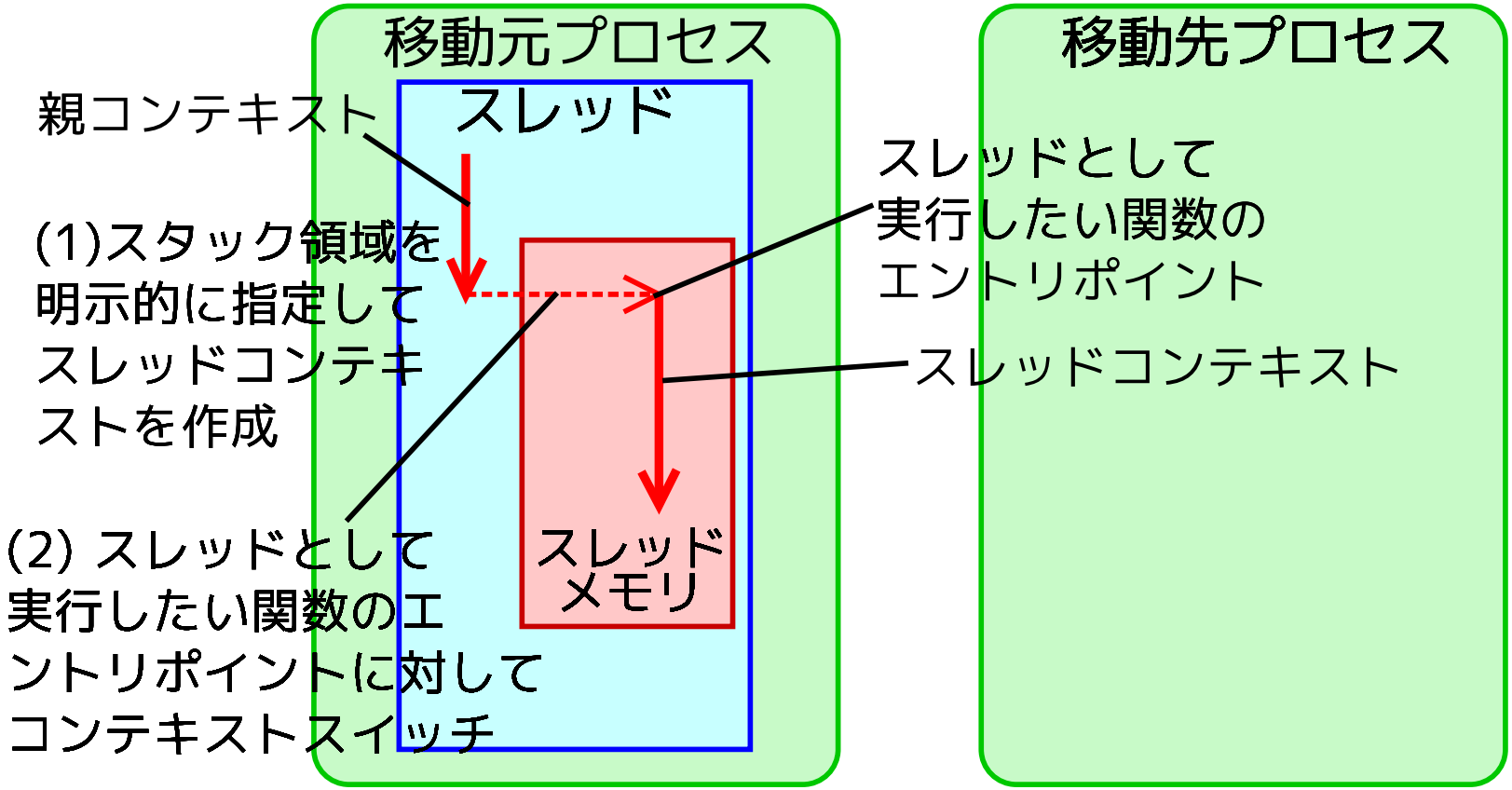
- 「一般的な」問題設定：
  - → プロセスメモリとスレッドメモリがある
  - → スレッド移動時にスレッドメモリだけ移動させる
- 実装上の課題：
  - (1) 課題 1：スレッドのチェックポイント・リスタートの実装
  - (2) 課題 2：random-address の実装



課題１：チェックポイント・リスタートの実装 (1)

PC ➡

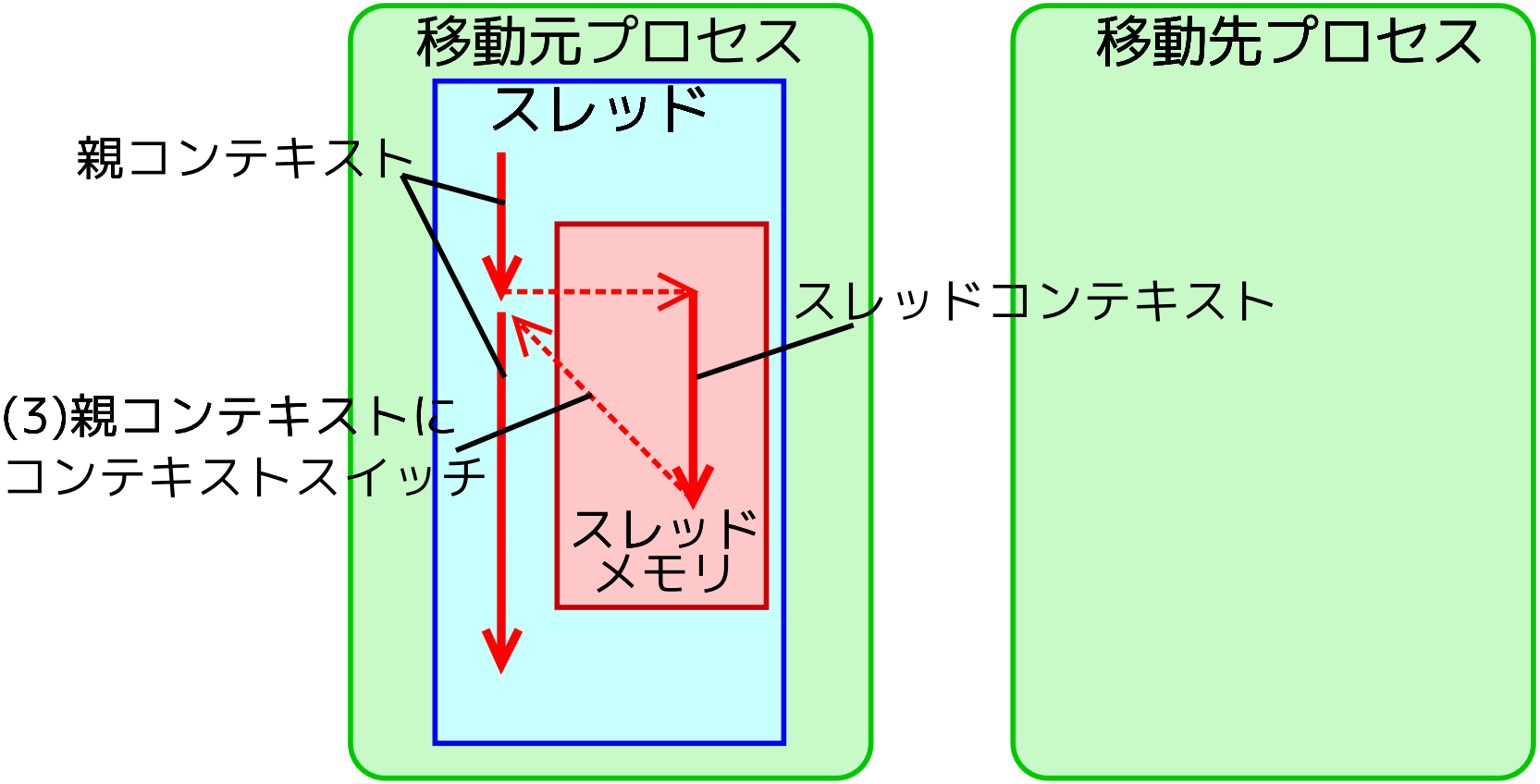
```
各スレッドの関数() {
    ...;
    yield(); // スレッド移動
    ...;
}
```



課題１：チェックポイント・リスタートの実装 (2)

PC

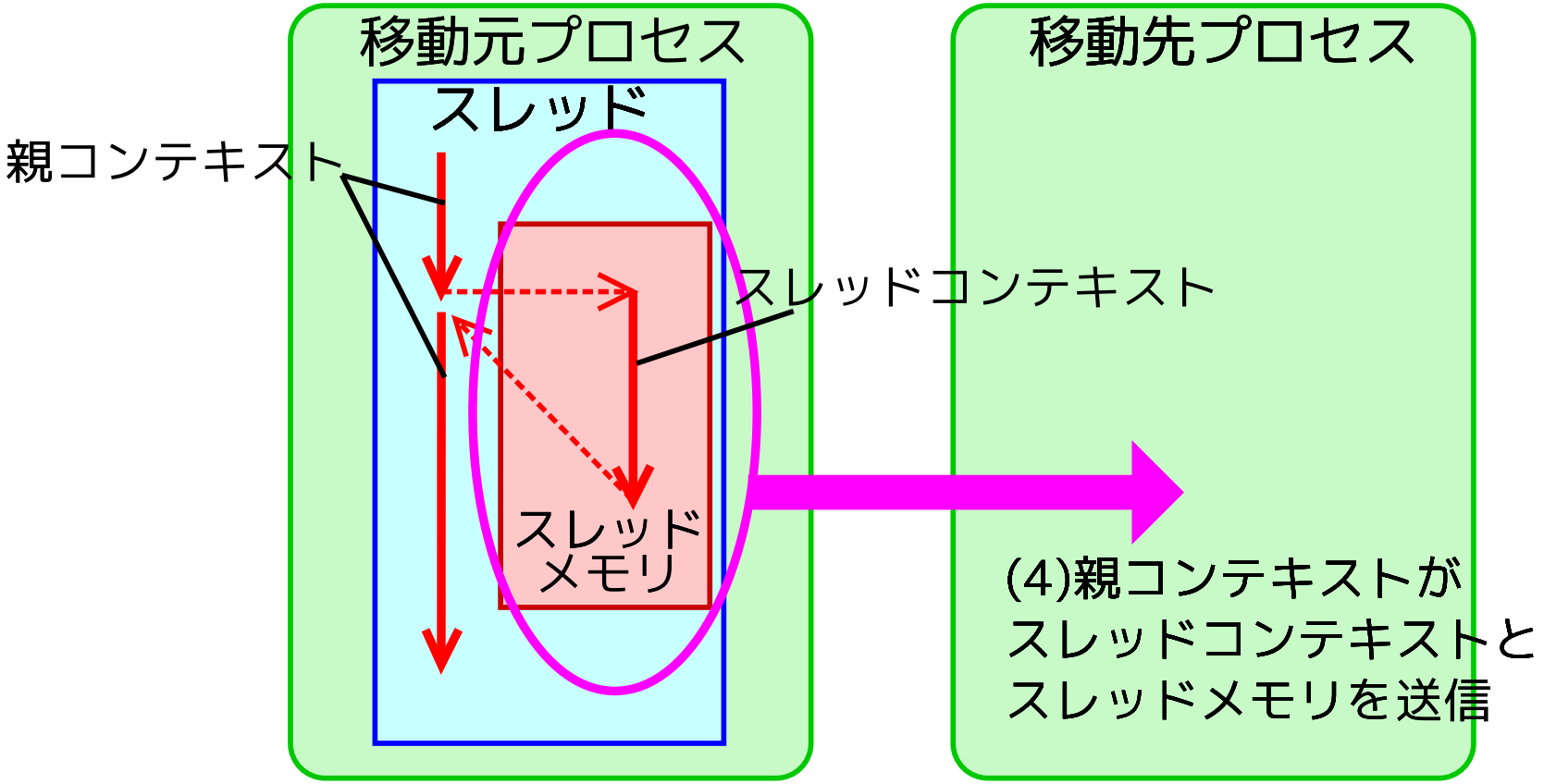
```
各スレッドの関数() {
    ...;
    yield(); // スレッド移動
    ...;
}
```



課題１：チェックポイント・リスタートの実装 (3)

```
各スレッドの関数() {
    ...;
    yield(); // スレッド移動
    ...;
}
```

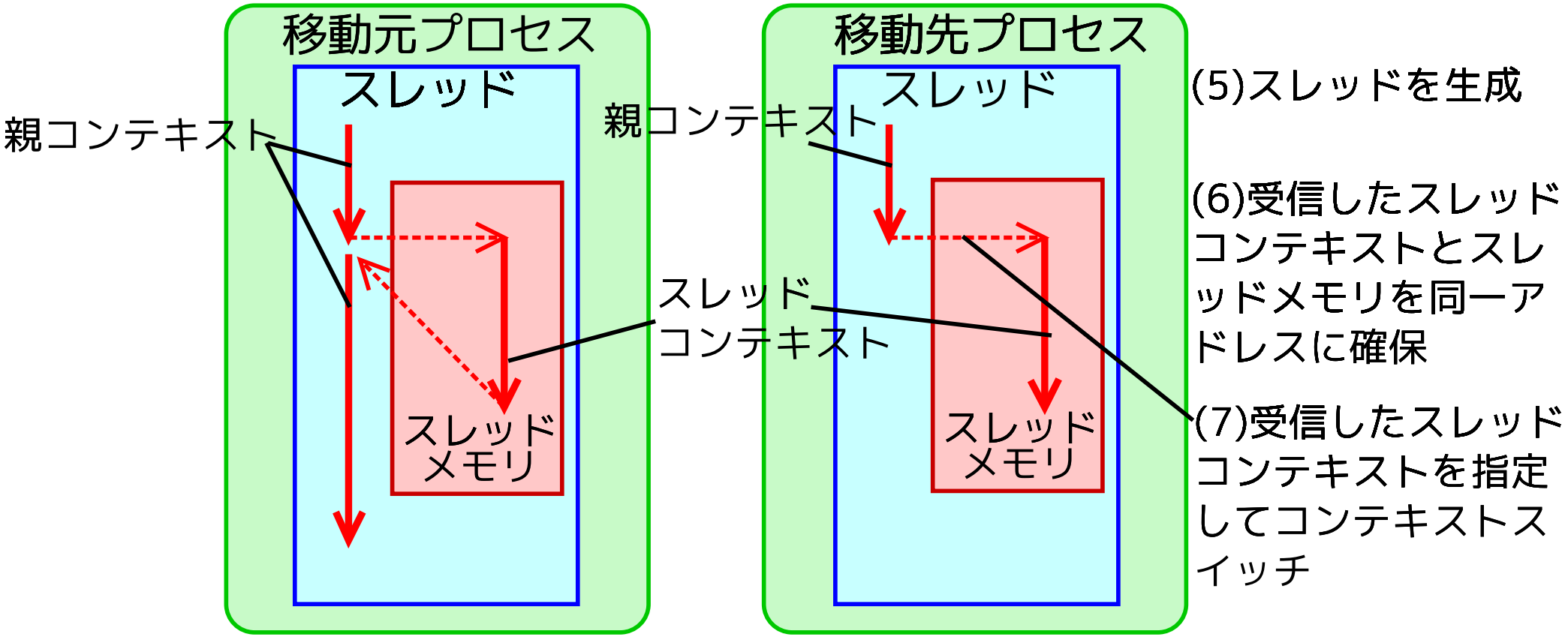
PC



課題１：チェックポイント・リスタートの実装 (4)

```
各スレッドの関数() {
    ...;
    yield(); // スレッド移動
    ...;
}
```

PC ➡



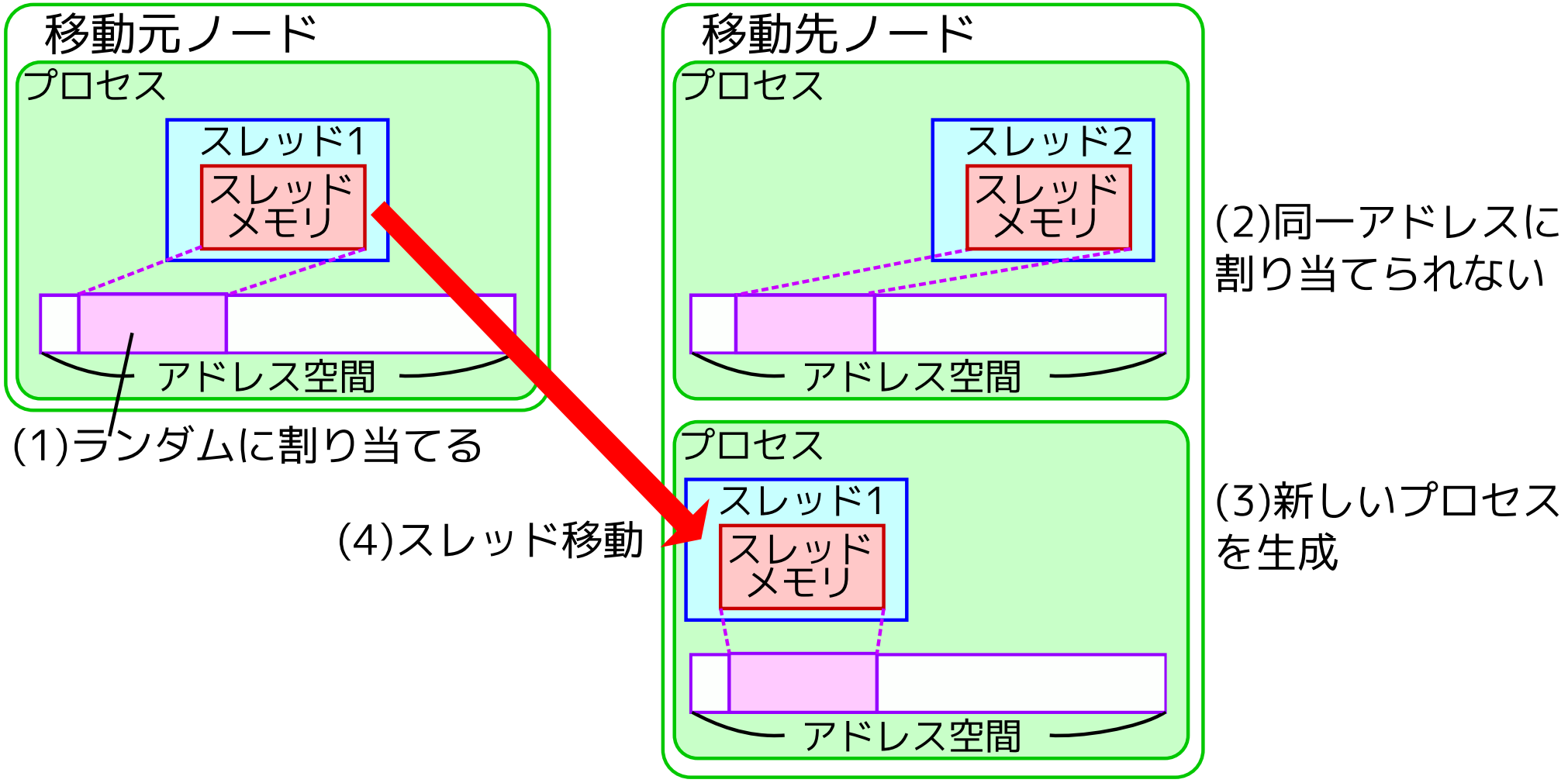
課題 2：random-address の実装

➤ **復習：**random-address

(1) **「運が悪くて」アドレスが衝突したら，**

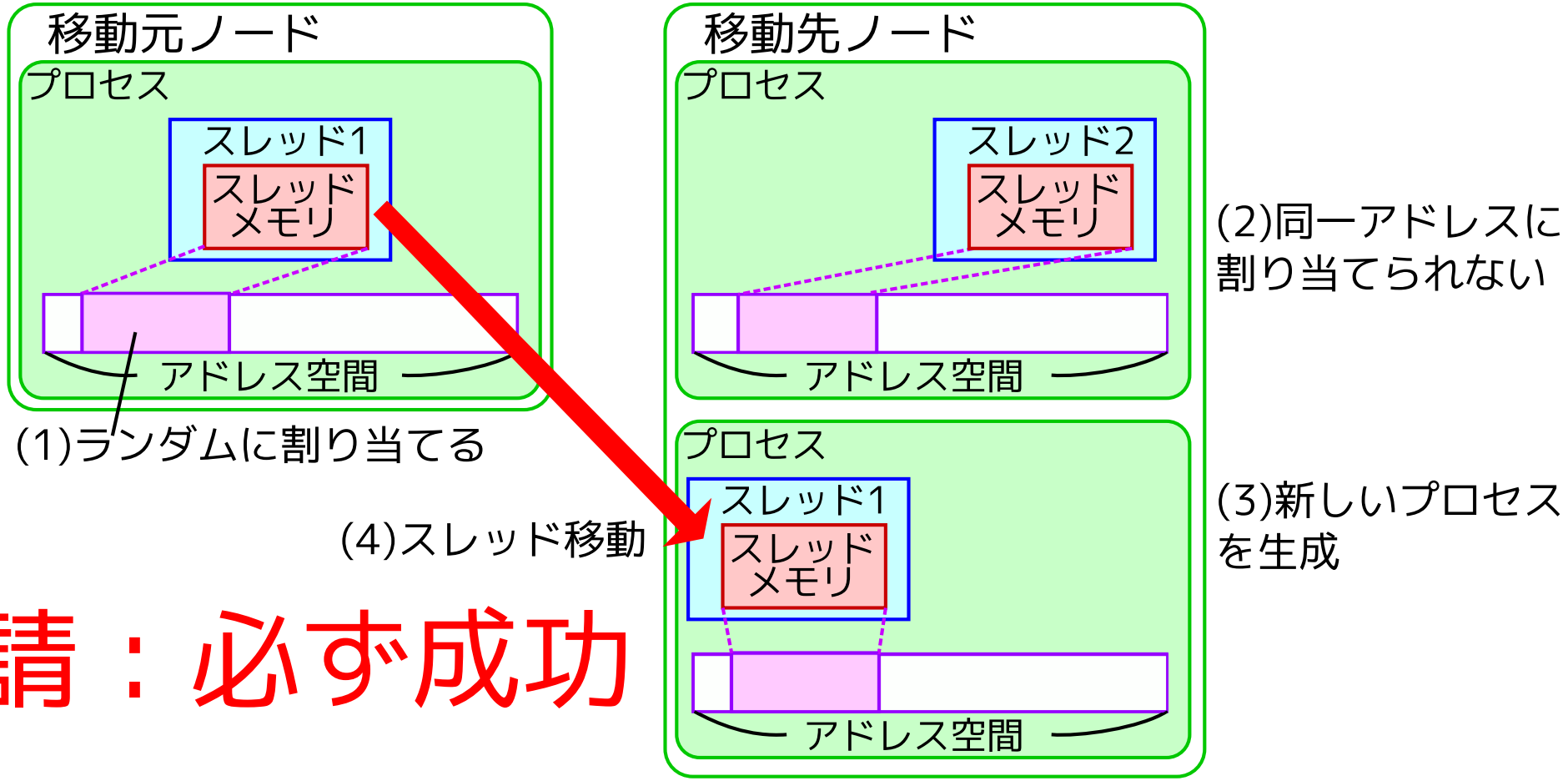
(2) **そのノード上に新しいプロセスを生成して，**

(3) **そのプロセスの中へスレッドを移動**



random-address における要請 (1)

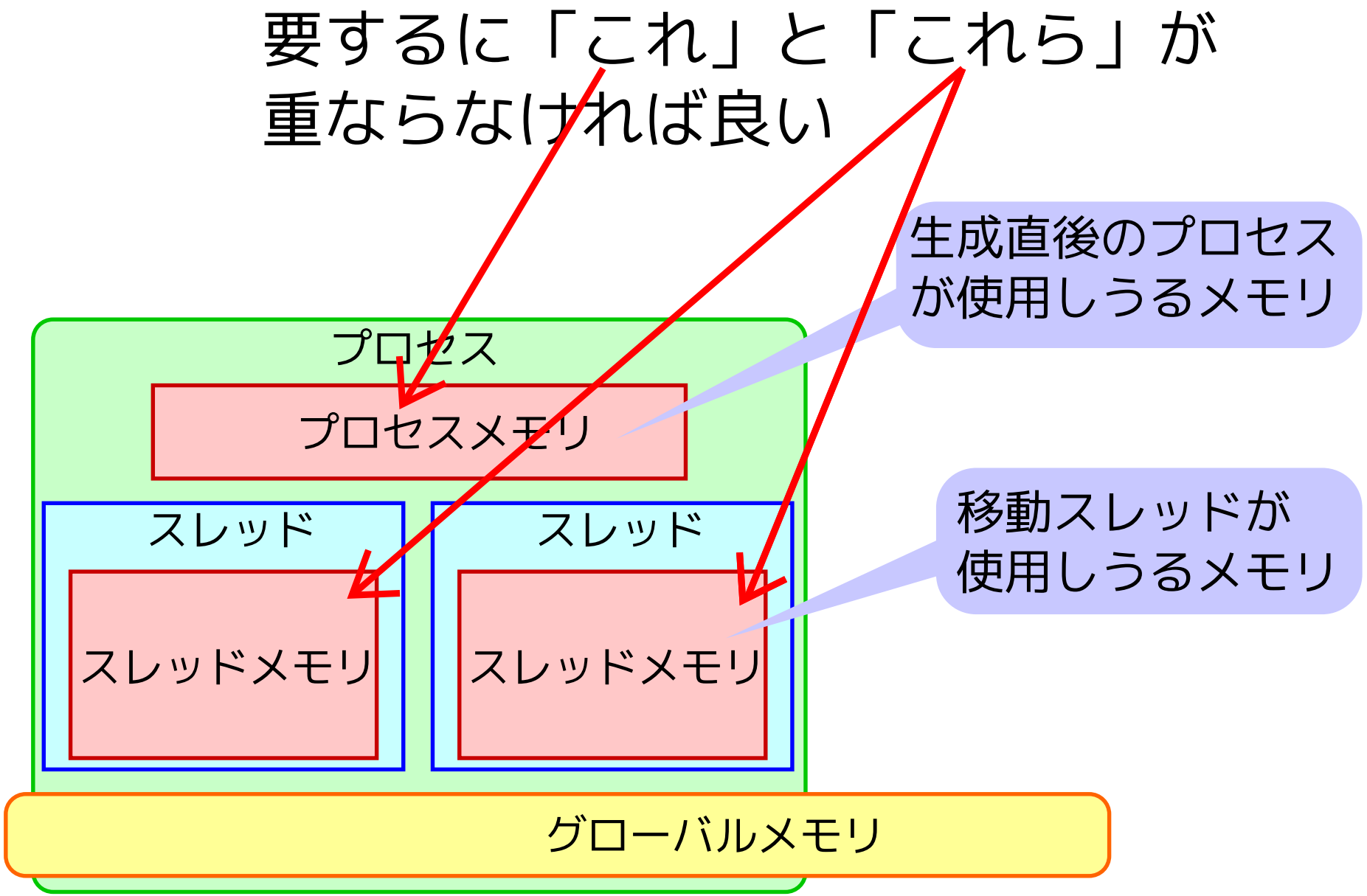
- 当然，生成直後の新しいプロセスへのスレッド移動は必ず成功しなければならない
  - → 生成直後のプロセスもすでに何らかのアドレス領域を使っている
  - → ということは，そのアドレス領域と，移動スレッドが使っているアドレス領域が重ならないことをどうにかして保証しないといけない



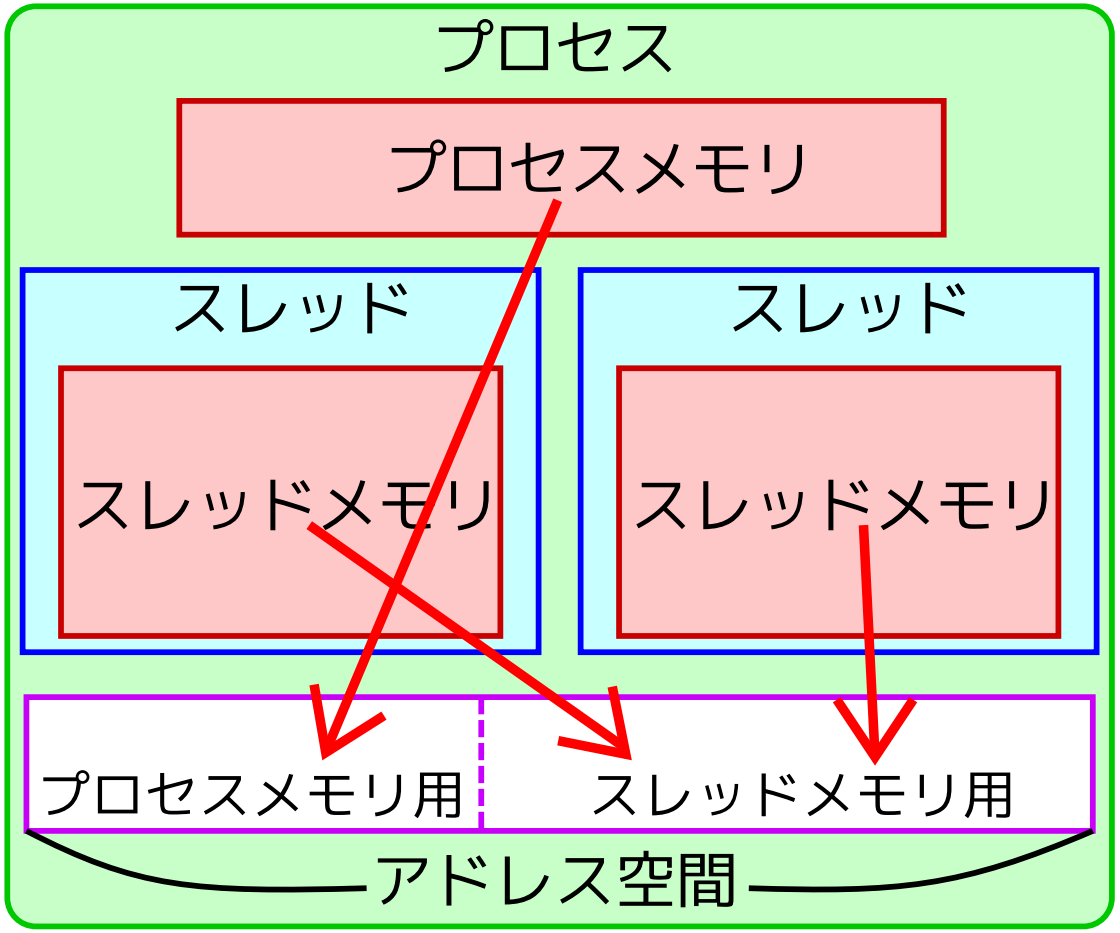
要請：必ず成功

random-address における要請 (2)

- 具体的には何を保証すればいいのか?

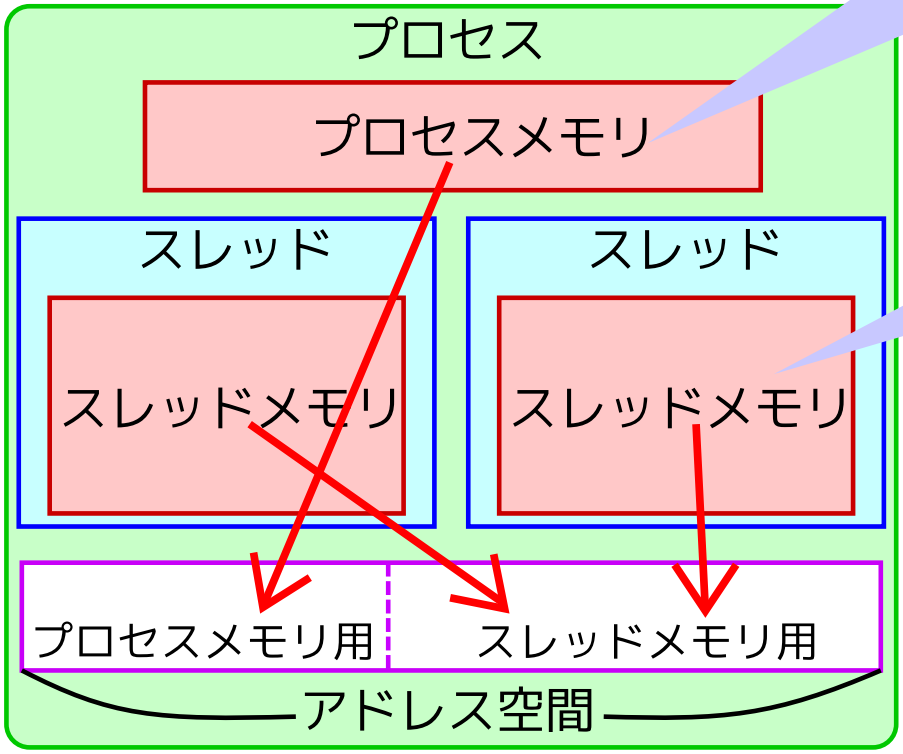


解決策



- **プロセスメモリとスレッドメモリを割り当てるアドレス空間を論理的に分離**
- **どうやって，プロセスメモリとスレッドメモリが割り当てるアドレスを明示的に操作するのか?**

割り当てるアドレスの明示的な操作



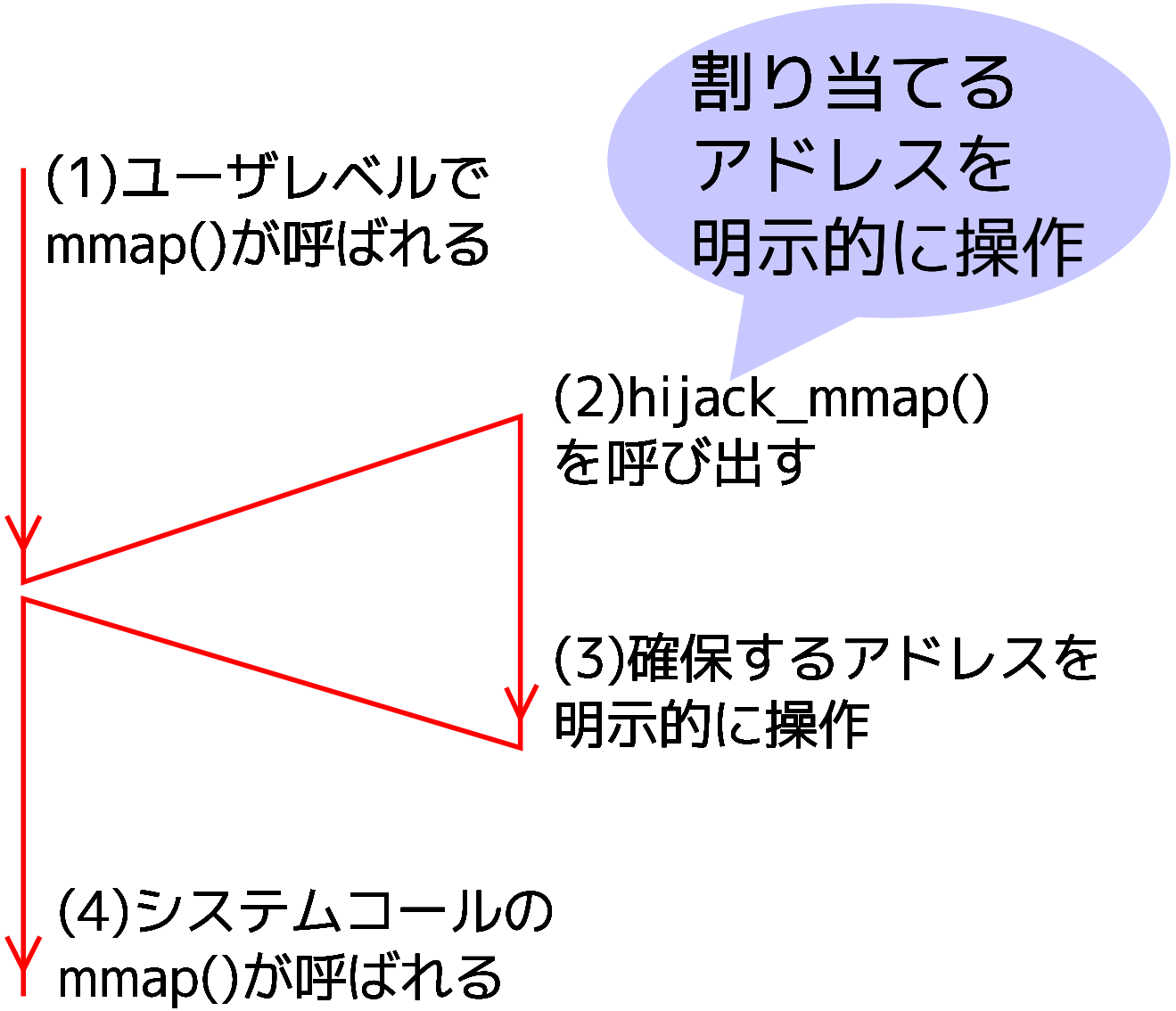
★グローバル変数
★(mmap/munmapで確保/解放される普通の)ヒープ

★レジスタ
★スタック
★(処理系のAPIによって確保/解放できる)スレッド固有のヒープ

- **スレッドメモリのアドレスを操作するのは容易**
- **プロセスメモリのアドレスを操作するのは困難**
  - → (malloc()/free() **などが)「処理系が知らないうちに」システムコールの** mmap()/munmap() **を呼び出してアドレスを割り当ててしまうから**

アプローチ：システムコールをハイジャックする

- どうやるか?

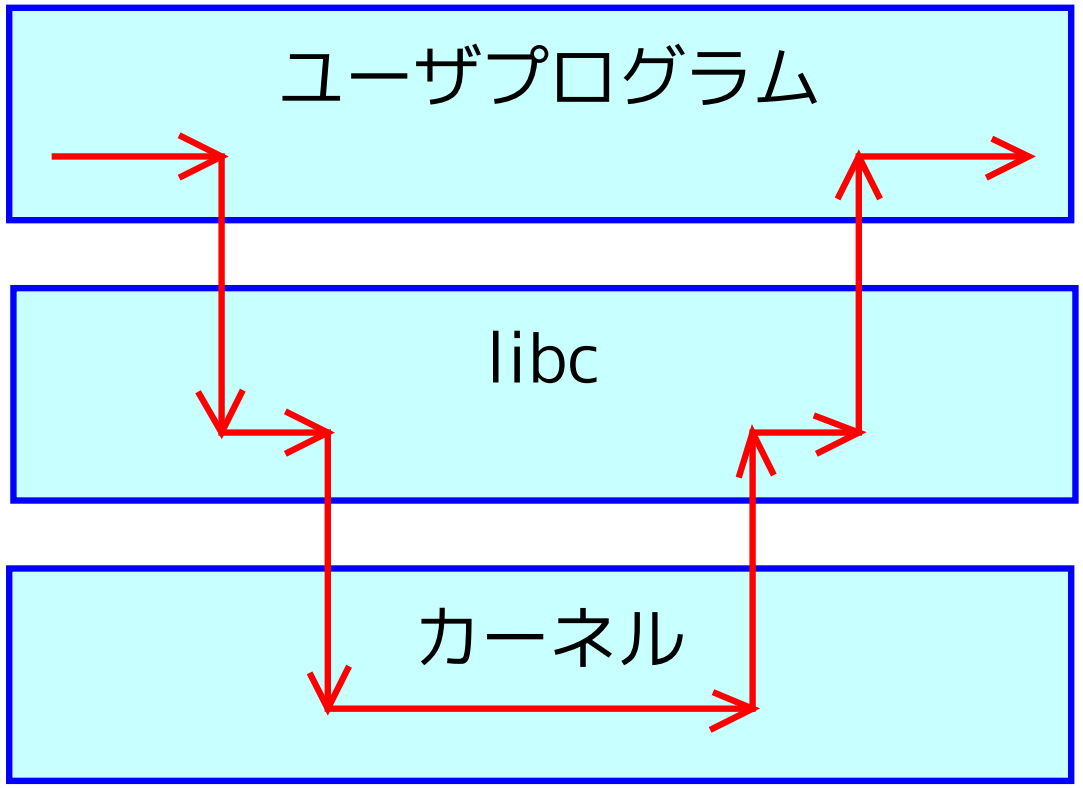


前提知識：システムコールのしくみ

(1)ユーザプログラムがlibcのmmapを呼び出す

(2)libcがシステムコールのmmapを呼び出す

(3)カーネルがシステムコールのmmapを実行

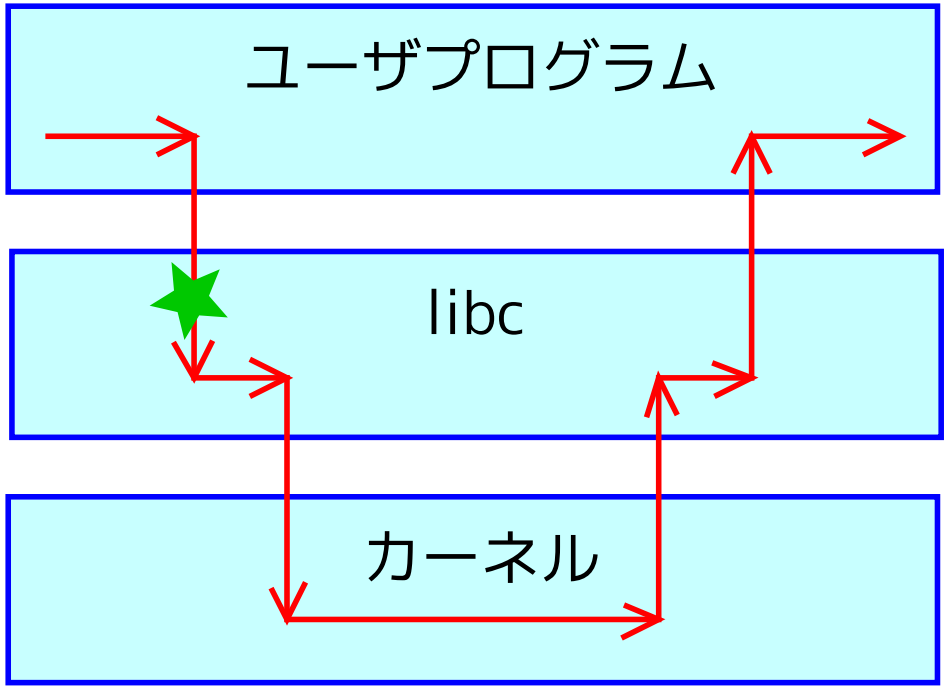


どのレイヤでハイジャックするか?(1)

(1)ユーザプログラムがlibcのmmapを呼び出す

(2)libcがシステムコールのmmapを呼び出す

(3)カーネルがシステムコールのmmapを実行



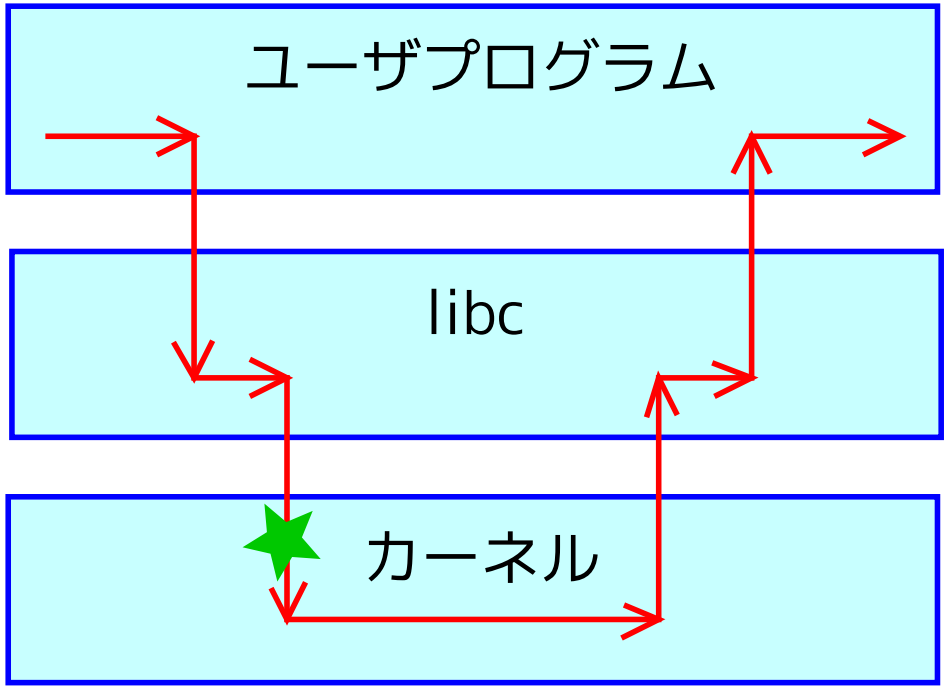
- **環境変数 LD_PRELOAD を使って共有ライブラリの検索順序を変更し，hijack_mmap() に処理を飛ばす**
- **問題点：静的リンクされている場合にはハイジャックできない**
  - → **通常，libc は静的リンクされているため，malloc() 内部で呼ばれる mmap() をハイジャックできない**

どのレイヤでハイジャックするか?(2)

(1)ユーザプログラムがlibcのmmapを呼び出す

(2)libcがシステムコールのmmapを呼び出す

(3)カーネルがシステムコールのmmapを実行



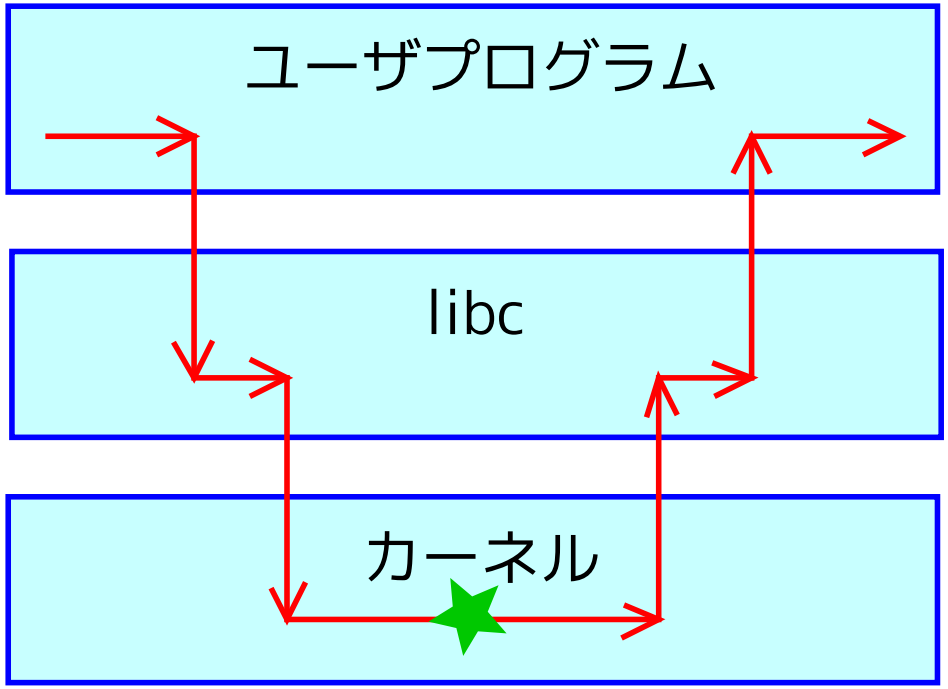
- **カーネルモジュールを使ってカーネルのシステムコールテーブルのエントリを書き換え，hijack_mmap() に処理を飛ばす**
- **問題点：カーネル 2.6 では，セキュリティ上の理由からシステムコールテーブルの先頭アドレスが extern されておらず，カーネルモジュールから利用できない**
  - → **カーネルを書き換えれば可能だが移植性に欠ける**

どのレイヤでハイジャックするか?(3)

(1)ユーザプログラムがlibcのmmapを呼び出す

(2)libcがシステムコールのmmapを呼び出す

(3)カーネルがシステムコールのmmapを実行



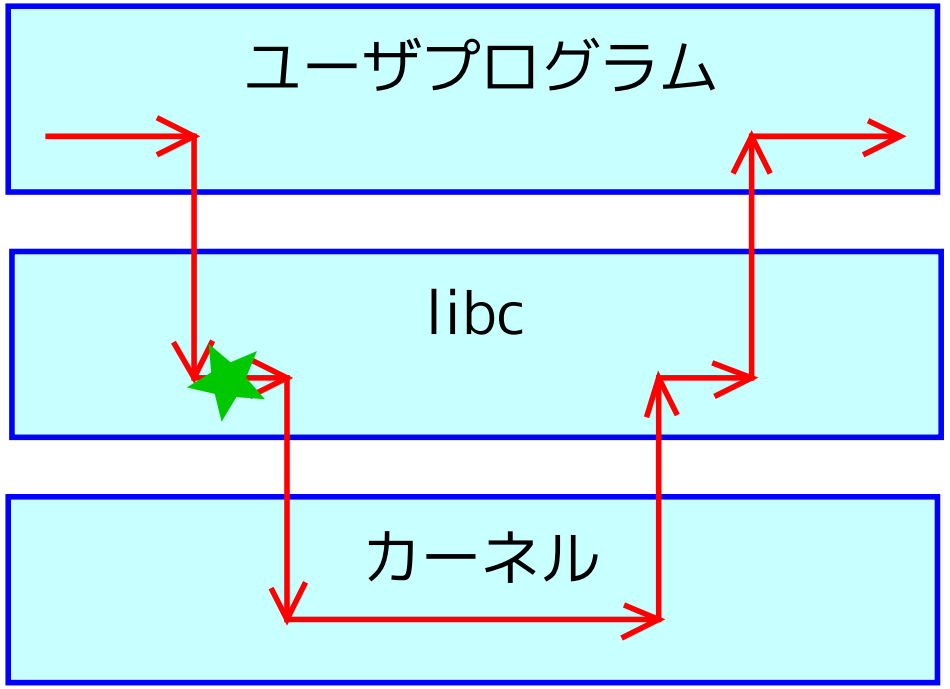
- ptrace **を使って，外部プロセスからシステムコール呼び出しを監視し，**mmap() **が呼ばれた瞬間に** hijack_mmap() **に処理を飛ばすよう，コードをインジェクションする**
- **問題点：**ptrace **はプロセス監視用であってスレッド監視の機能が不十分**
  - → pthread **の場合，プロセス内の「どの」**pthread **がシステムコールを呼び出したか判別できない**

どのレイヤでハイジャックするか?(4)

(1)ユーザプログラムがlibcのmmapを呼び出す

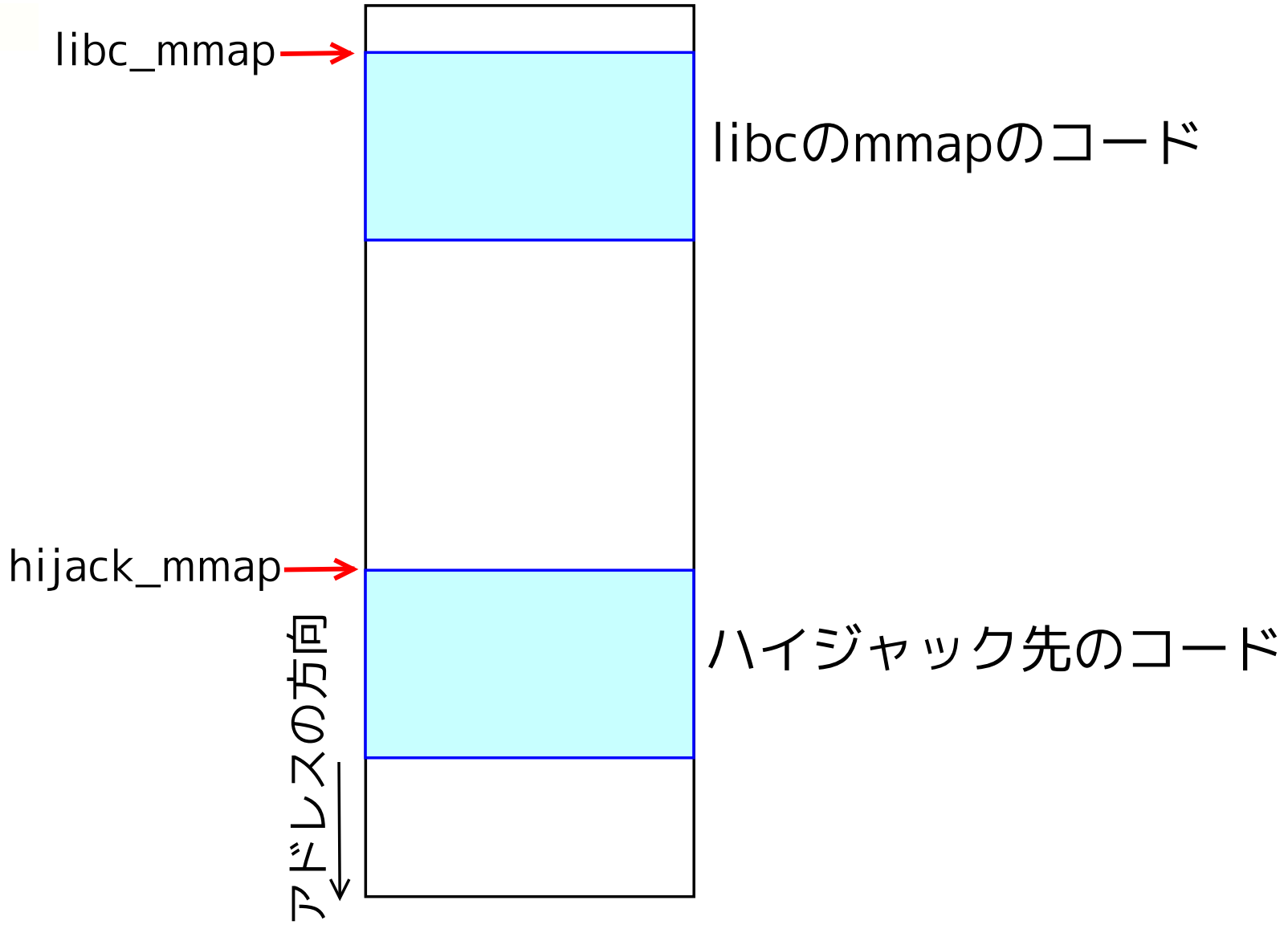
(2)libcがシステムコールのmmapを呼び出す

(3)カーネルがシステムコールのmmapを実行

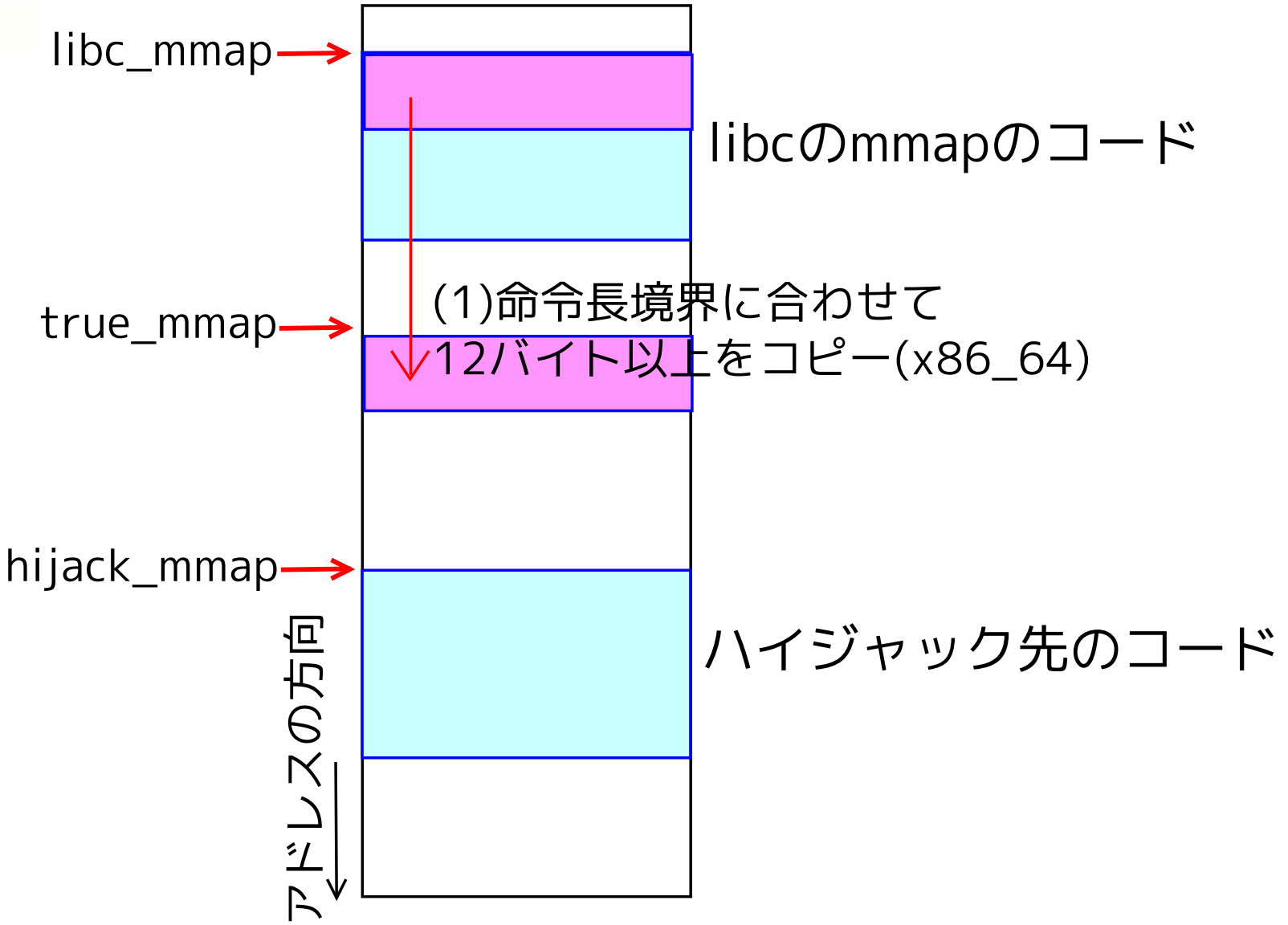


- **libc のコードを書き換えて hijack_mmap() に処理を飛ばす**
  - **(1) 静的に書き換える方法：改造 libc を作る**
    - **安全な方法だが...**
    - **問題点 1：変更箇所があまりに多い**
    - **問題点 2：各実行環境ごとに libc を準備する必要があり移植性が低い**
  - **(2) 動的に書き換える方法：採用!**

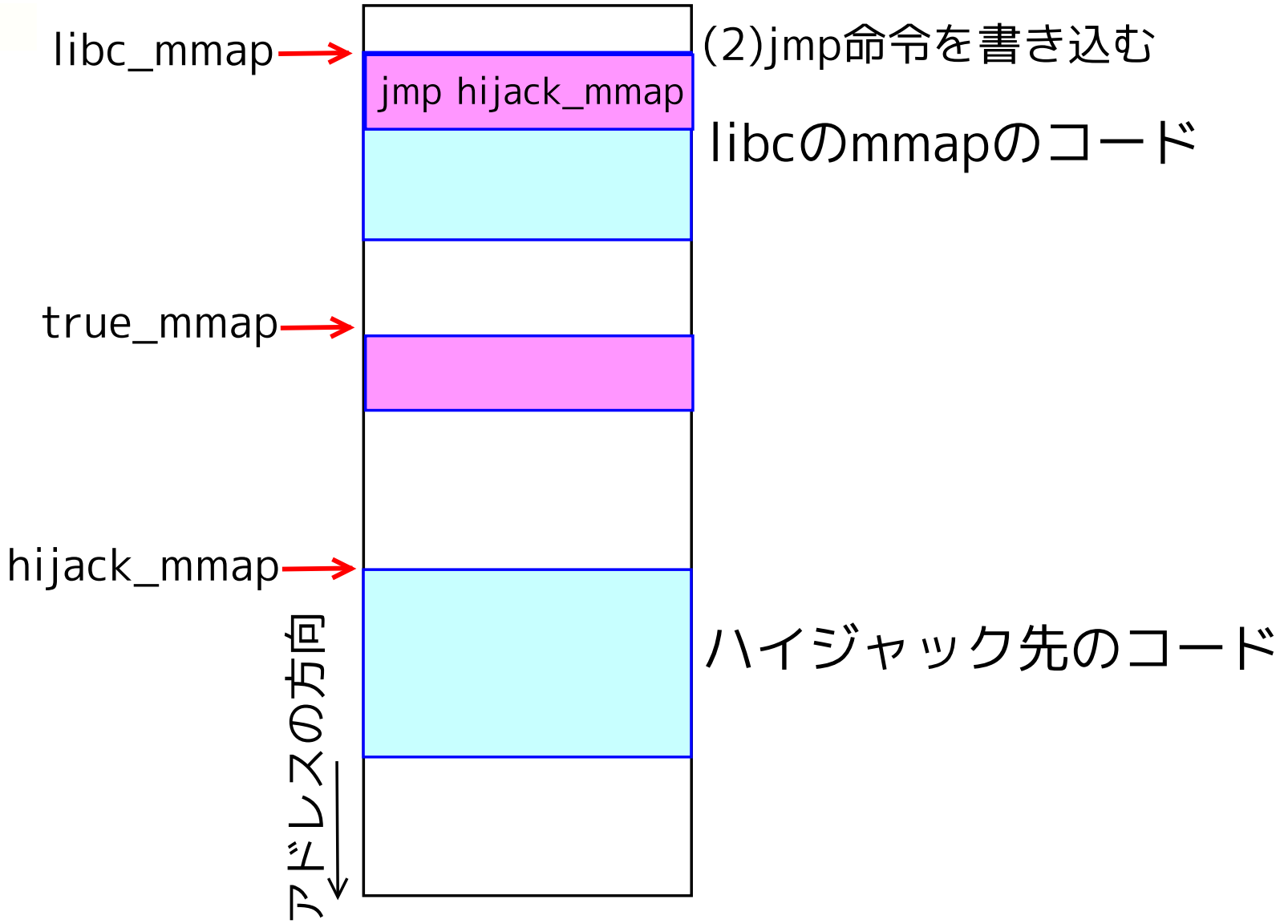
libc のコード領域を動的に書き換える (1)



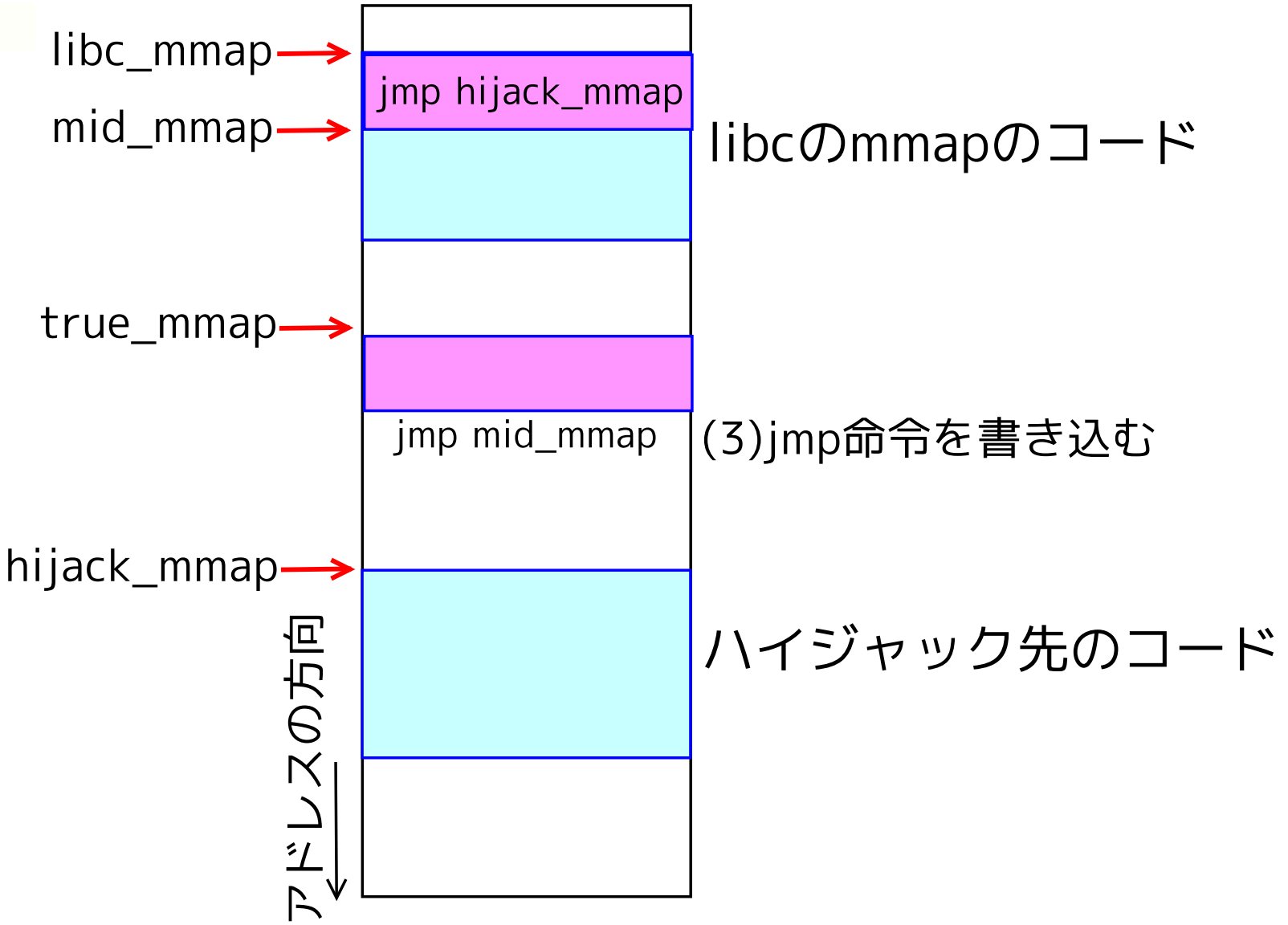
libc のコード領域を動的に書き換える (2)



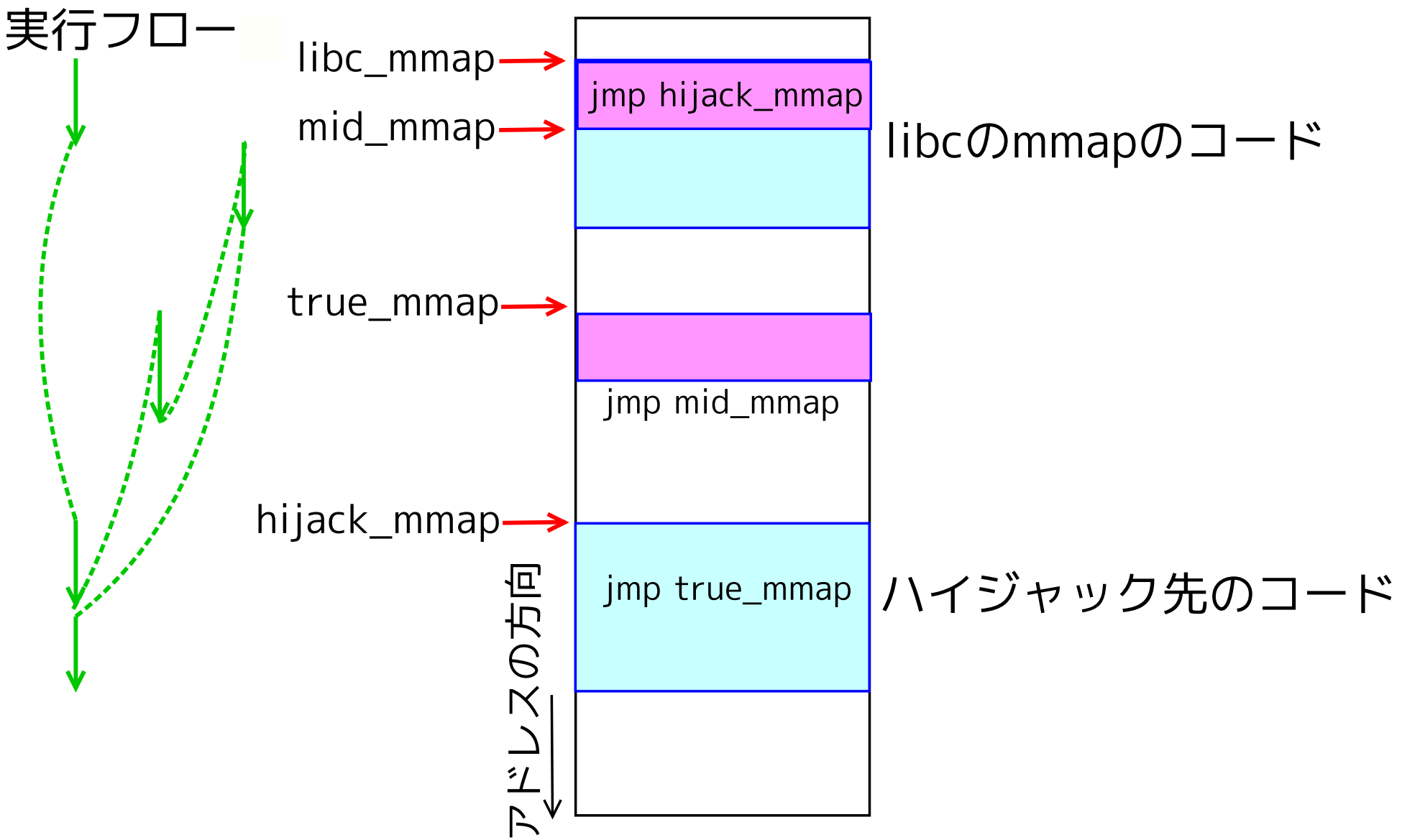
libc のコード領域を動的に書き換える (3)



libc のコード領域を動的に書き換える (4)

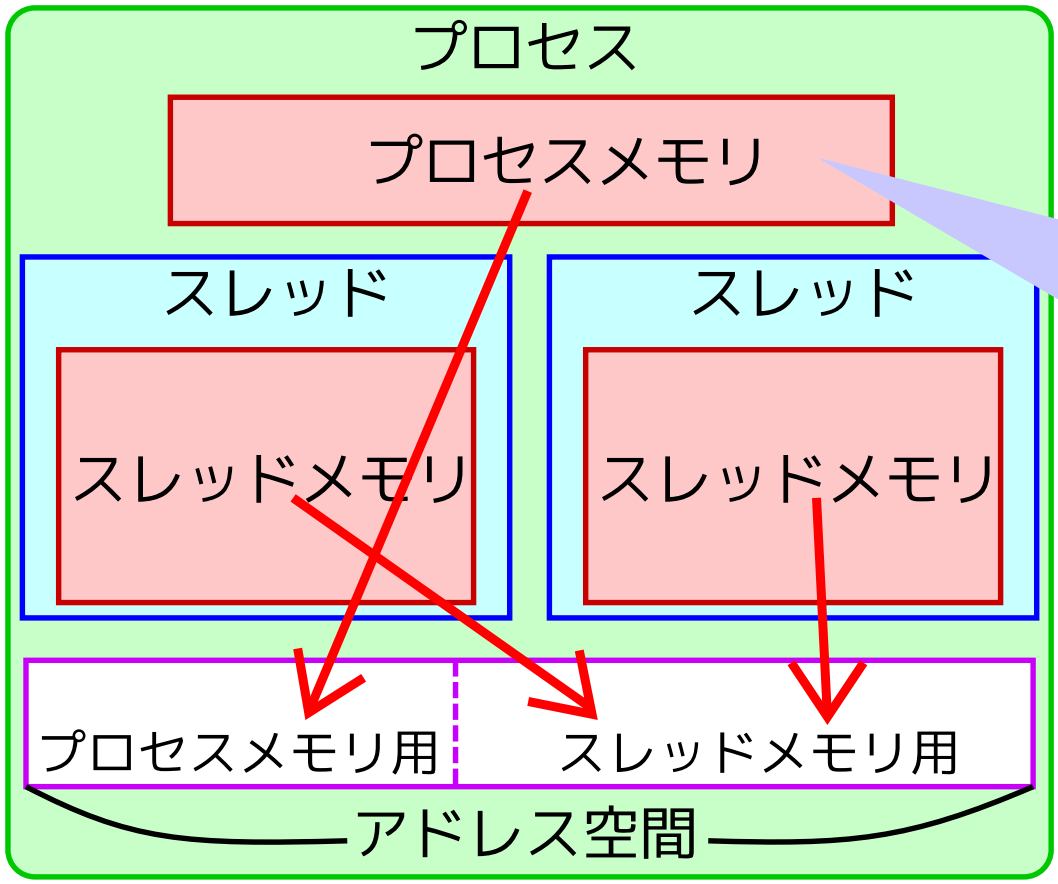


libc のコード領域を動的に書き換える (5)



ややこしくなったので，まとめ

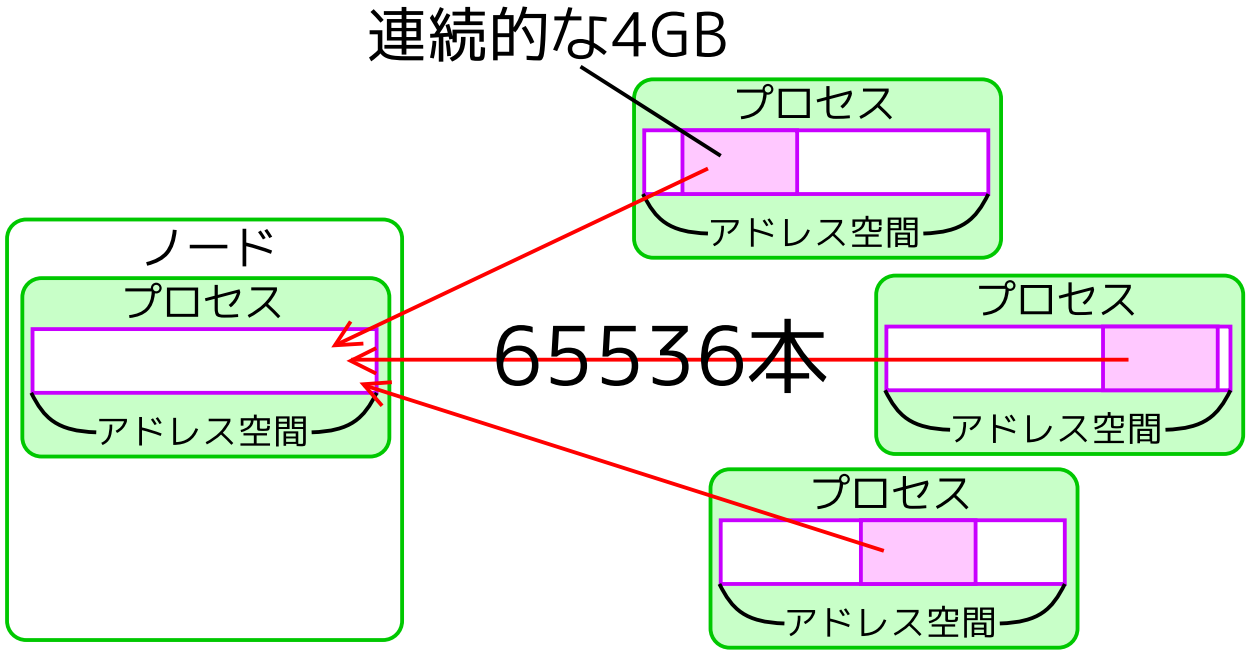
- そもそもの目的：**プロセスメモリとスレッドメモリを割り当てるアドレス空間を論理的に分離**
- プロセスメモリが割り当てるアドレスを明示的に操作するためには mmap()/munmap()(など) のハイジャックが必要
- **汎用的な**システムコールのハイジャック手法を提案



★グローバル変数
★(mmap/munmapで確保/解放される普通の)ヒープ

性能評価

random-address のシミュレーション (1)

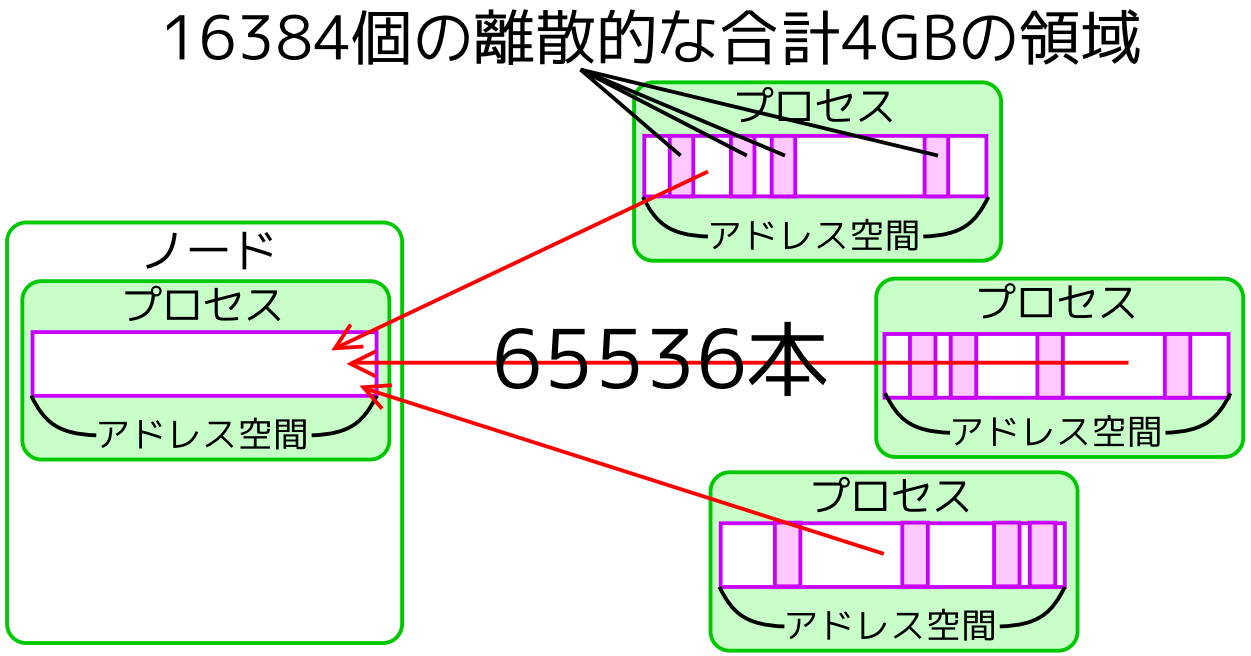


- **アドレス空間：** $128\mathrm{TB}(=2^{47}$ **バイト**$)$
- **プロセス数：** 65536 **本**
- **各プロセスの使用メモリ量：** $4\mathrm{GB}$
- **各プロセスは連続的にアドレス領域を使用**
- **全プロセス内の全スレッドを** 1 **ノードへとスレッド移動して集約させた結果，そのノードに何本のプロセスが生成されたか?**

  → 結果：平均 4 **本**

  → 「この規模ではほとんどアドレス衝突は起きない」

random-address のシミュレーション (2)



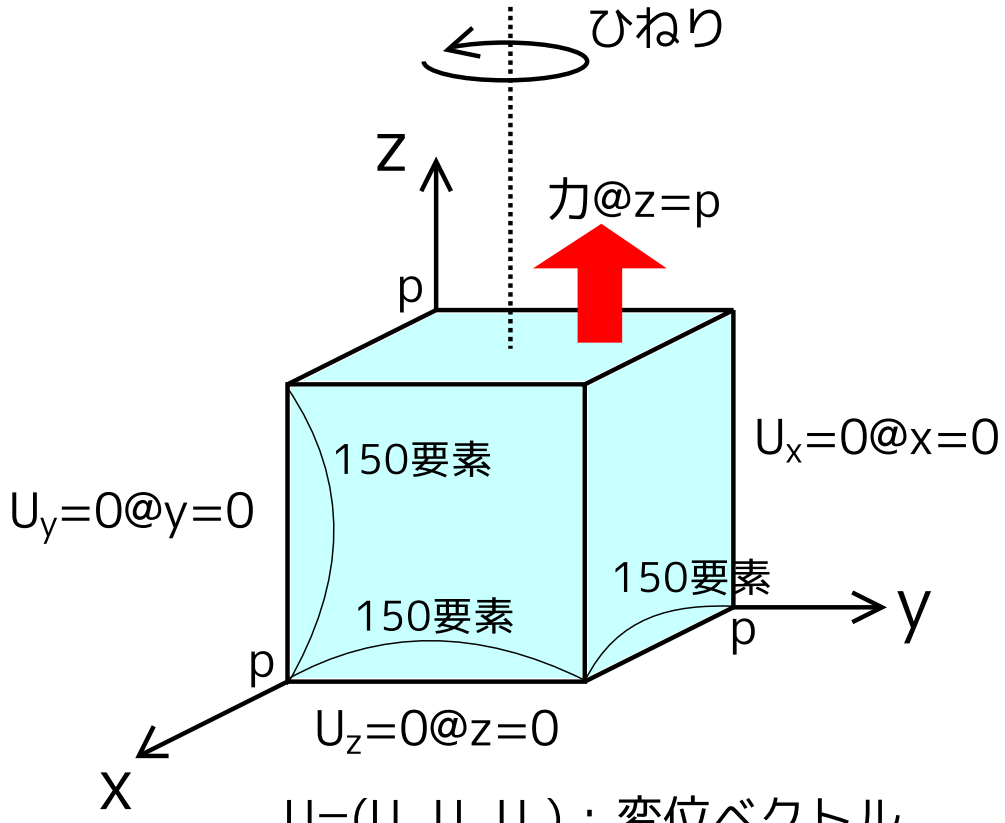
- **アドレス空間：**$128\mathrm{TB}(=2^{47}$ **バイト**$)$
- **プロセス数：**65536 **本**
- **各プロセスの使用メモリ量：**$4\mathrm{GB}$
- **各プロセスは** 16384 **個の離散的なアドレス領域を使用**
- **全プロセス内の全スレッドを** 1 **ノードへとスレッド移動して集約させた結果，そのノードに何本のプロセスが生成されたか?**

  → 結果：平均 5828 **本**

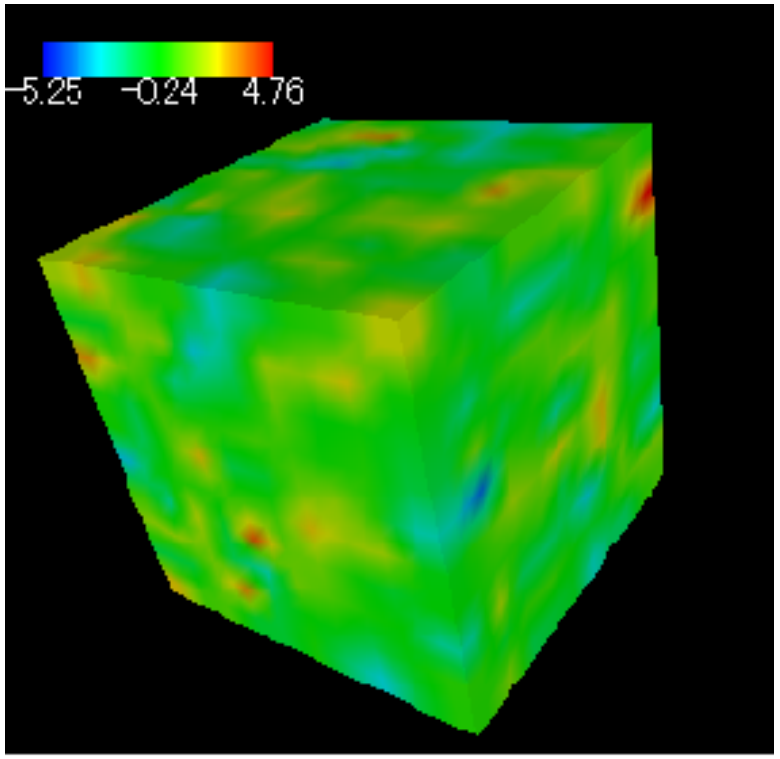
  →「連続的にアドレスを使うほどアドレス衝突が起きにくい」

有限要素法：実世界の並列科学技術計算

- 有限要素法による応力解析：
  - → 第 2 回並列プログラミングコンテストの題材
  - → 疎行列係数の連立一次方程式の解を反復法で求める
  - → **実世界の工学**に基づく非常に収束させづらい問題
- 実験環境：8 コア ×16 ノード，10GbitEthernet



U=(U_x,U_y,U_z)：変位ベクトル



[Nakajima,2009]

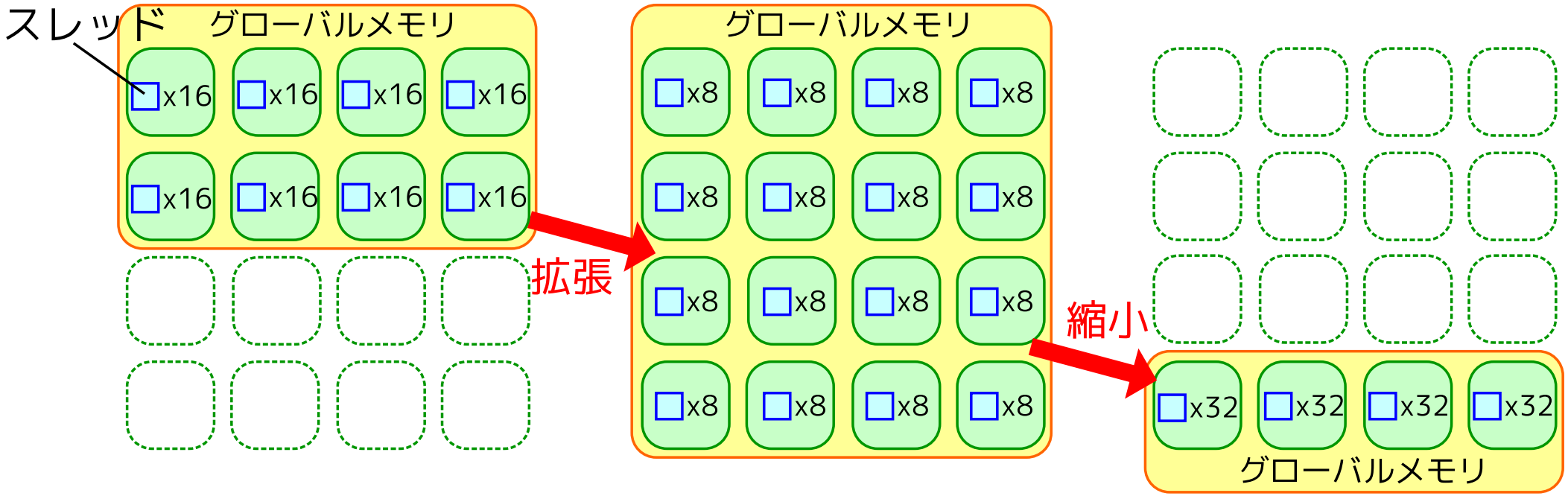
有限要素法：プログラミングの簡単さ

- (計算規模の拡張・縮小非対応の DMI プログラムに対する) わずかな変更点：
  - → 各イテレーションの先頭に DMI_yield() を追加
  - → 一部の malloc()/free() を DMI の API に変更

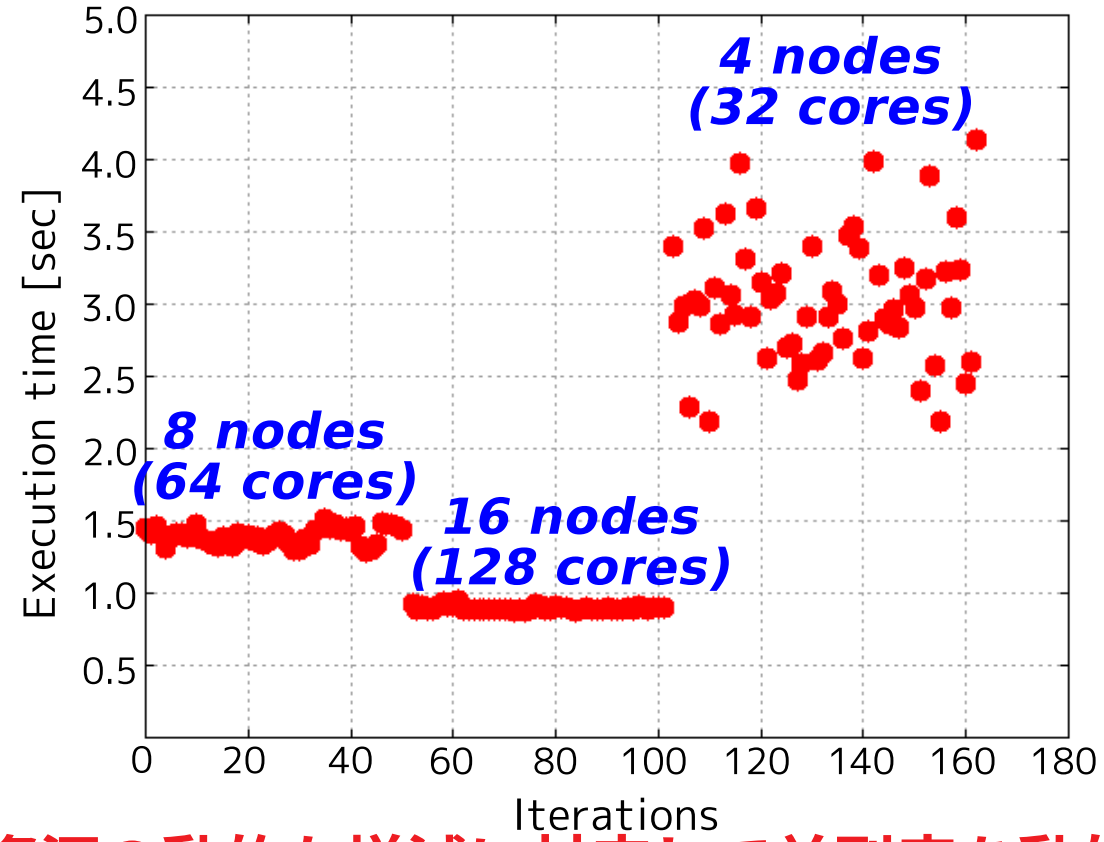
```
for (iter = 0; 収束するまで; iter++) {
    DMI_yield();
    ......;
}
```

有限要素法：実験シナリオ

- 128 本のスレッドを生成
  - → 各スレッドはスレッドヒープ領域を 500MB の使用 (全体では 64GB)
  - → グローバルメモリ領域を 335MB 使用
- 利用可能なノード数を増減：
  - → ノード 1 からノード 8 で実行
  - → ノード 9 からノード 16 を参加させる
  - → ノード 1 からノード 12 を脱退させる



有限要素法：結果

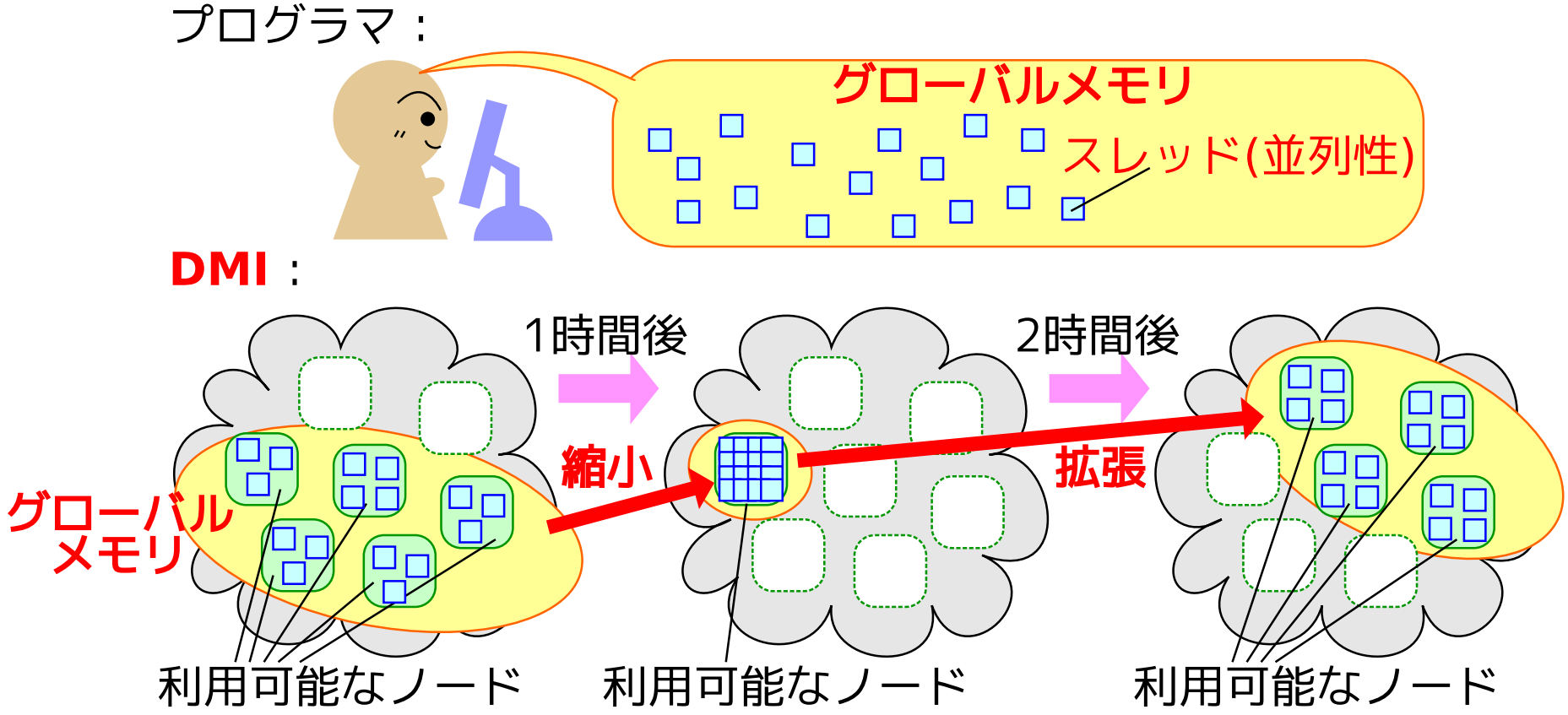


- 利用可能な計算資源の動的な増減に対応して並列度を動的に増減できた
- アドレス衝突は発生せず
- 移動時間：
  - → 8 ノードの参加に伴う 120 スレッド (58GB) の移動：17.3 秒
  - → 12 ノードの脱退に伴う 120 スレッド (58GB) の移動：30.9 秒

結論

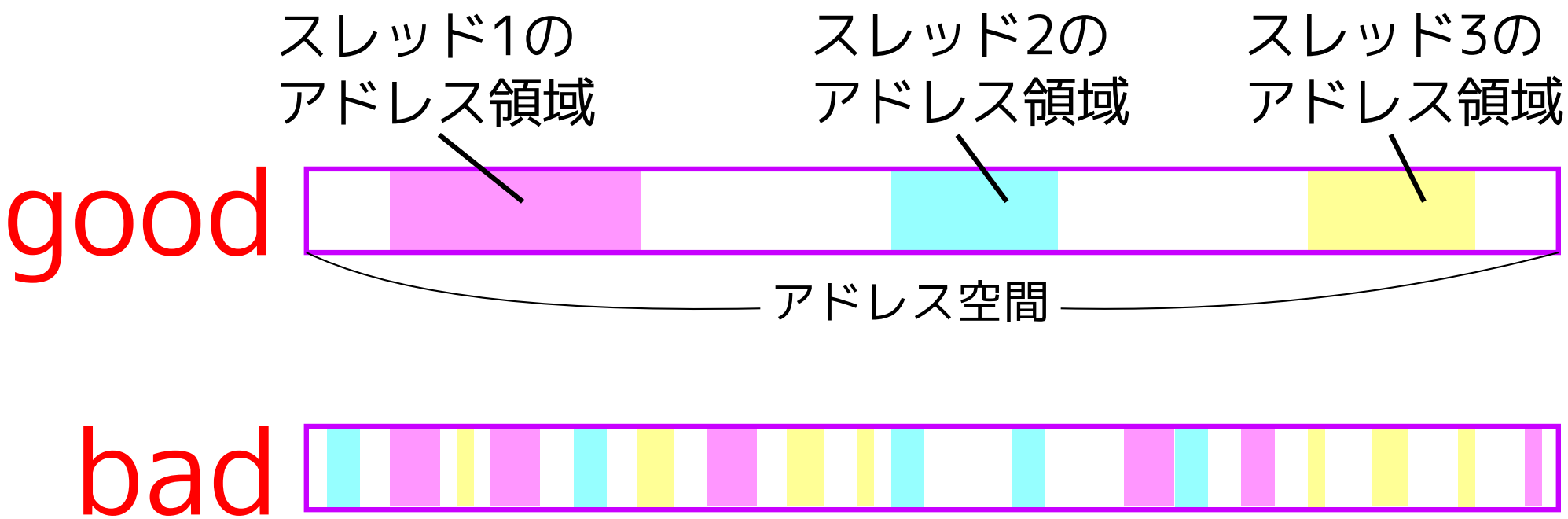
まとめ (1)

- DMI：並列計算の規模を「透過的に」拡張・縮小可能な PGAS 処理系
  - (1) プログラマは，「単に」十分な数のスレッドを生成するだけで良い
  - (2) あとは DMI が，それら大量のスレッドを利用可能な計算資源に動的にマップして，「透過的に」計算規模を拡張・縮小してくれる
  - (3) スレッド間のデータ共有レイヤーとして，高性能なグローバルメモリが提供される



まとめ (2)

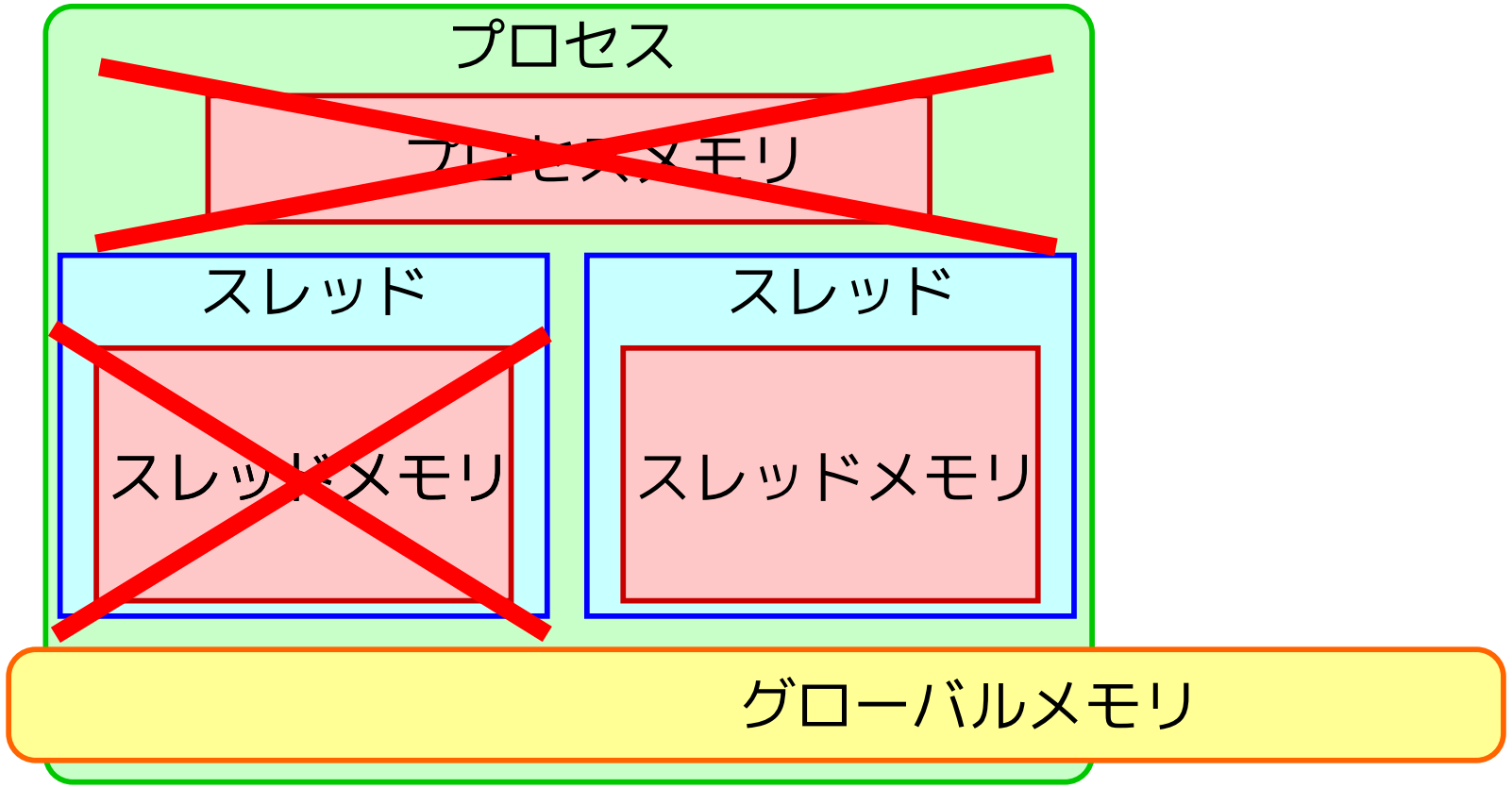
- random-address：アドレス空間の大きさに制限されないスレッド移動のためのアドレス管理手法
  - → 「全スレッドができるだけアドレスを連続的に使う」のが最適
  - → 汎用的なシステムコールのハイジャック手法



今後の課題 (1)

- ノード間スレッドスケジューリングの最適化
  - → 現実装ではノード間のスレッド数の均等化しか考えていない
  - → 以下を総合的に考慮：
    - (1) ノード間のスレッドの負荷バランス
    - (2) スレッド移動に要する時間
    - (3) ノード内のプロセス数が増えることに起因するオーバーヘッド
    - (4) 各スレッド間でのデータ共有度合い

今後の課題 (2)



- DMI_yield() **呼び出し時に「実行状態がプロセスメモリに存在してはダメ (グローバル変数を使ってはダメ)」という不便なプログラミング制約の撤廃**
  - → **真に「透過的な」スレッド移動**
  - → **「部分的にアドレス空間が独立したスレッド」をカーネルレベルで実装 (詳細略)**

Thank you!