【講座3】
MP T-Kernel入門

（株）日立超LSIシステムズ
豊山　祐一

マルチコアプロセッサの必要性

- 組込み機器の高機能化　⇒　プロセッサの高速化
- 高速化の限界　・・・　消費電力、発熱、etc.

- マルチプロセッサによる処理能力の向上
- 組込み機器向けのマルチコアの登場により注目が集まる

- マルチコア：一つのプロセッサ（チップ）の中に複数のCPUコア

マルチコアプロセッサの方式

- ヘテロジニアス型とホモジニアス型
  - ヘテロジニアス型：
    異なった種類のコアからなるマルチコア
  - ホモジニアス型：
    同一の種類のコアからなるマルチコア
- AMP型とSMP型
  - AMP(Asymmetric Multiple Processor) 非対称型：
    各コアの役割が異なる　⇒ プログラムの実行コアが決定
  - SMP(Symmetric Multiple Processor)　対象型：
    各コアの役割に違いが無い　⇒ プログラムはどのコアでも実行可
    ホモジニアス型、メモリ共有が前提

MP T-Kernelとは

MP T-Kernelとは

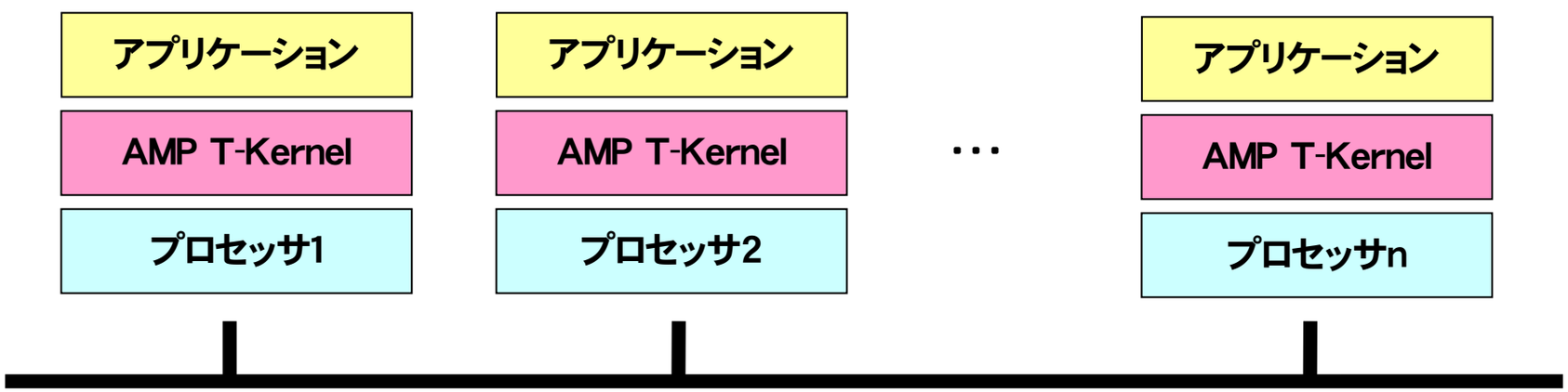
- マルチコアやマルチプロセッサに対応したリアルタイムOS
  - リアルタイム性が要求される組込みシステムが主たる対象
- 標準のT-Kernelとの互換性を重視
  - 既存のソフトウエア資産を有効活用できる
- プロセッサの構成やシステムの要求に応じてOSを選択可能
  - AMP T-Kernel (非対称型マルチプロセッサに対応)
  - SMP T-Kernel (対称型マルチプロセッサに対応)

AMP T-Kernelとは (1/4)

- 非対称型マルチプロセッサ（AMP: Asymmetric Multiple Processor）に対応したMP T-Kernel
- 個々のプロセッサの用途が限定されており、プロセッサごとに固有のプログラムが実行される

AMP T-Kernelとは (2/4)

- プロセッサごとにカーネルが割り当てられる
- カーネル間の接続はプロセッサ間通信機能によって実現されている
  - カーネル内に隠蔽されているのでアプリケーションは意識する必要はない

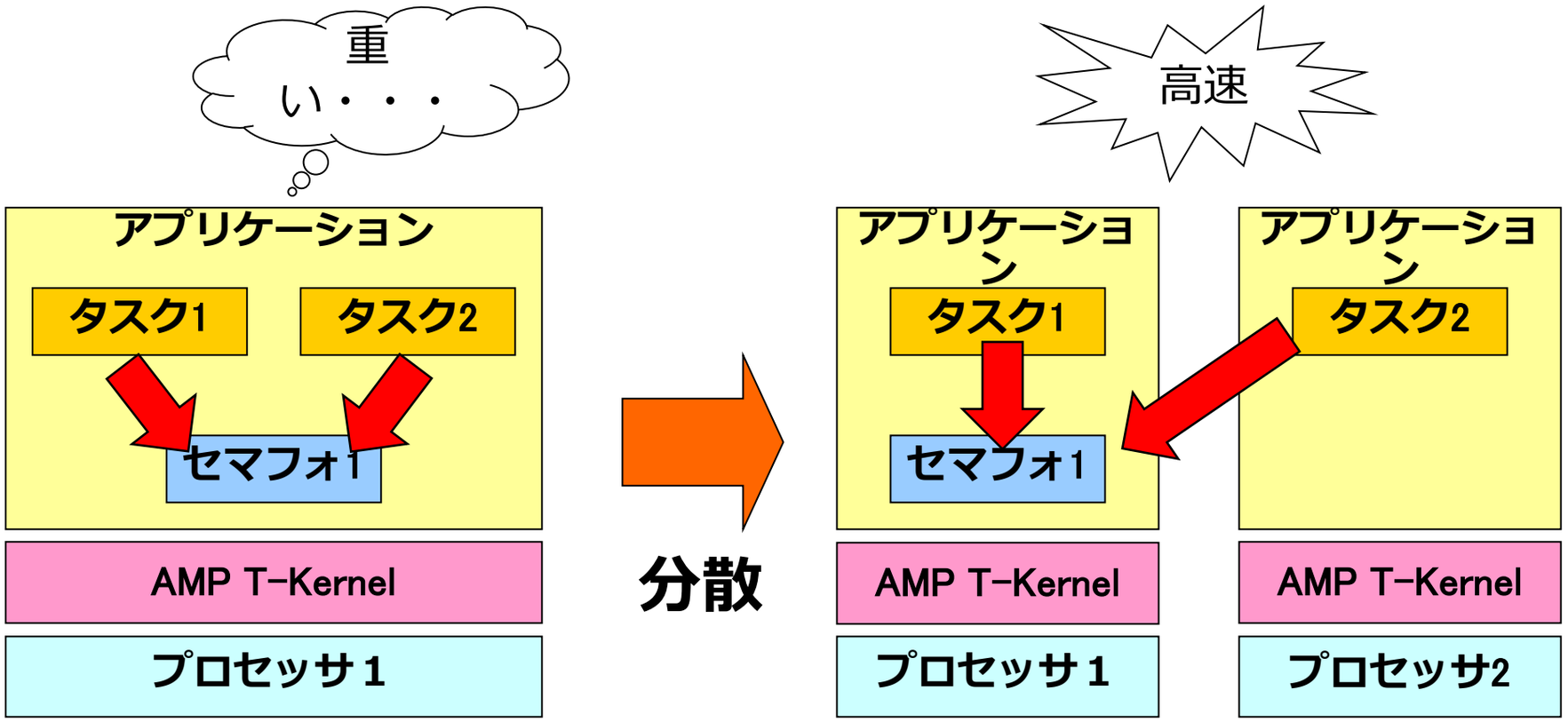


AMP T-Kernelとは (3/4)

- 資源の管理は各カーネルで独立している
  - カーネルオブジェクトやメモリなどの管理が独立している
- 個々のカーネルは標準のT-Kernelとほぼ同じ動作をする
  - 互換性が高いのでT-Kernelのプログラムを変更せずに利用できる

AMP T-Kernelとは (4/4)

- オブジェクトの所在を意識せずに同期・通信機能を使用可能
  - オブジェクトを別のカーネルに移動しても動作する



SMP T-Kernelとは (1/6)

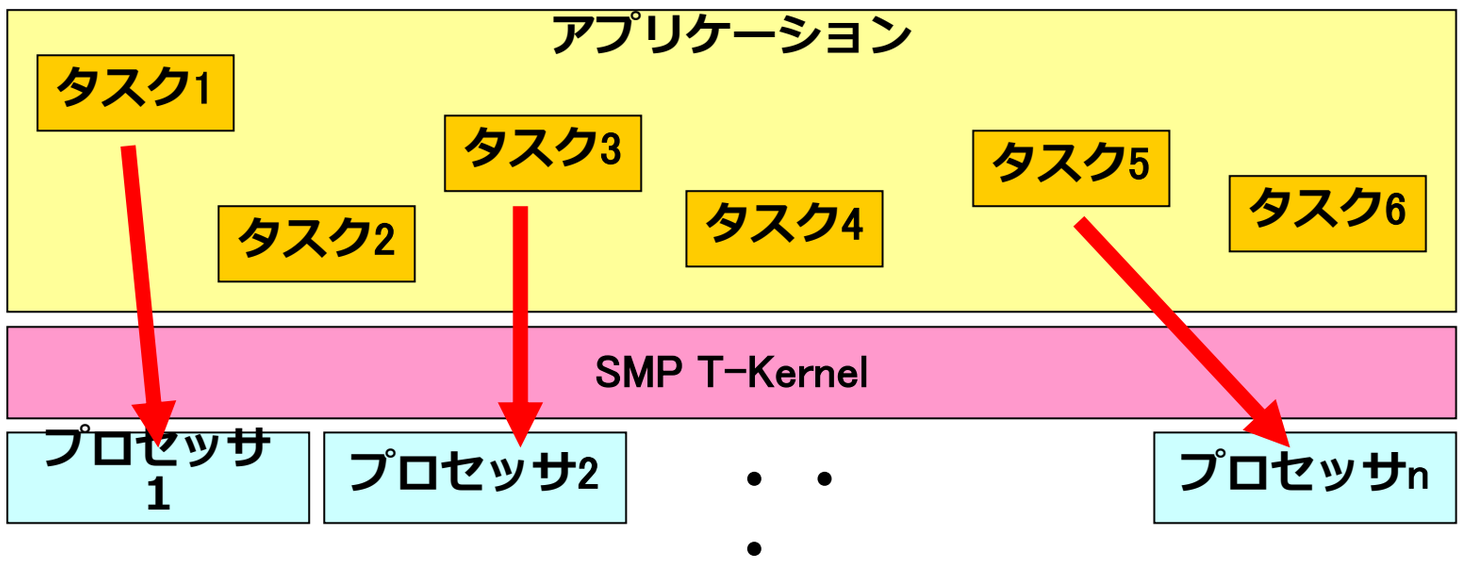
- 対称型マルチプロセッサ（SMP: Symmetric Multiple Processor）に対応したMP T-Kernel

- 個々のプロセッサの用途は限定されておらず、各プロセッサが実行するプログラムはカーネルによって動的に割り当てられる

- 資源は1つのカーネルによって管理される
  - カーネルオブジェクトやメモリなどは全てのプロセッサから同等に操作できる

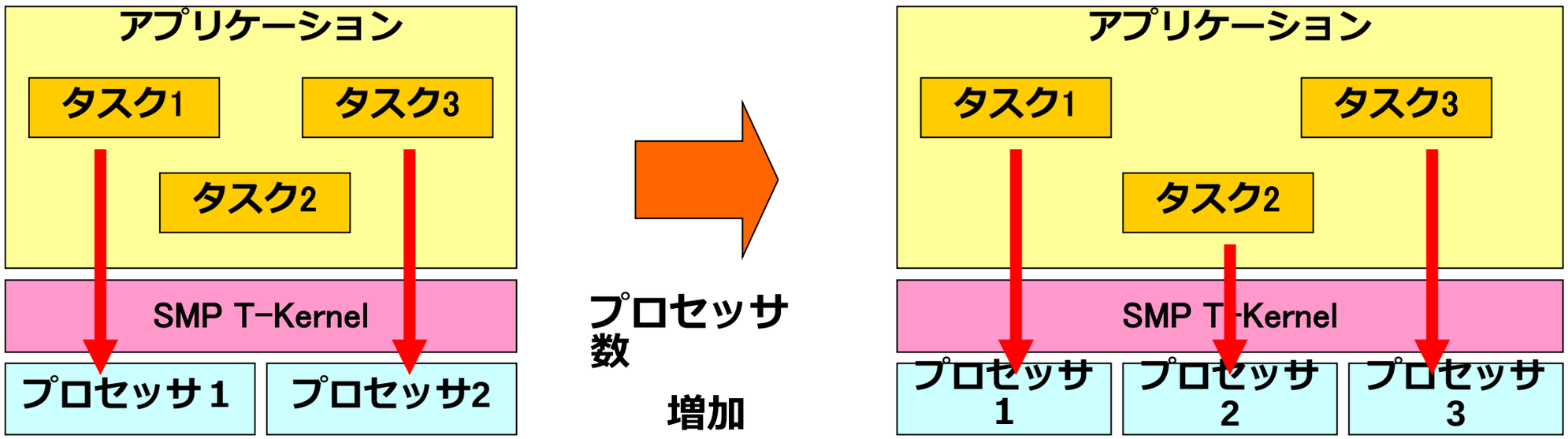
SMP T-Kernelとは (2/6)

- システム全体でカーネルは1つ
- タスクはカーネルによって各プロセッサに動的に割り当てられる



SMP T-Kernelとは (3/6)

- スケーラビリティが高い
  - プロセッサ数を増加させると自動的に性能が向上するプログラムを「作成可能」



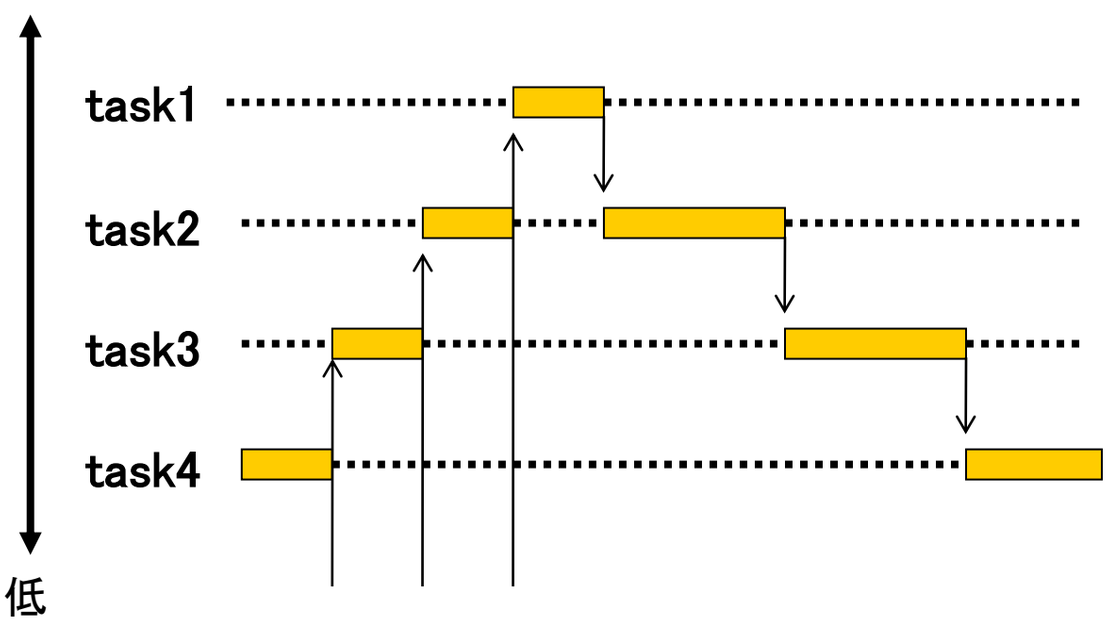
SMP T-Kernelとは (4/6)

- ＜注意点＞
  シングルコアのソフトウェアとの互換性が低い
- タスクやハンドラが複数同時に実行される可能性がある
  - タスクが複数同時に実行される可能性がある
  - ハンドラとタスクが同時に実行される可能性がある
  - 標準のT-Kernelのプログラムはそのままでは動作しない可能性が高い

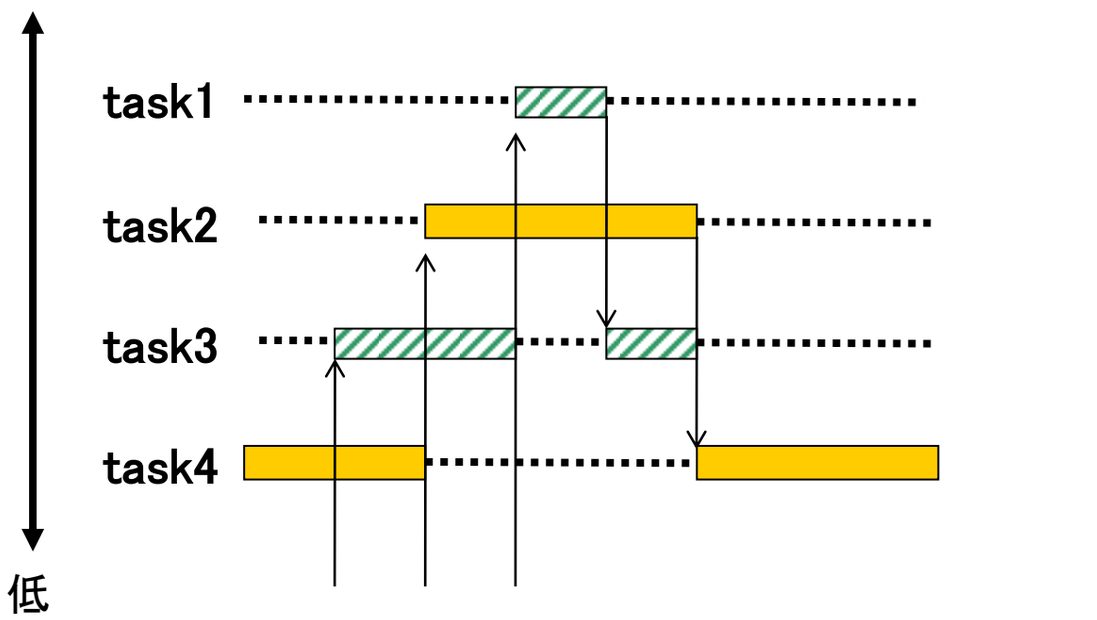
SMP T-Kernelとは (5/6)

- タスクが同時に実行される

コア数　1の場合



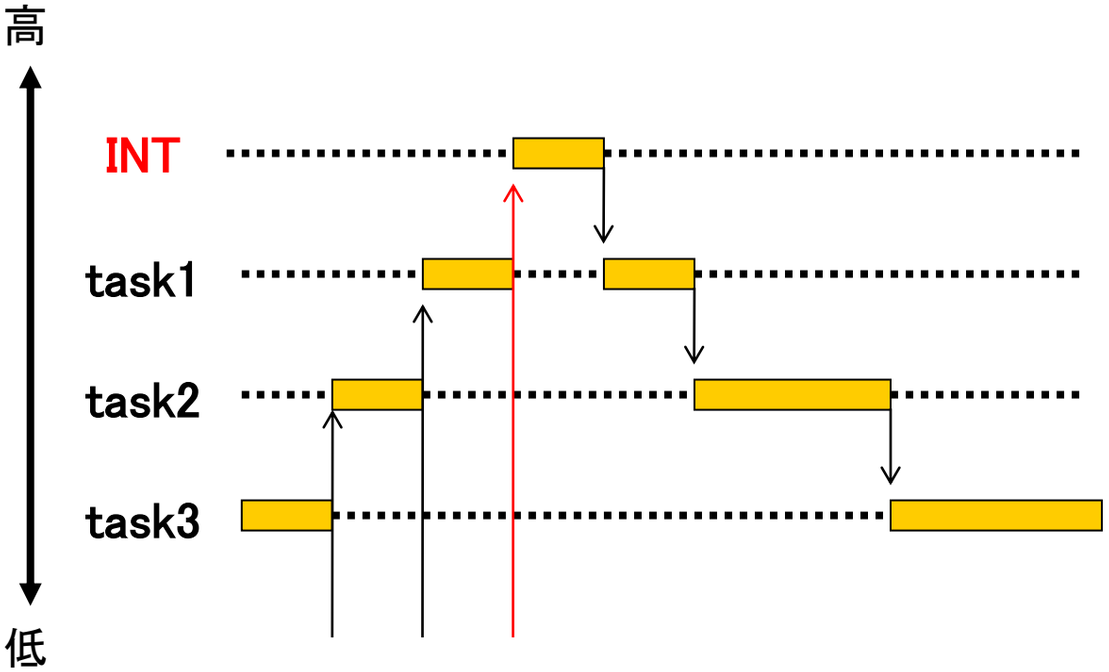
コア数　2の場合



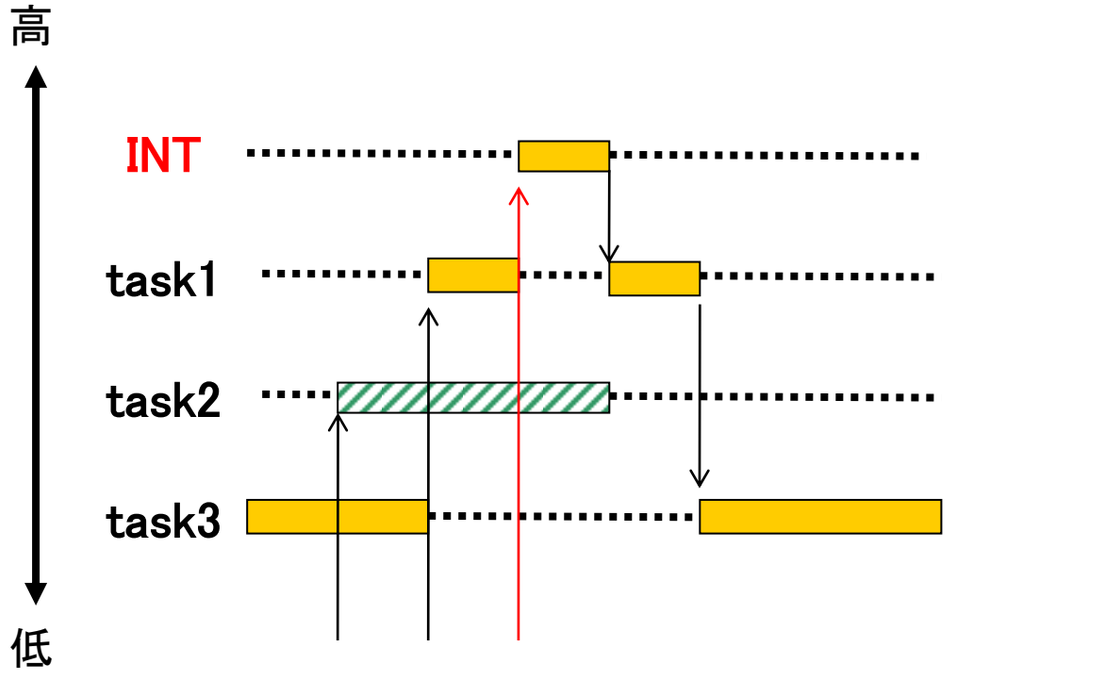
SMP T-Kernelとは (6/6)

- ハンドラとタスクが同時に実行される

コア数　1の場合

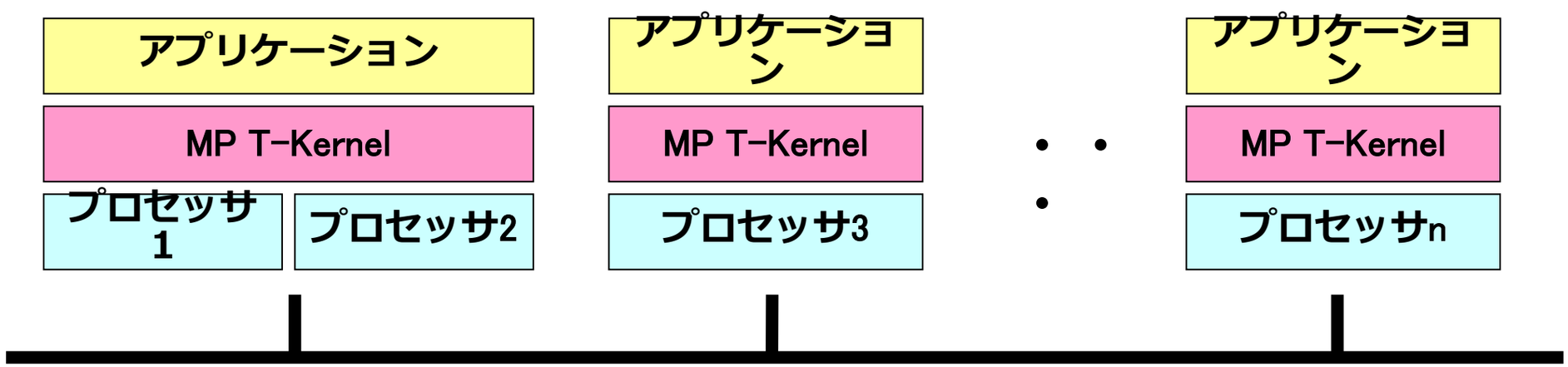


コア数　2の場合



統合したMP T-Kernel

- AMPとSMPを混在させることが可能
  - SMPの互換性の問題を解消しやすい
    - 標準のT-Kernelのプログラムは単独プロセッサで実行
    - 高い処理能力が必要なプログラムは複数プロセッサで実行
  - プロセッサ数の増加に対して柔軟に対応可能

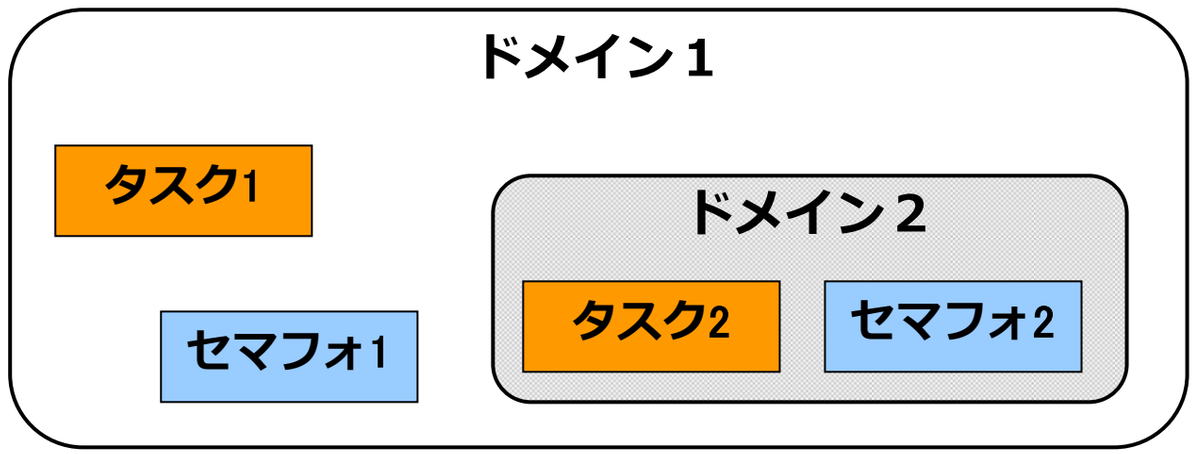


MP T-Kernel 共通の機能

1. ドメインとオブジェクト名

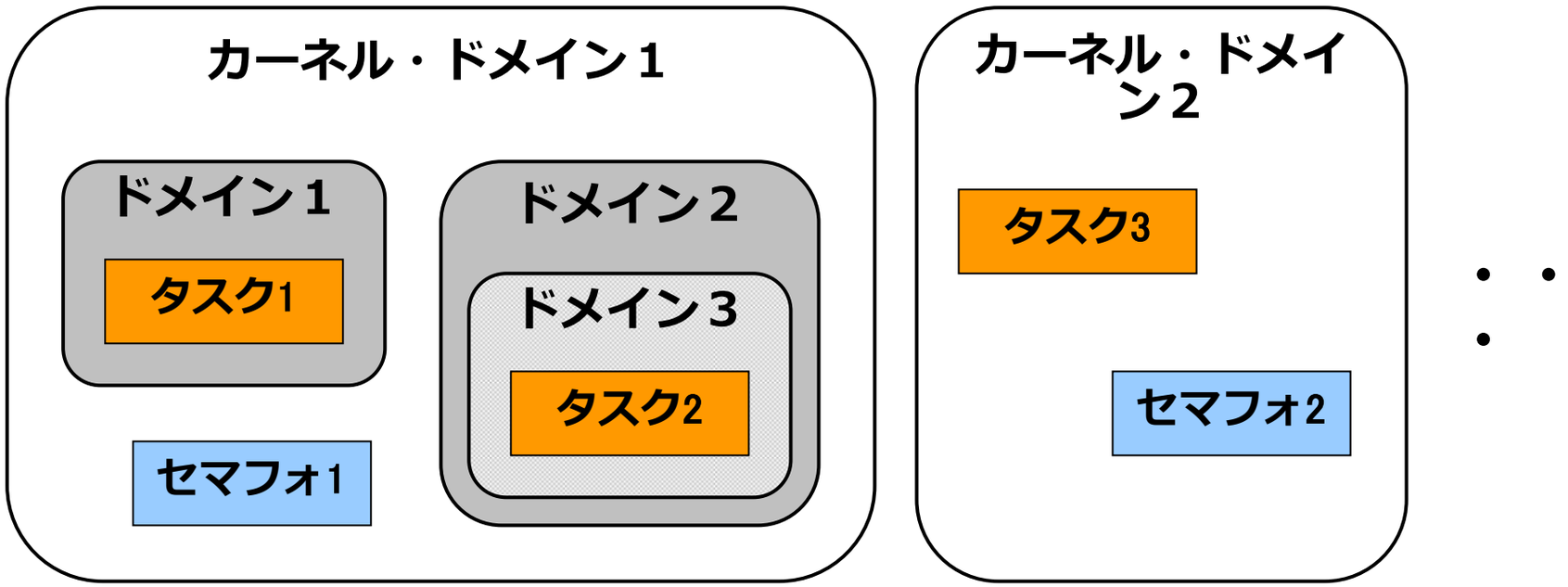
ドメイン (1/2)

- MP T-Kernelで追加されたカーネルオブジェクト
- オブジェクトが所在する場所を示す
- カーネルオブジェクトはいずれかのドメインに所属する
  - ドメインもいずれかのドメインに所属する



ドメイン (2/2)

- 各カーネルに1つカーネル・ドメインが存在する
  - カーネルの初期化時に生成され、削除できない
  - カーネル・ドメインは自身に所属しているとみなす

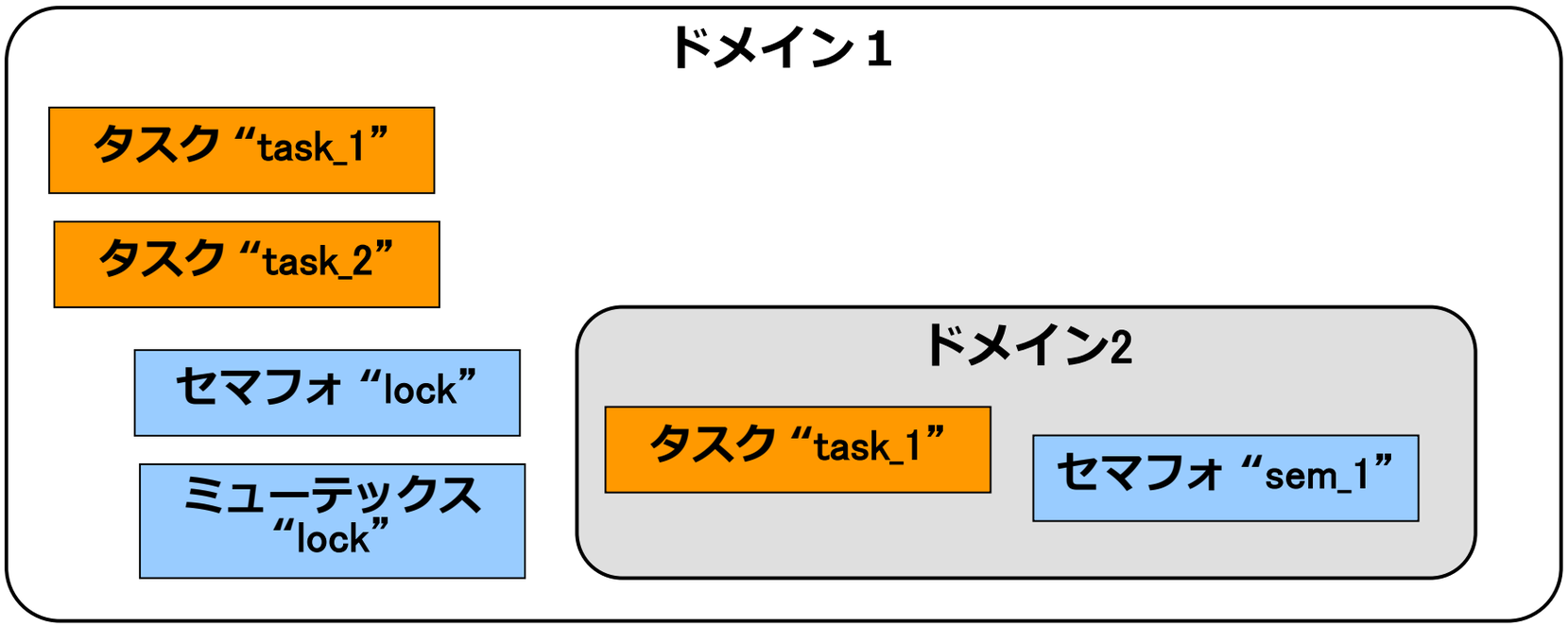


オブジェクト名称 (1/2)

- カーネルオブジェクトには任意のオブジェクト名を指定可能
- オブジェクトのID番号を検索する場合に使用
  - ドメインIDとオブジェクト名称からID番号を検索
  - オブジェクト名称が設定されていないオブジェクトは検索できない

オブジェクト名称 (1/2)

- 同じドメインに所属する同じ種類のオブジェクトは固有の名称でなければならない
  - ドメインが異なれば同じ名前を設定可能
  - オブジェクトの種類が異なれば同じ名前を設定可能



MP T-Kernel 共通の機能

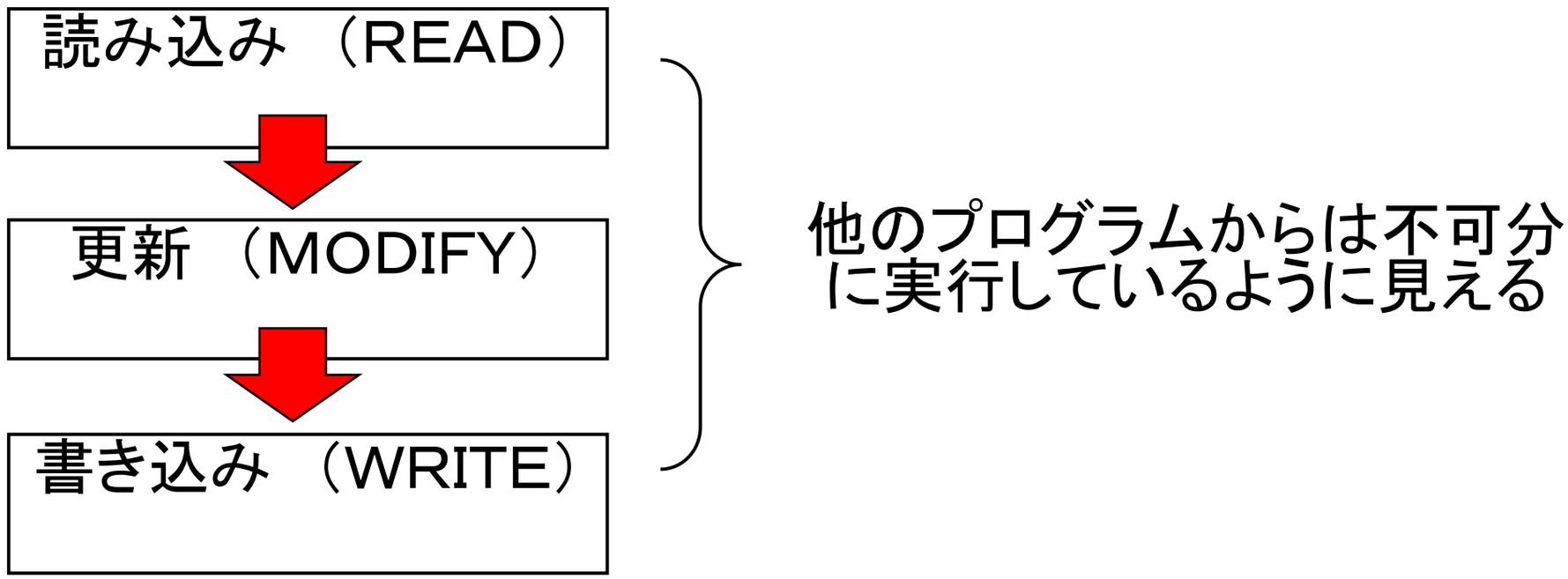
2. プロセッサ間管理機能

プロセッサ間管理機能

- アトミック関数
- スピンロック
- メモリバリア

アトミック関数 (1/6)

- メモリ操作を不可分に実行する関数
  - 一連の処理の途中経過が他のプログラムからは見えないことを保証する関数



アトミック関数 (2/6)

- アトミック関数一覧
  - UW atomic_inc(UW *addr) +1
  - UW atomic_dec(UW *addr) -1
  - UW atomic_add(UW *addr, UW val) +val
  - UW atomic_sub(UW *addr, UW val) -val
  - UW atomic_xchg(UW *addr, UW val) 交換
  - UW atomic_cmpxchg(UW *addr, UW val, UW cmp) 比較と交換
  - UW atomic_bitset(UW *addr, UW setptn) ビットセット
  - UW atomic_bitclr(UW *addr, UW clrptn) ビットクリア

アトミック関数 (3/6)

- 不可分性が考慮されていないと・・・

初期条件

```
INT a = 0;
```

プロセッサ 1

```
a++;
```

プロセッサ 2

```
a++;
```

(ほぼ)同時に実行

a の値は 1 or 2

(1) a = 2 問題なし

ステップ	プロセッサ1	プロセッサ2	a
1	LOAD tmp, [a]		0
2	ADD tmp, 1		0
3	STORE tmp, [a]		1
4		LOAD tmp, [a]	1
5		ADD tmp, 1	1
6		STORE tmp, [a]	2

(2) a = 1 問題あり

ステップ	プロセッサ1	プロセッサ2	a
1	LOAD tmp, [a]		0
2	ADD tmp, 1	LOAD tmp, [a]	0
3	STORE tmp, [a]	ADD tmp, 1	1
4		STORE tmp, [a]	1

アトミック関数 (4/6)

- セマフォなどを使い排他制御をおこなえば不可分操作を実現可能だが・・・
  - 一番の問題は実行速度
  - 待ち状態になれない場所では使用できない

プロセッサ1

```
tk_wai_sem();
a++;
tk_sig_sem();
```

プロセッサ2

```
tk_wai_sem();
a++;
tk_sig_sem();
```

不可分性は保証される

速度が問題

a = 2 問題なし

ステップ	プロセッサ1	プロセッサ2	a
1	tk_wai_sem();	tk_wai_sem();	0
2	a++;		1
3	tk_sig_sem();		1
4		a++;	2
5		tk_sig_sem();	2

アトミック関数 (5/6)

- マルチプロセッサの場合、割込み禁止では不可分操作を実現できない
  - 割込みを禁止しても他のプロセッサは停止しない

プロセッサ1

```
DI(intsts);
a++;
EI(intsts);
```

プロセッサ2

```
DI(intsts);
a++;
EI(intsts);
```

同時実行が可能

不可分操作になっていない

a = 1 問題あり

ステップ	プロセッサ1	プロセッサ2	a
1	DI(intsts);	DI(intsts);	0
2	LOAD tmp, [a]	LOAD tmp, [a]	0
3	ADD tmp, 1	ADD tmp, 1	0
4	STORE tmp, [a]	STORE tmp, [a]	1
5	EI(intsts);	EI(intsts);	1

アトミック関数 (6/6)

- アトミック関数は不可分操作を高速に実行
  - 数個の命令で実現可能
  - どこでも使用可能

プロセッサ1

```
atomic_inc(&a);
```

プロセッサ2

```
atomic_inc(&a);
```

不可分性を保証

速度も高速

atomic_inc(&a)の実装例

```
loop:
  LL  tmp, [a]        // tmp = a; アドレスに「タグ」
  ADD tmp, 1
  SC  sts, tmp, [a]   // 「タグ」があったら a = tmp
  CMP sts, 0
  BNE loop            // 成功するまで繰り返す
  RET
```

スピンロック (1/3)

- プロセッサ間で資源を排他制御する場合に使用する機能
- ロックの獲得を待っている間はビジーウエイト状態
  - 対策：リードライト・スピンロック、トライロック
- 主にドライバなどのシステムプログラムが使用する
  - アプリケーションは通常使用しない

スピンロック (2/3)

- 2種類のロック
  - スピンロック
    - ロックを獲得できるのは1つのプログラムのみ
  - リードライト・スピンロック
    - 複数のプログラムがロックを獲得できる可能性がある
- 割込み禁止を伴うスピンロック操作が用意されている
  - スピンロックは一般的に割込み禁止状態で獲得する

```
DI(intsts);
SpinLock(&lock);
/* 処理 */
SpinUnlock(&lock);
EI(intsts);
```

```
ISpinLock(&lock, &intsts);
/* 処理 */
ISpinUnlock(&lock, &intsts);
```

スピンロック (3/3)

- ロック獲得中の処理は最小限に
  - システム全体のパフォーマンスに影響を与える可能性がある

```
INT     a;

SpinLock(&lock);
a = 0;
for ( i = 0; i < 10; i++ ) {
        a += shared_data[i];
}
SpinUnlock(&lock);
```

```
INT     a;

a = 0;
SpinLock(&lock);
for ( i = 0; i < 10; i++ ) {
        a += shared_data[i];
}
SpinUnlock(&lock);
```

```
INT     a;

a = 0;
for ( i = 0; i < 10; i++ ) {
        SpinLock(&lock);
        a += shared_data[i];
        SpinUnlock(&lock);
}
```

メモリバリア (1/2)

- プロセッサのメモリアクセスの順序が命令順とは異なる可能性がある
  - プロセッサに都合の良い順番でメモリをアクセス
- プログラム側がメモリアクセスの順序を意識しなければならない場合がある
  - 他のプロセッサからどのように見えるか
    - SH7776、SH7786は順序を意識する必要なし
    - NaviEngineは順序を意識する必要がある

メモリバリア (2/2)

- 複数プロセッサ間でメモリアクセスの順序を保証したい場合には明示的なメモリバリアが必要
  - メモリを使って同期・通信をするプログラム
- メモリアクセスの順序が完全に保証されているプロセッサでは何も実行しない
  - エラーにならない

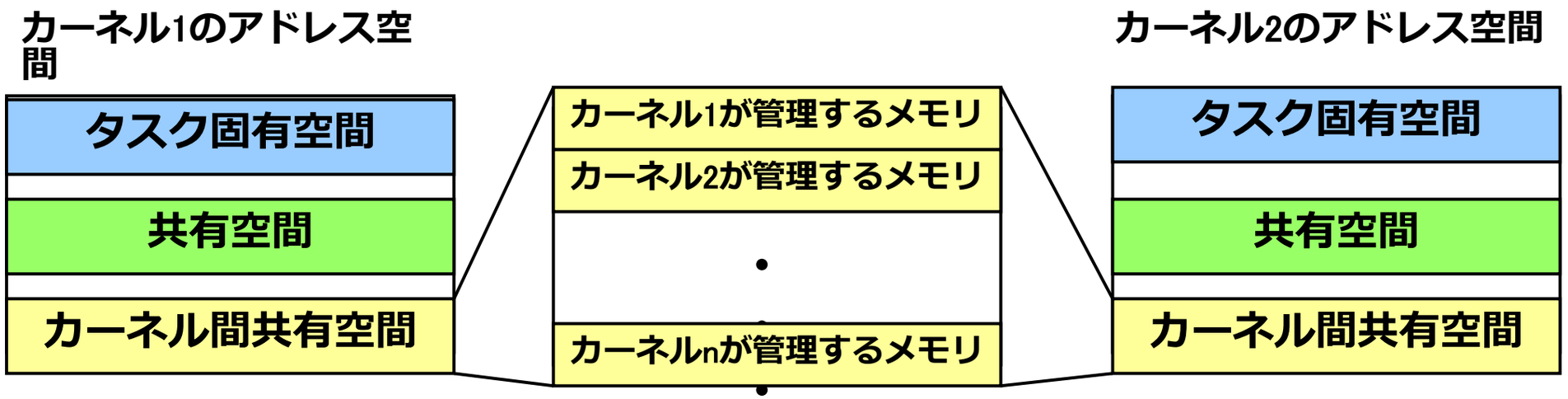
AMP T-Kernel 固有の機能

1.カーネル間共有空間

カーネル間共有空間とは

- 全てのカーネルで共有する空間
  - カーネル間でデータを共有したい場合に使用する

- 各カーネルは、自身が管理するカーネル間共有空間のメモリを有する



カーネル間共有空間の注意点

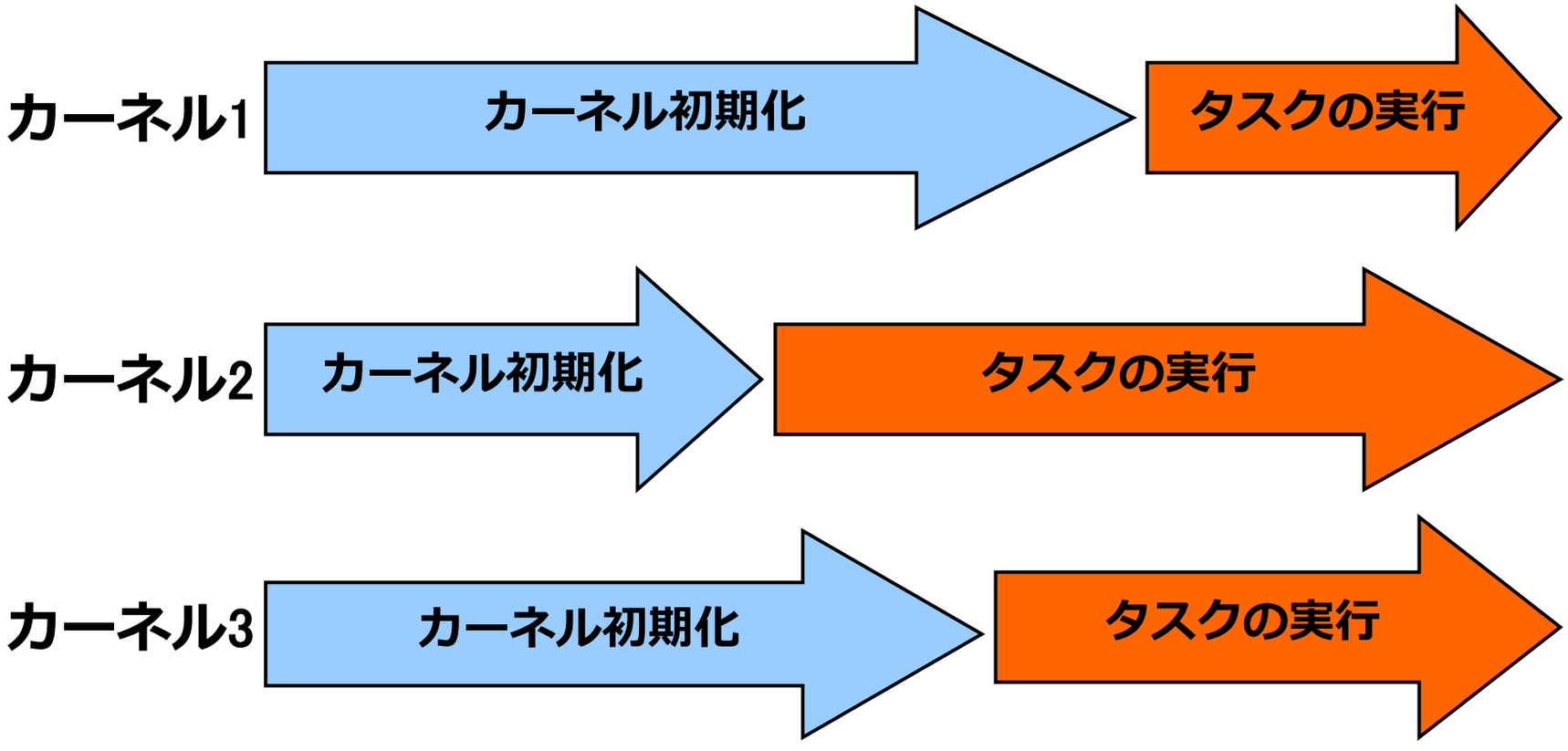
- カーネル間共有空間の有無は実装定義
  - 実装仕様書の確認が必要
    - SH7776、SH7786、NaviEngineはカーネル間共有空間が存在する
- メモリの一貫性が保証されていない可能性がある
  - カーネル間共有空間の見え方を保証していないハードウエアは注意が必要
    - NaviEngineは見え方を保証していない
    - メモリバリアを使用しなければならない場合がある

AMP T-Kernel 固有の機能

2.初期化ハンドラ

カーネルの初期化処理

- 各AMP T-Kernelの初期化処理の完了は同期しない
  - タスクの実行が開始した段階では他カーネルとの通信は保証されていない



初期化ハンドラとは (1/3)

- カーネル初期化処理の最後に実行されるハンドラ
  - 初期タスクの実行開始前に実行される
- 初期化処理の終了をカーネル間で同期させたい場合に使用する
  - 同期させないことも可能
- 各カーネルに1つだけ登録することができる
  - 登録方法は実装定義
  - 登録しないことも可能

初期化ハンドラとは (2/3)

- 初期化ハンドラはタスク独立部として実行される
- 初期化ハンドラは以下のように定義される

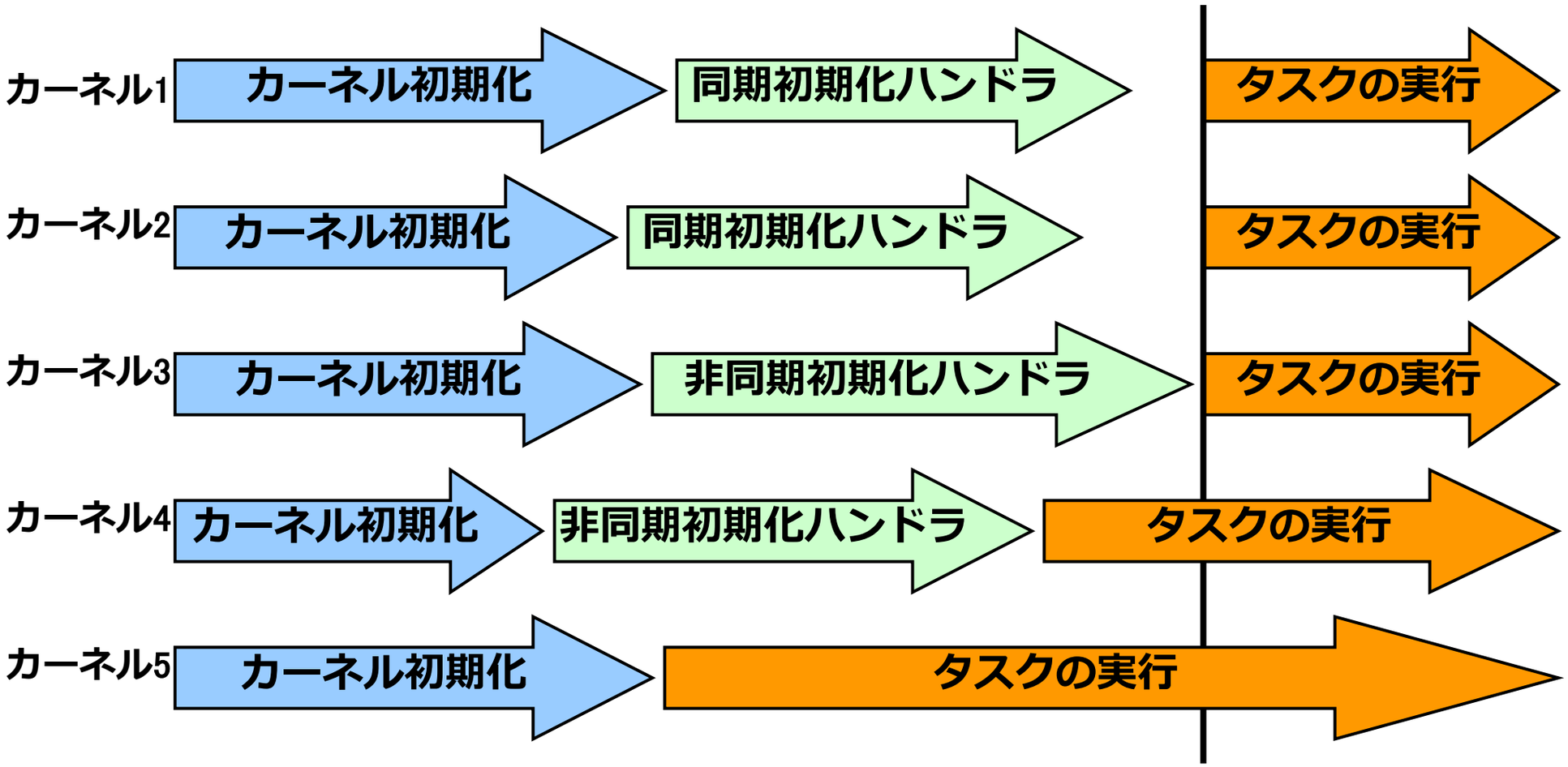
```
W init_hdr( void );
```

初期化ハンドラの戻値:

```
#define TINI_SYNC      1    // 初期化時の同期あり
#define TINI_NOSYNC    0    // 初期化時の同期なし
```

初期化ハンドラとは (3/3)

- 初期化ハンドラによる同期の例



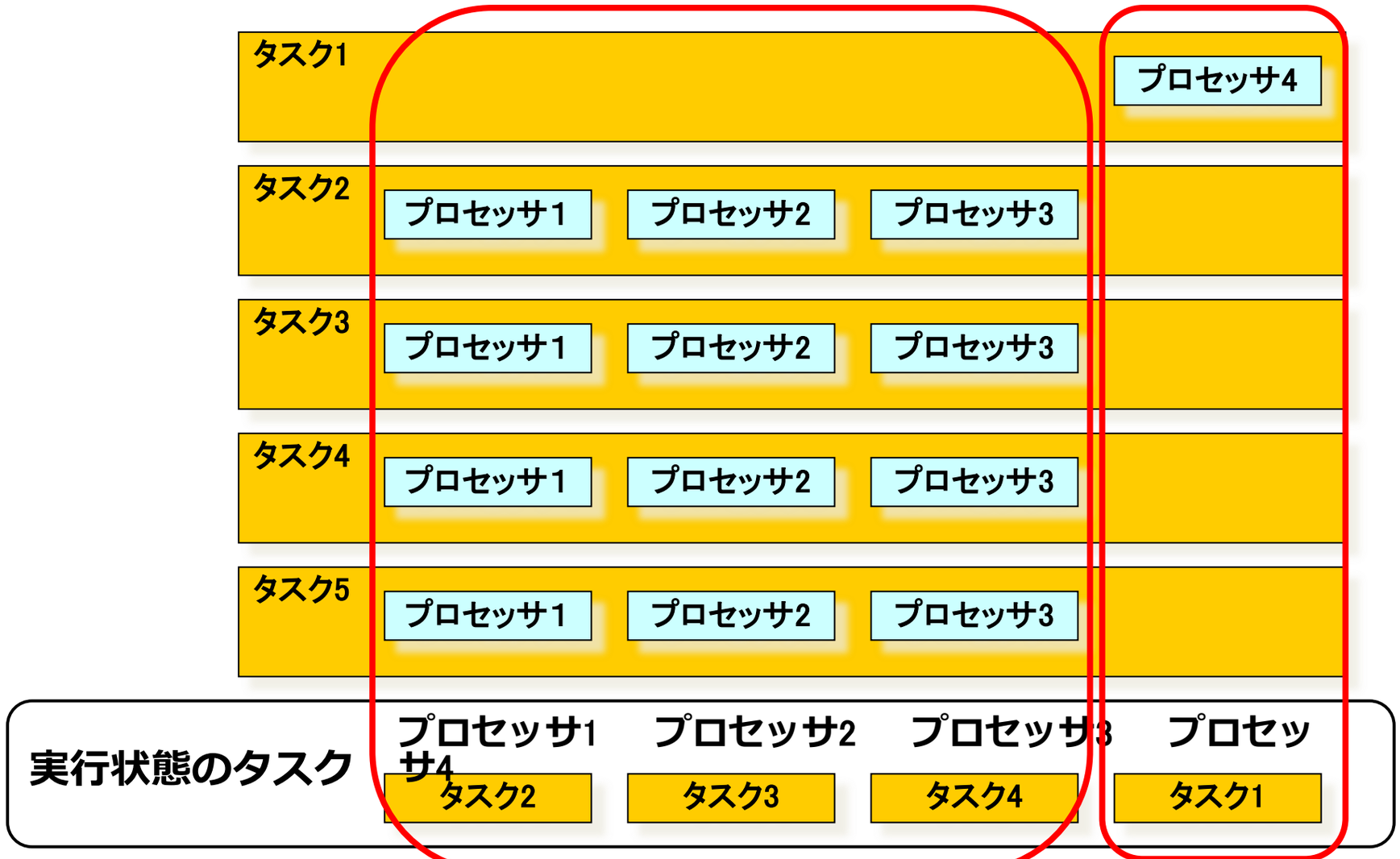
SMP T-Kernel 固有の機能

実行プロセッサ指定

実行プロセッサ指定 (1/4)

- プログラム（タスク、ハンドラ）を実行するプロセッサを指定する機能
  - タスク、周期ハンドラ、アラームハンドラ、割込みハンドラの実行プロセッサを指定できる
- タスクやハンドラを特定のプロセッサで実行させたい（orさせたくない）場合に使用する
  - 排他制御の問題を解消したい場合やリアルタイム性を向上させたい場合などに利用できる

実行プロセッサ指定 (2/4)

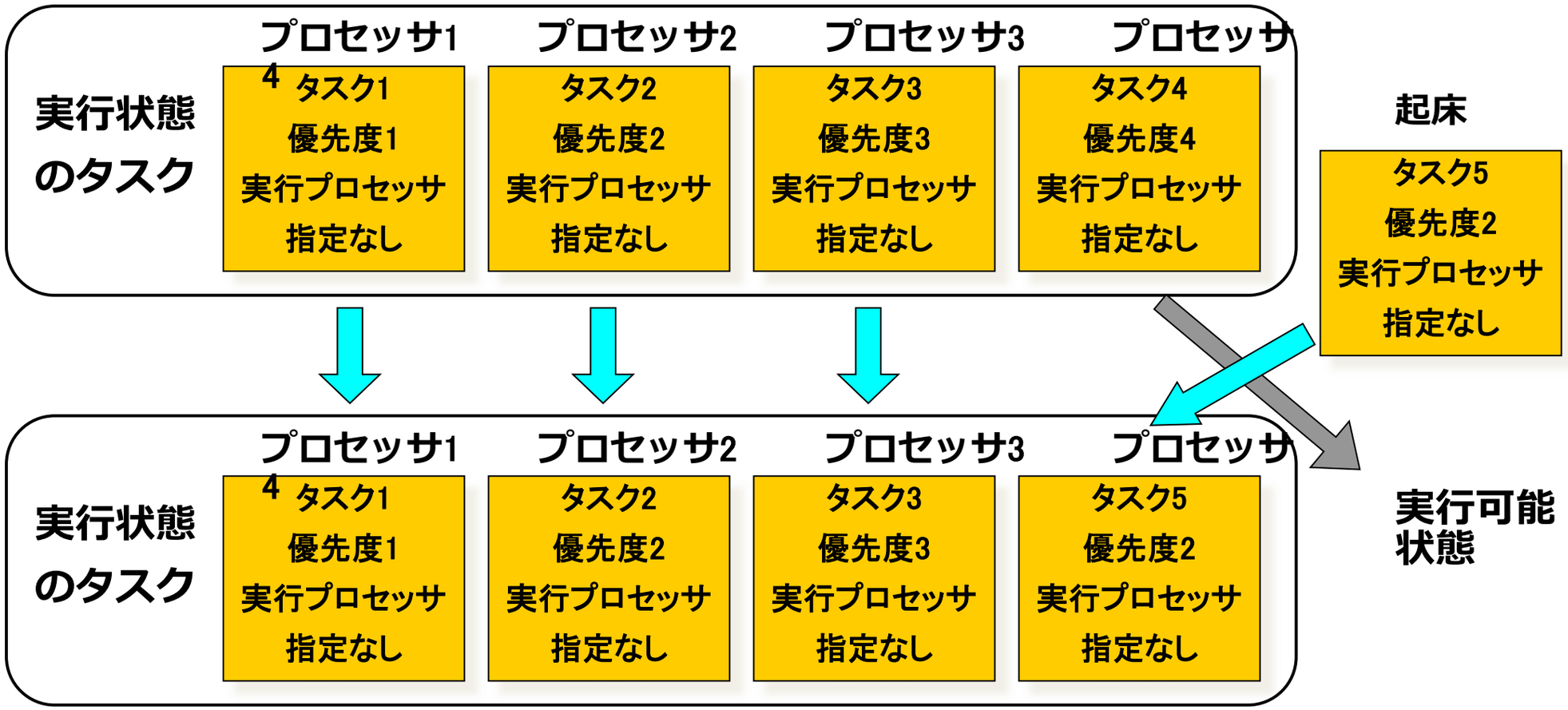


タスクの実行プロセッサ指定 (2/6)

- タスクの実行プロセッサを指定した場合の注意点
  - タスクの実行状態が継続される場合でもプロセッサが切り替わる可能性がある
    - 性能低下の要因となる
  - 優先順位の逆転が生じる可能性がある
  - スケーラビリティが低下する
    - プロセッサ数の異なるハードウエアに移植した場合に問題となる可能性がある

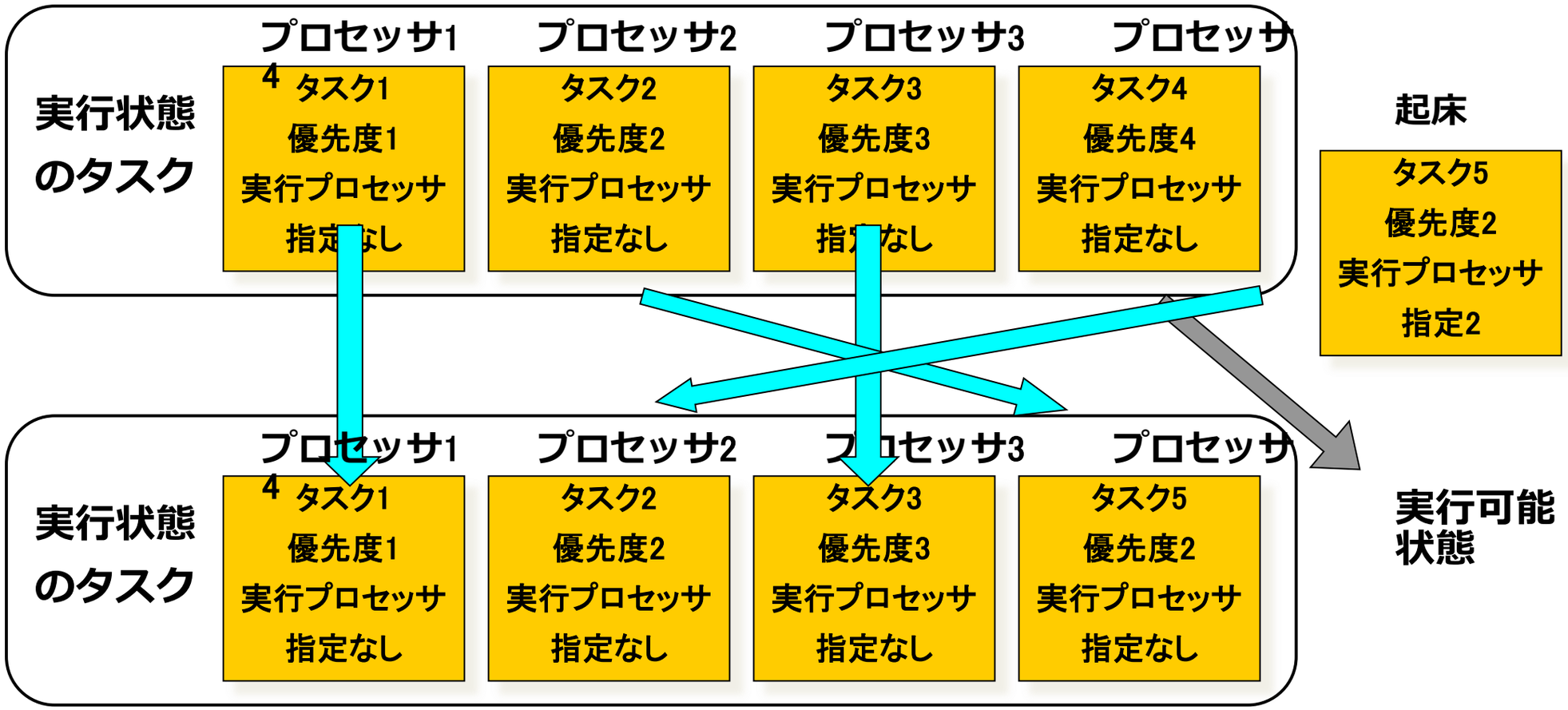
タスクの実行プロセッサ指定 (3/6)

- 実行プロセッサを指定していない場合



タスクの実行プロセッサ指定 (4/6)

- プロセッサが切り替わる例



タスクの実行プロセッサ指定 (5/6)

MP T-Kernelの入手方法

MP T-Kernelの入手方法 (1/2)

- トロンフォーラムのWebサイト
  http://www.tron.org/ja/
  から入手可能

- ライセンス
  - 無償
  - ソースコードの開示義務なし
  - 商用利用可能

MP T-Kernelの入手方法 (2/2)

- 仕様書
  - AMP/SMP T-Kernel 仕様書
  - AMP/SMP T-Kernel Standard Extension 仕様書
- ソースコード
  - AMP/SMP T-Kernel ソースコード
  - AMP/SMP T-Kernel Standard Extension ソースコード
  - サンプルプログラム
- GNUツール (コンパイル環境)

【講座】T-Kernel/ITRON入門テキスト「MP T-Kernel入門 」

著者　TRON Forum

【ご注意およびお願い】

1.本テキストの中で第三者が著作権等の権利を有している箇所については、利用者の方が当該第三者から利用許諾を得てください。

2.本テキストの内容については、その正確性、網羅性、特定目的への適合性等、一切の保証をしないほか、本テキストを利用したことにより損害が生じても著者は責任を負いません。

3.本テキストをご利用いただく際、可能であれば office@tron.org までご利用者のお名前、ご所属、ご連絡先メールアドレスをご連絡いただければ幸いです。