オペレーティングシステム

第4回

プロセス管理とスケジューリング

プロセス生成とスレッド

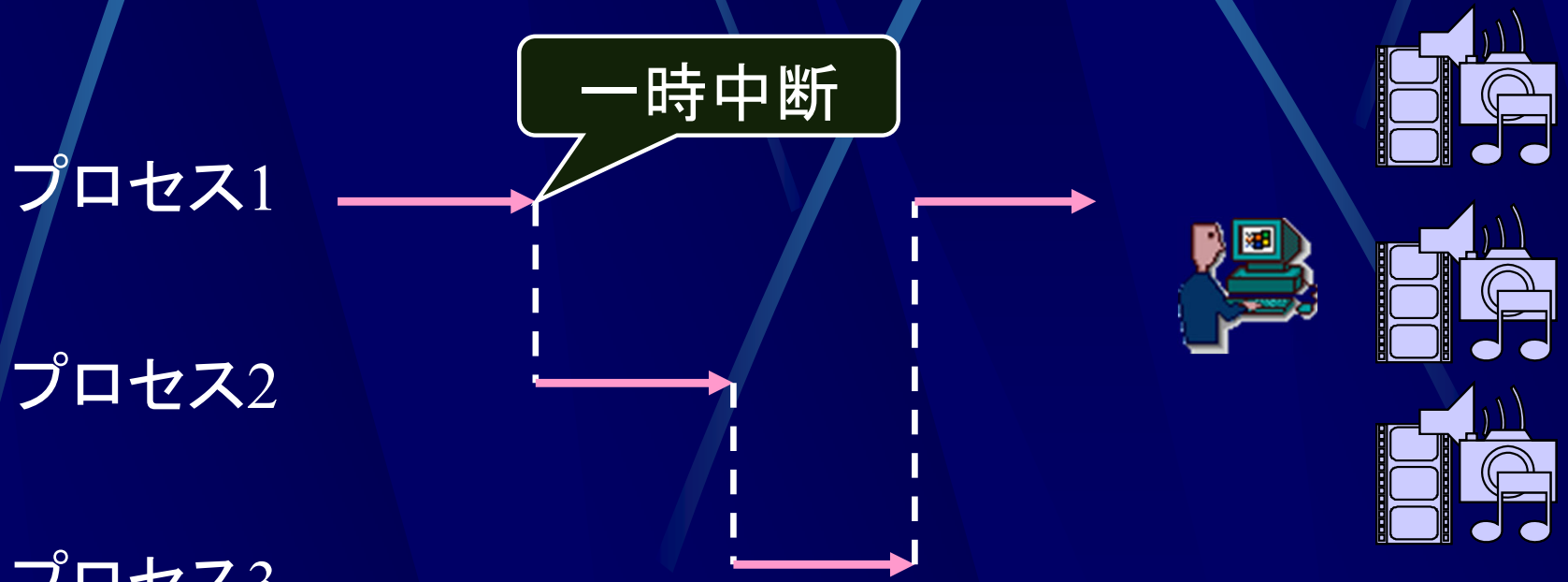
http://www.info.kindai.ac.jp/OS

E館3階E-331 内線5459

takasi-i@info.kindai.ac.jp

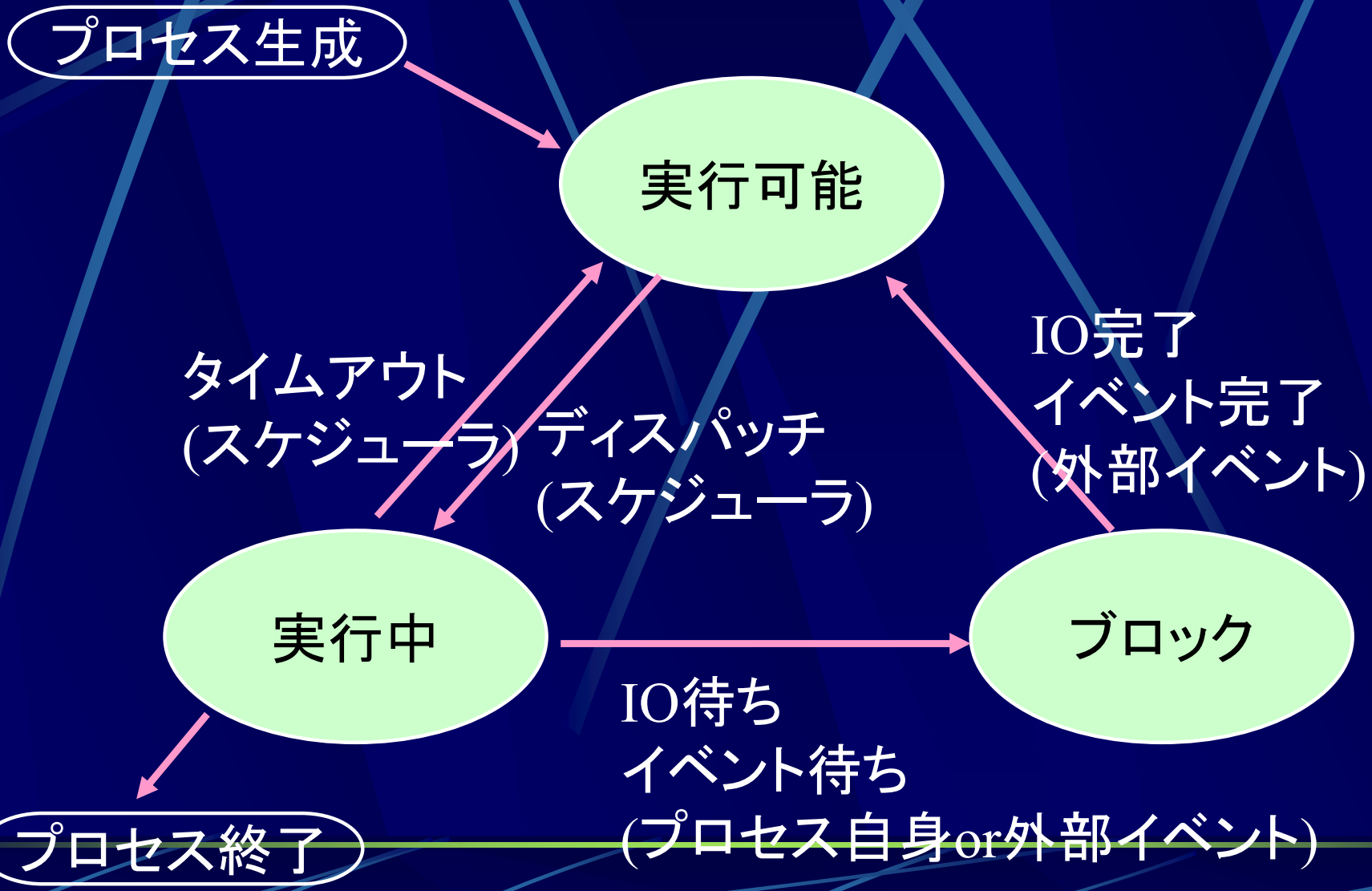
プロセスの並行処理

- 並行処理(concurrent processing)
  - 複数のプロセスを(見かけ上)同時に実行

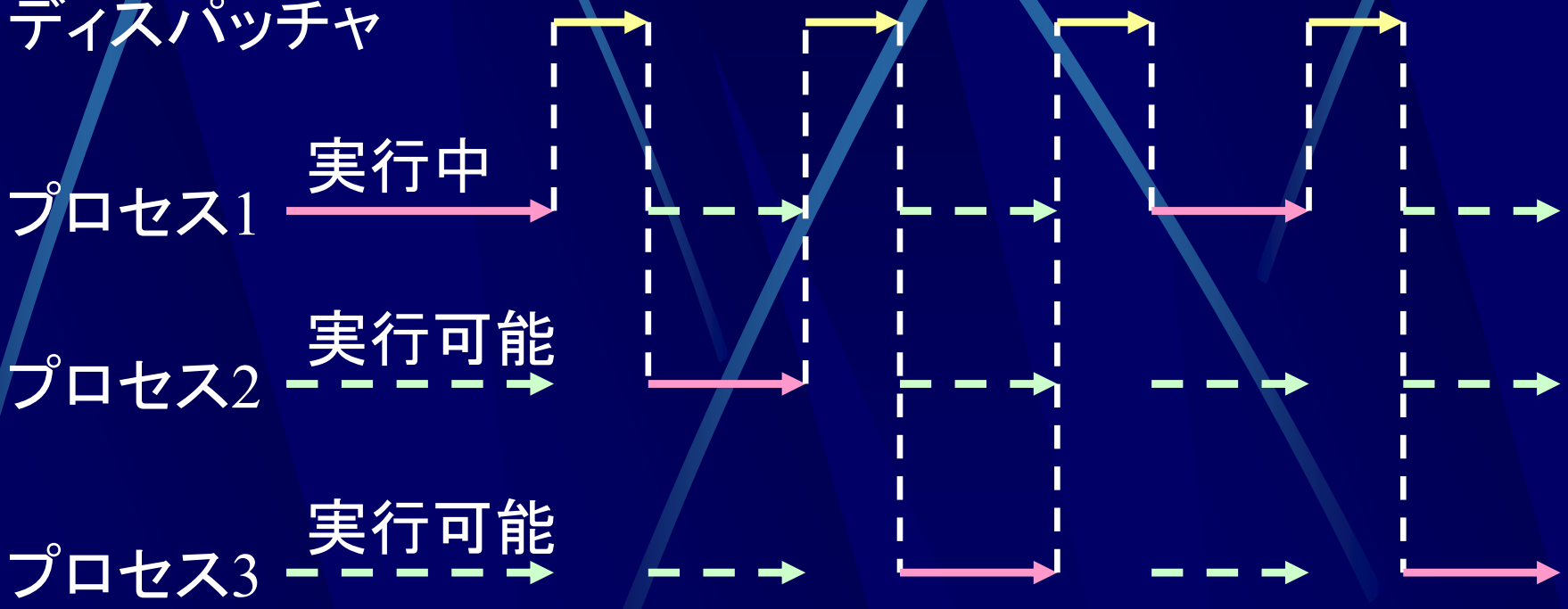


ユーザにとっては
3つのプロセスが
同時に実行されている

プロセスの状態遷移

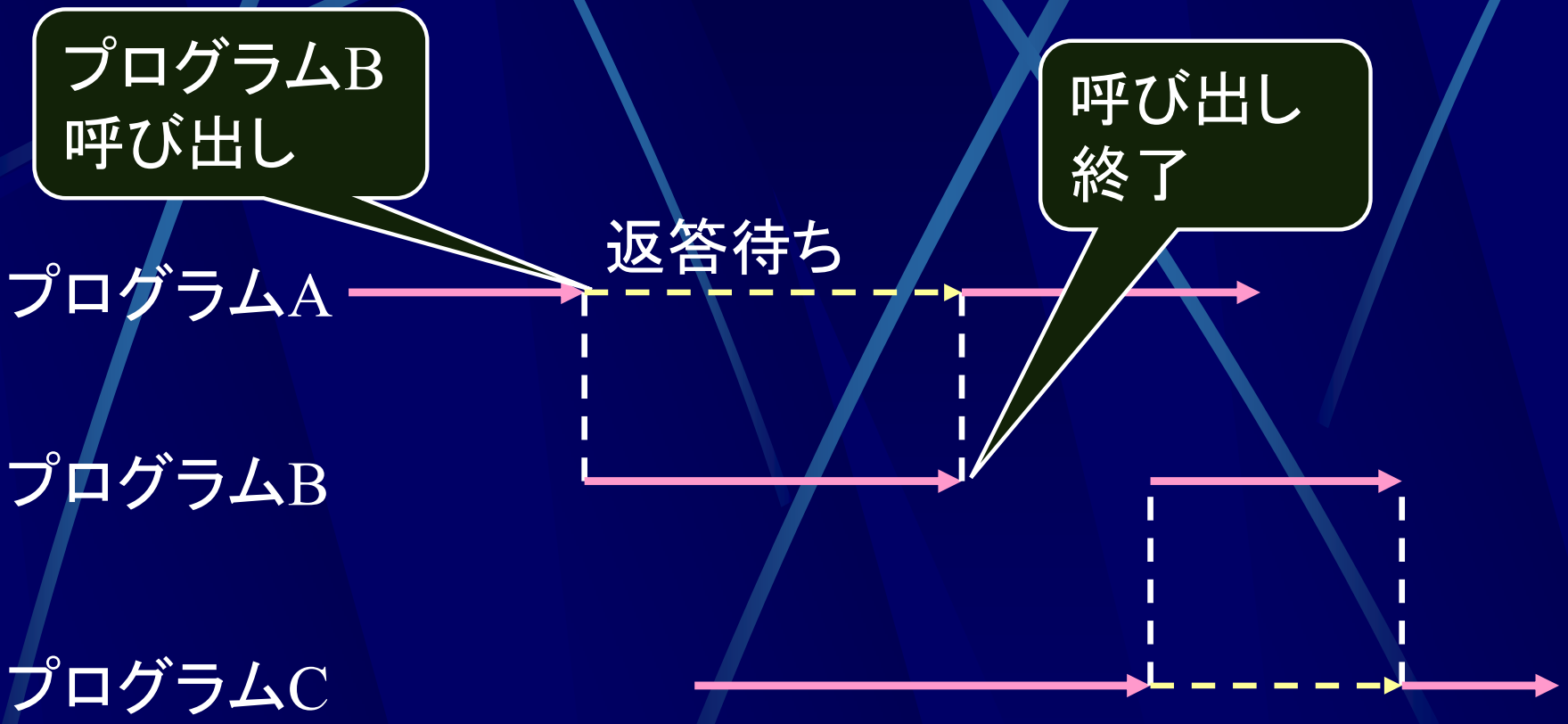

プロセス生成
実行可能
タイムアウト
(スケジューラ)
ディスパッチ
(スケジューラ)
IO完了
イベント完了
(外部イベント)
実行中
ブロック
IO待ち
イベント待ち
(プロセス自身or外部イベント)
プロセス終了

プロセスの状態遷移



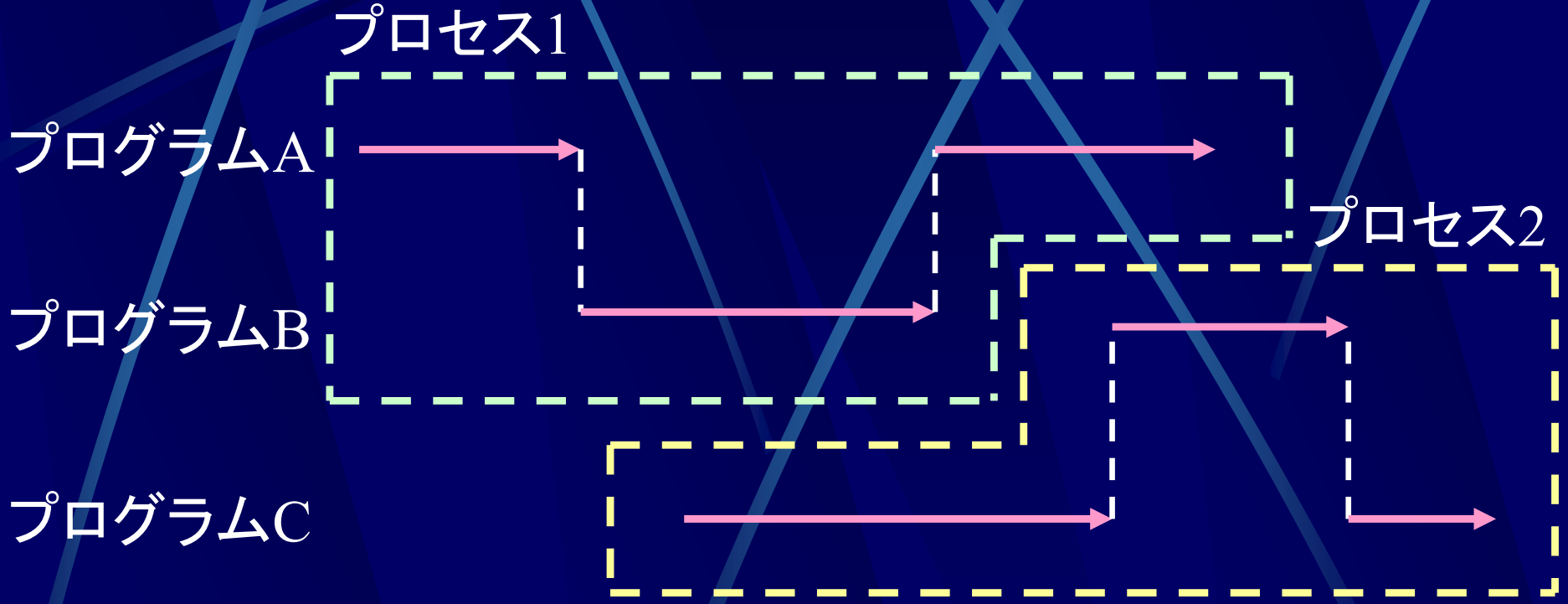
プロセスを高速で切り替えることにより見かけ上
複数のプロセスが同時に処理できる

プログラムの呼び出し



プログラムは他のプログラムから
呼び出される場合もある

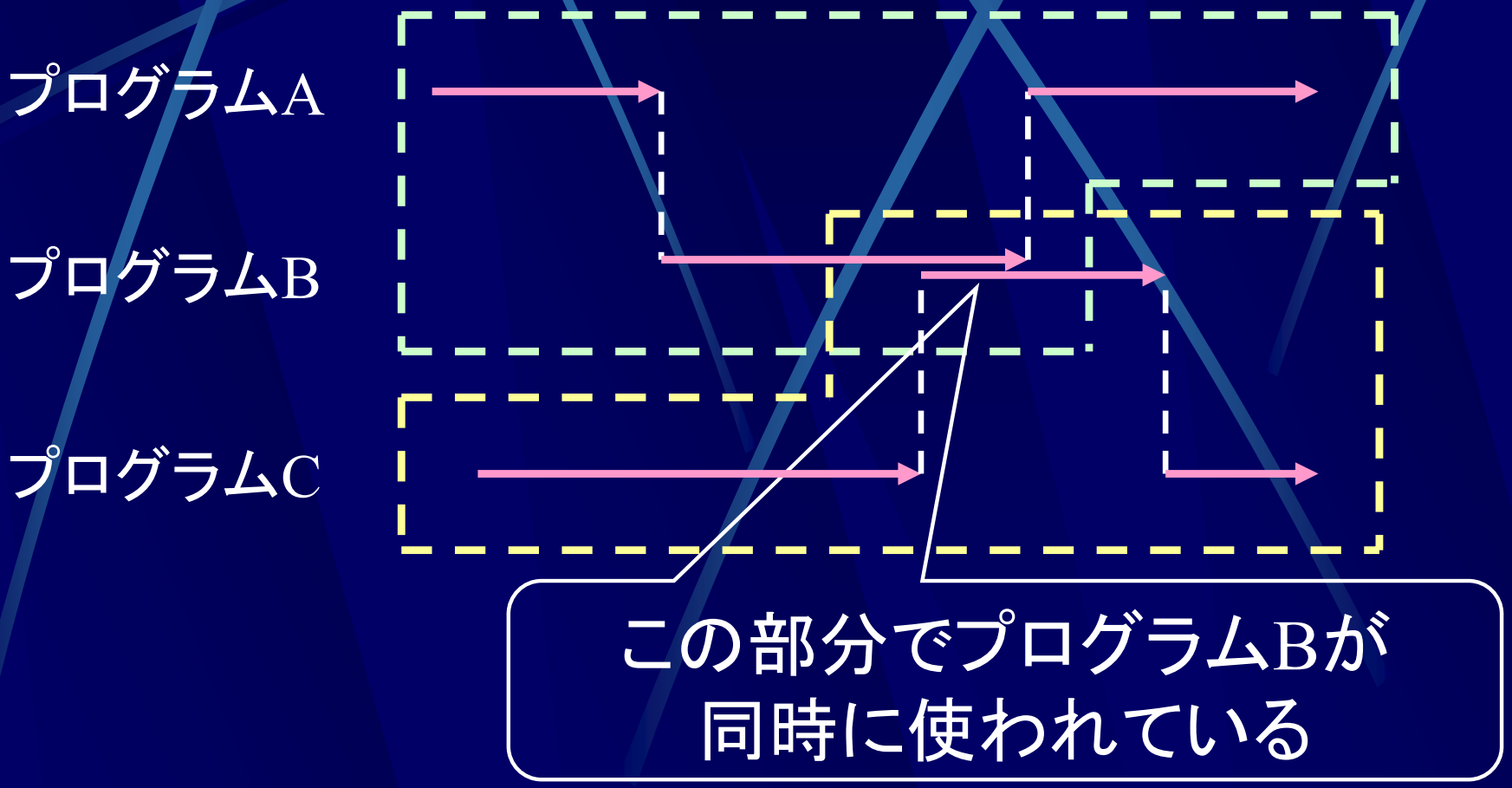
プロセスの同時処理



プロセスは見かけ上同時に処理できる

⇒上記のプロセス 1, 2 も同時に処理できる

プロセスの同時処理



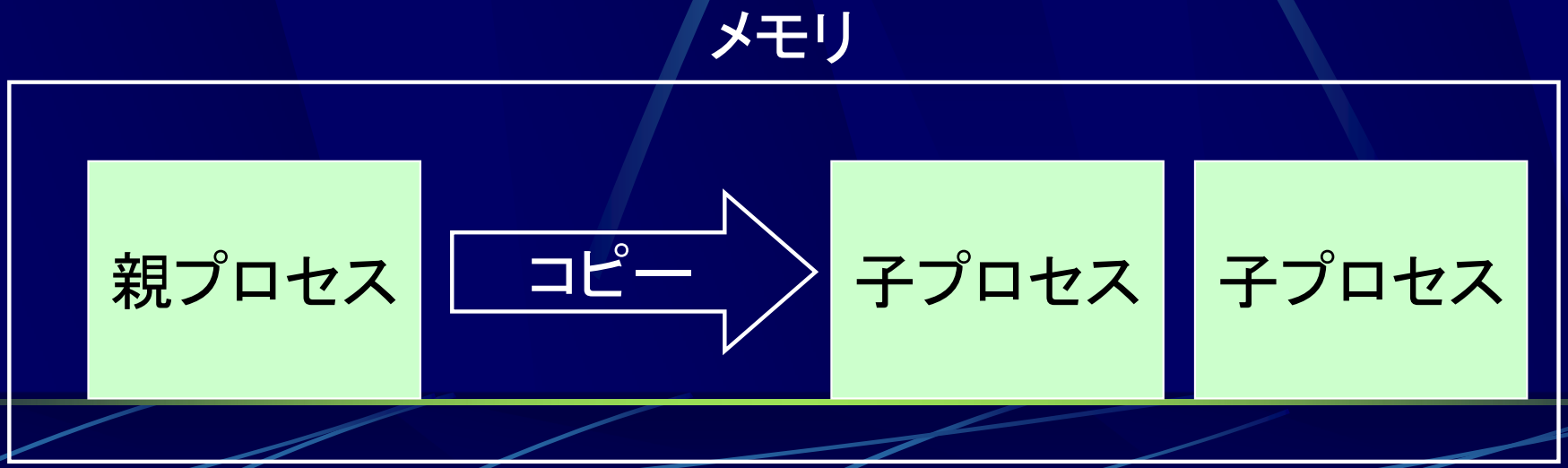
1つのプログラムは同時に複数個使える必要がある

プロセスのコピー

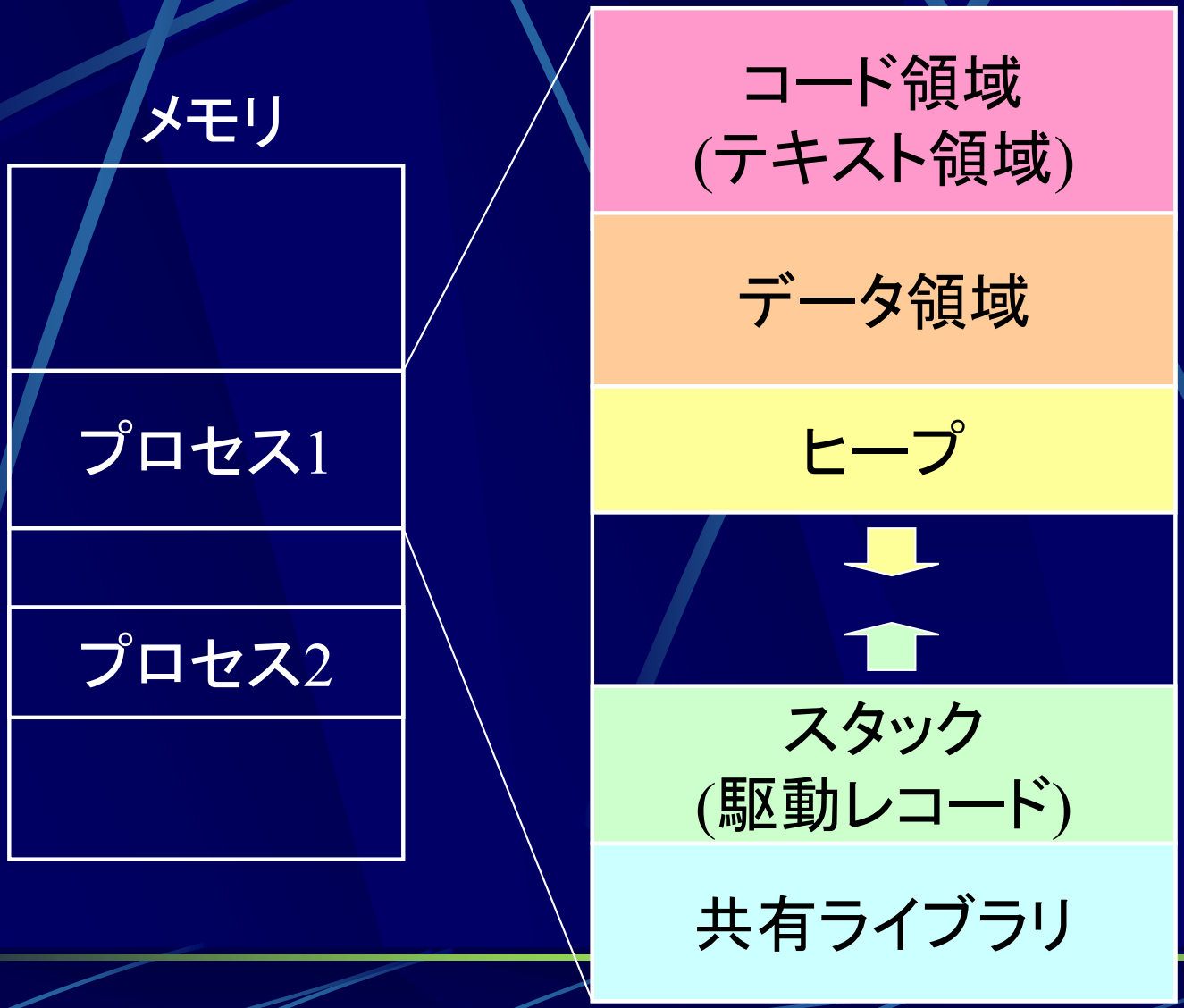
プロセス＝プログラムとその実行に必要なデータ

同じプロセスを(見かけ上)同時に実行するには？

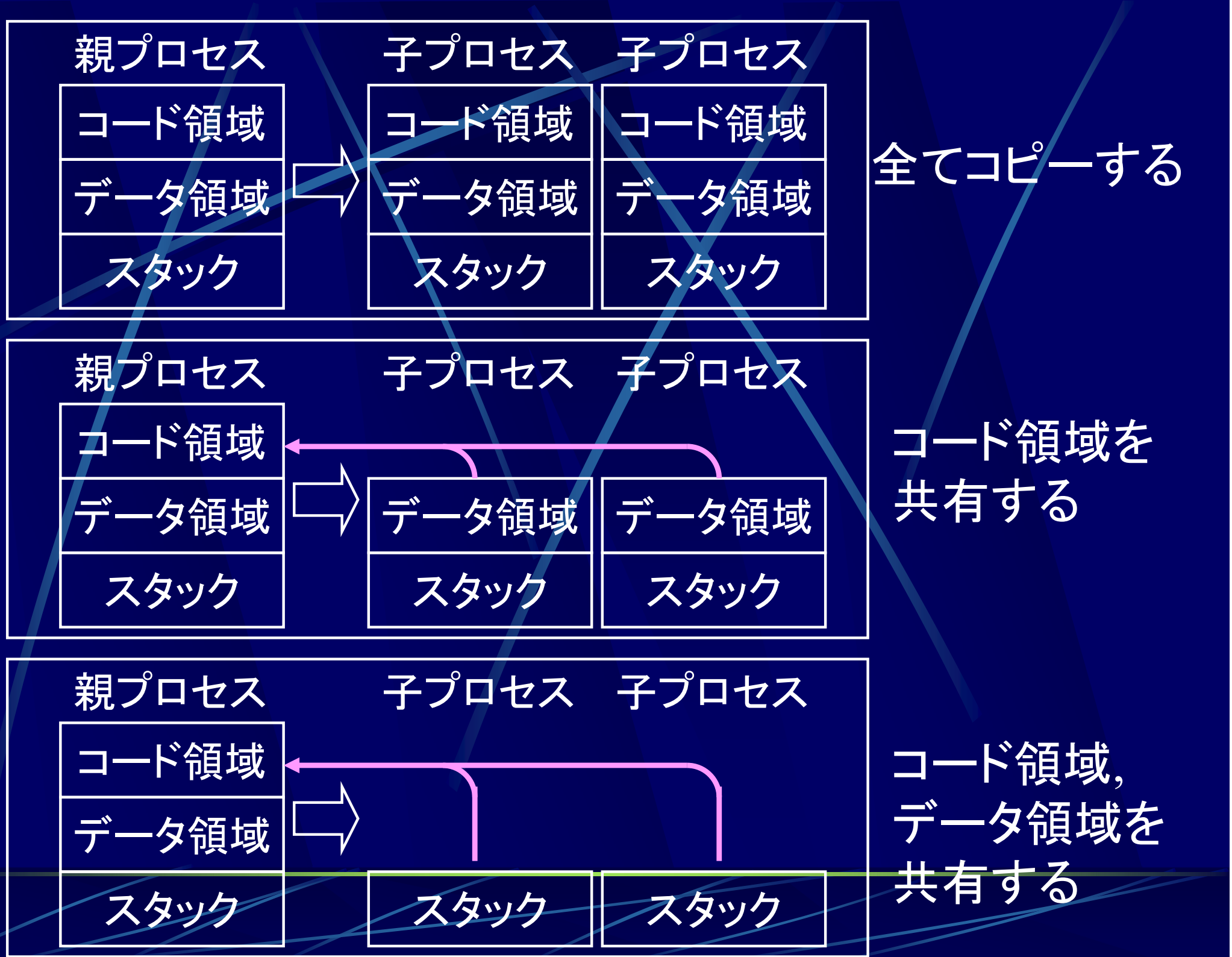
実行に必要なものをコピーする



プロセス(プログラム)の構造



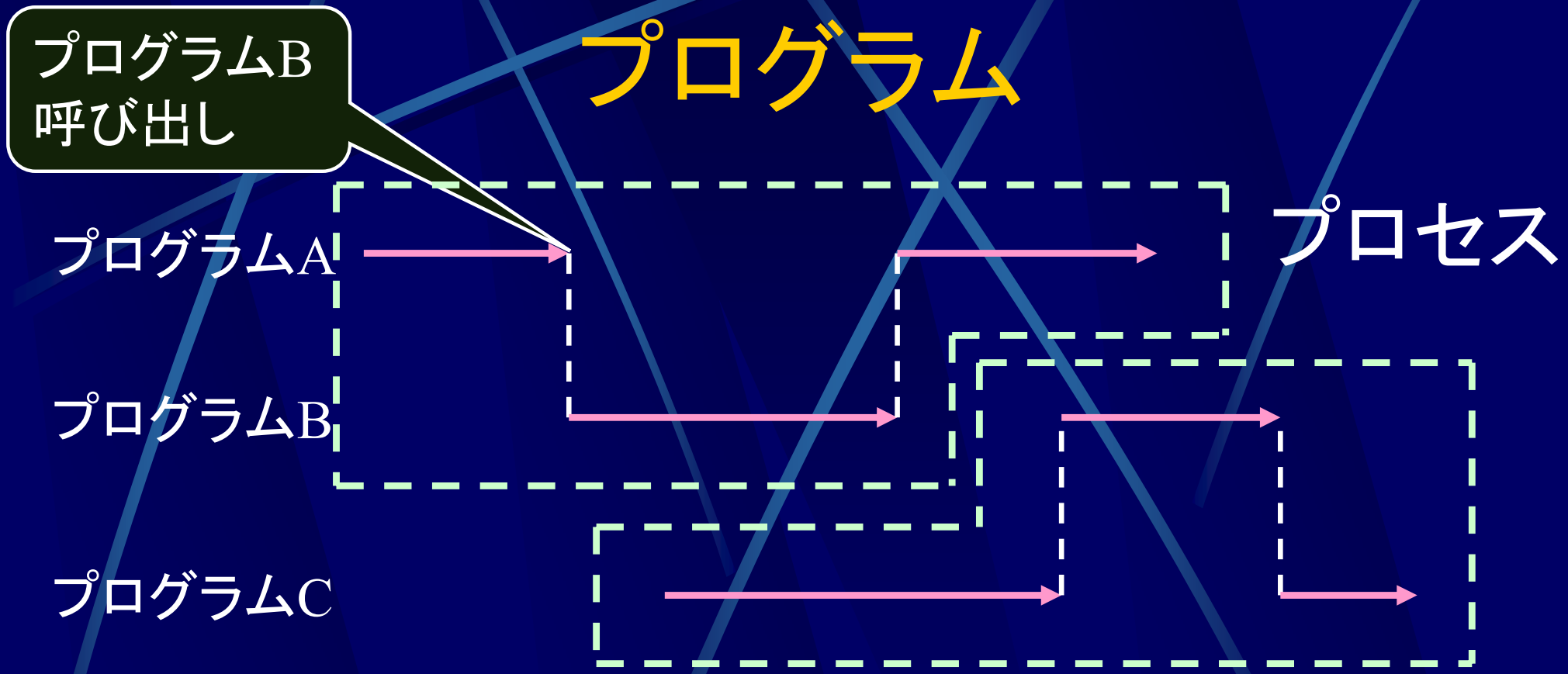
どの部分を
コピーするか？


親プロセス
子プロセス
子プロセス
コード領域
データ領域
スタック
全てコピーする
コード領域を
共有する
コード領域,
データ領域を
共有する

プログラム

- プログラム
  - プロセスの静的な実体
  - 以下のいずれかの属性を持つ
    - 再入可能(reentrant)
      - 再帰的再入可能, 非再帰的再入可能
    - 逐次再使用可能(serially reusable)
    - 再使用不能(nonreentrant)

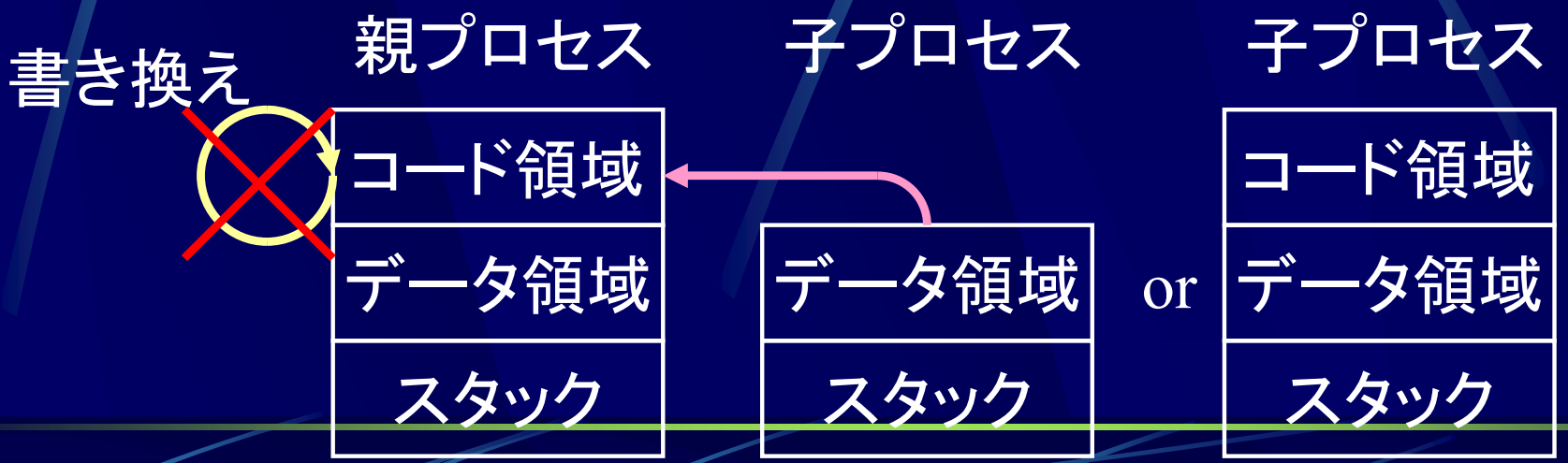
プログラム



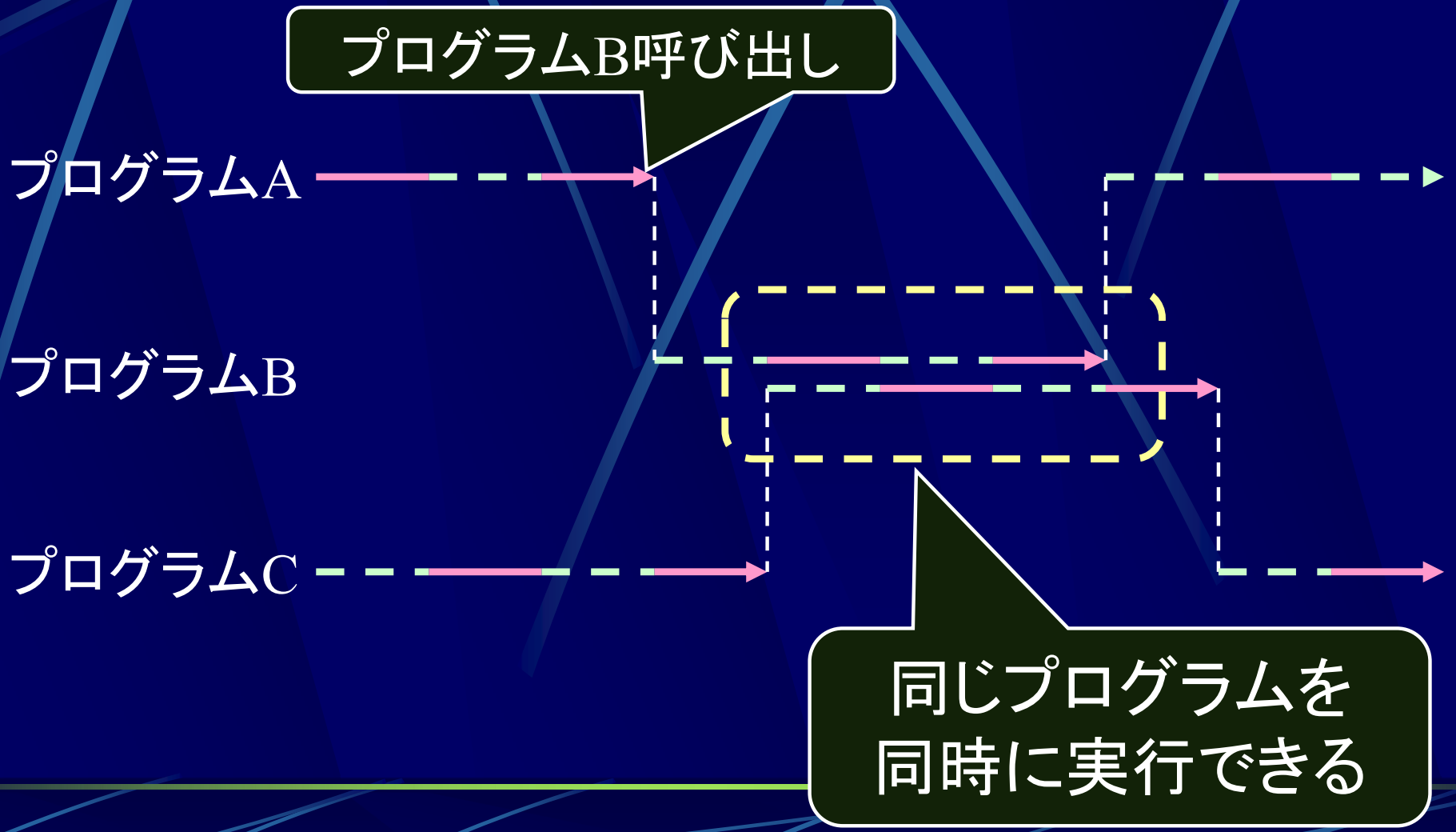
1つのプログラムは複数のプロセスで
使用される可能性がある

プログラムの属性
再入可能(reentrant)

- 複数のプロセスが同時に実行可能なプログラム
  - データ領域は独立
  - コード領域は独立、またはコード領域を書き換えない



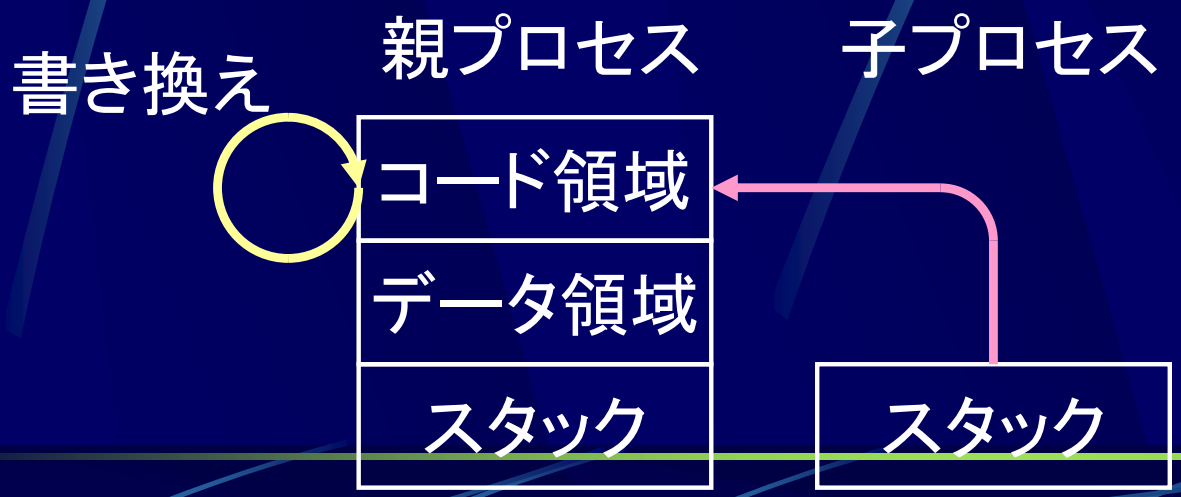
プログラムの属性
再入可能(reentrant)



プログラムの属性
再使用不能(nonreentrant)

- 1度しか実行できないプログラム
  - データ領域・コード領域が独立していない

かつ 実行するとデータ領域・コード領域が書き換わる

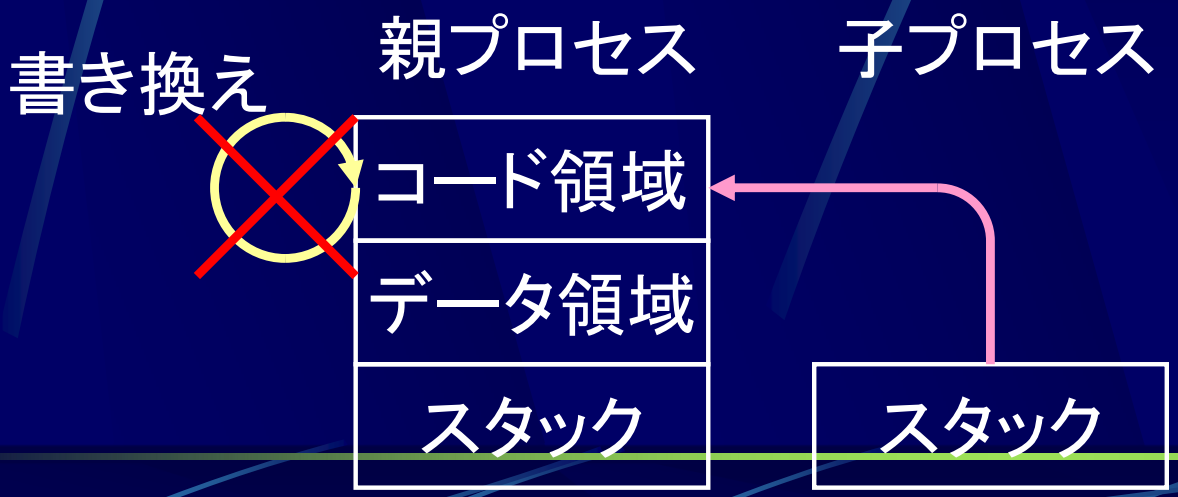


プログラムの属性
再使用不能(nonreentrant)

プログラムBは
1度実行すると
再使用できない

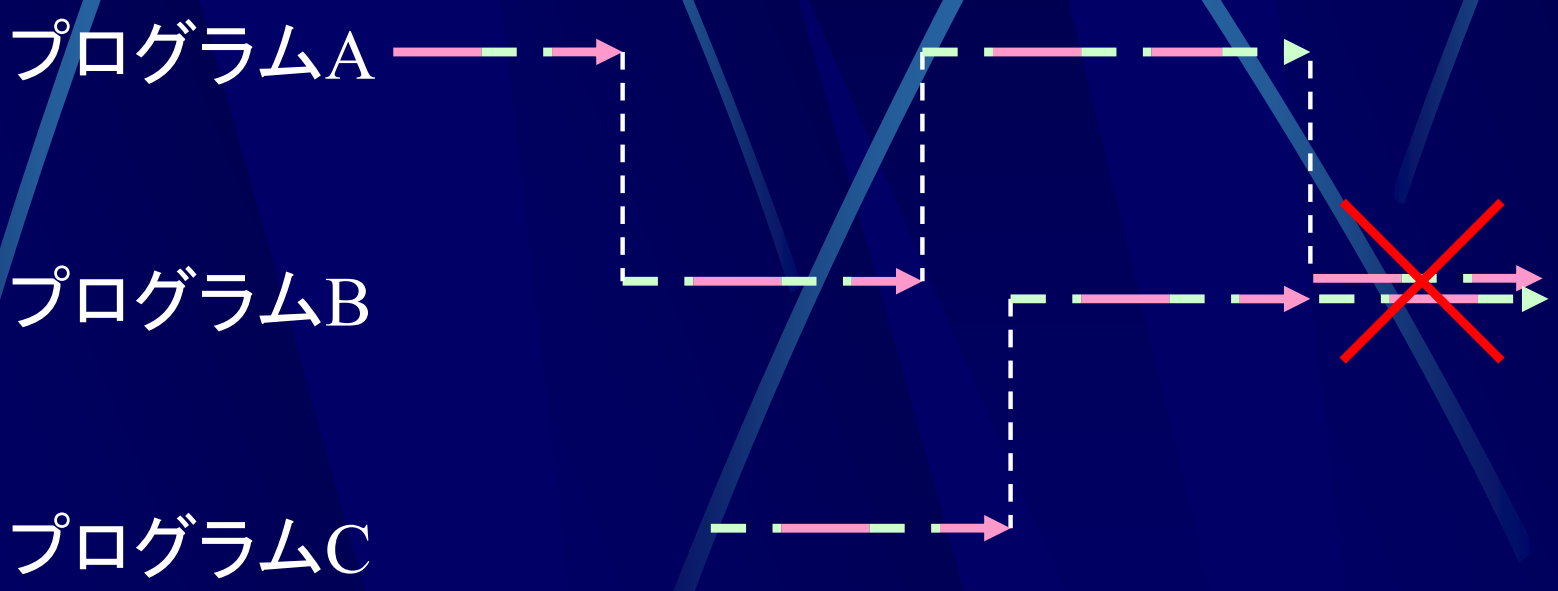
プログラムの属性
逐次再使用可能(serially reusable)

- 同時でなければ何度でも実行可能なプログラム
  - コード領域・データ領域は共通
  - コード領域・データ領域を書き換えない



プログラムの属性
逐次再使用可能(serially reusable)

同時使用は不可



同時でなければ
何度でも実行できる

プログラムの属性
再帰的再入可能(recursive)

自分自身を呼び出す
再帰呼び出しが可能

再入可能と再使用不能

```
func1 (int i) {
   int j;
      :
   j = i*10;
      :
```

動的変数

スタックに保存

```
func2 (int i) {
   static int j;
      :
   j = i*10;
      :
```

静的変数

データ領域に保存

データ領域に変更無し
⇒再入可能

データ領域を書き換え
⇒再使用不能

逐次的再使用可能と再使用不能

```
func3 (int i) {
   static int j=0;
        :
    j = j+i;
        :
```

初期値あり

以前の実行に依存しない
⇒逐次的再使用可能

```
func4 (int i) {
   static int j;
        :
    j = j+i;
        :
```

初期値無し

以前の実行に依存
⇒再使用不能

プログラムの属性

属性	プログラム書き換え	データ領域	データ依存関係	再帰呼び出し	同時呼び出し	複数回呼び出し
再帰的再入可能	不可	独立	独立	○	○	○
再入可能	不可	独立	独立		○	○
逐次再使用可能	不可	親と共通	依存しない			○
再使用不能	可	親と共通	依存する			

プロセスの操作

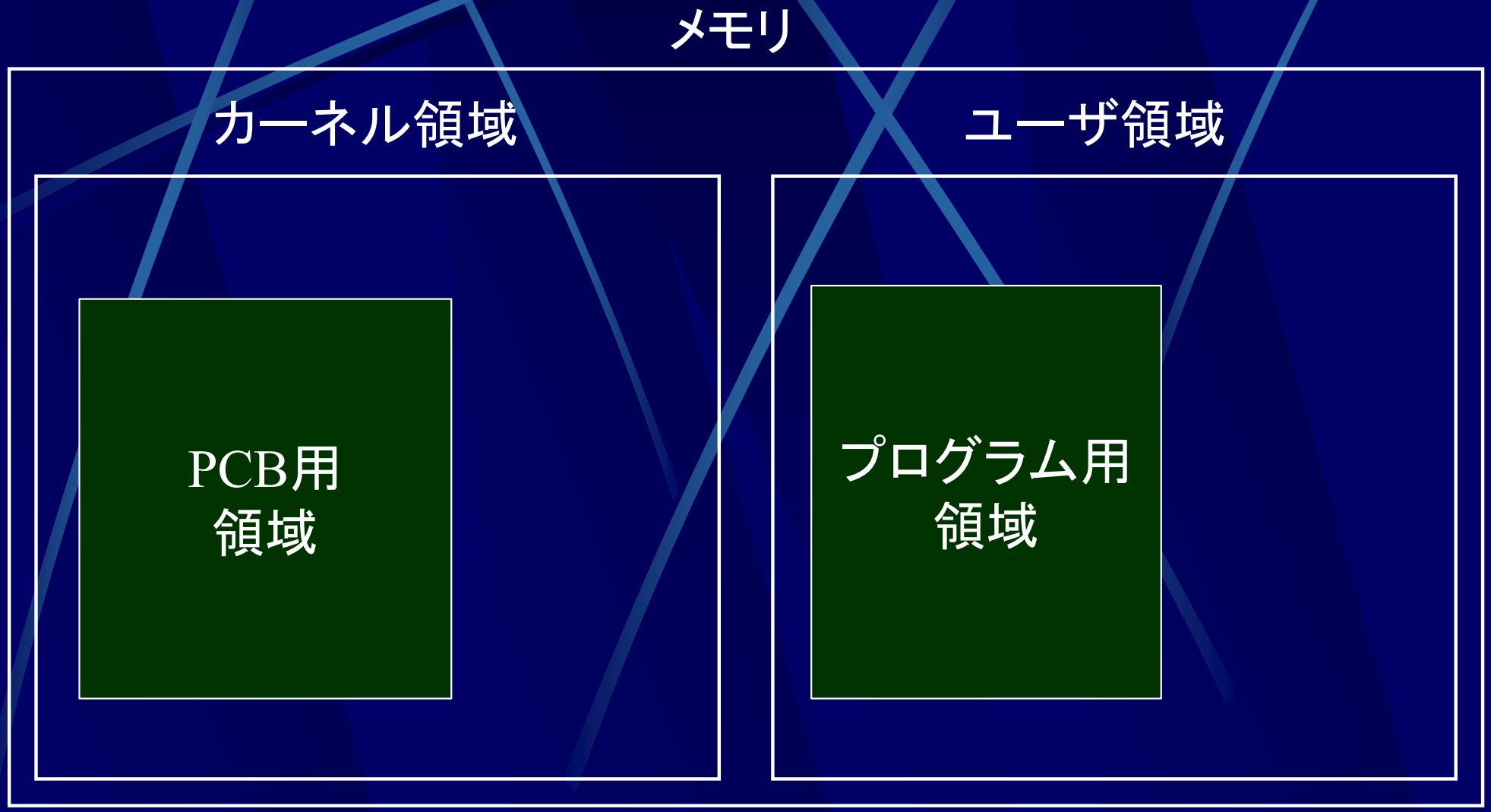
- プロセスに関する操作
  - 生成(create)
  - 消滅(destroy)
  - 中断(suspend)
  - 再開(resume)
  - 閉塞(block)
  - 起床(wakeup)
  - ディスパッチ(dispatch)
  - 優先度の変更

プロセスの生成(create)

- プロセス生成

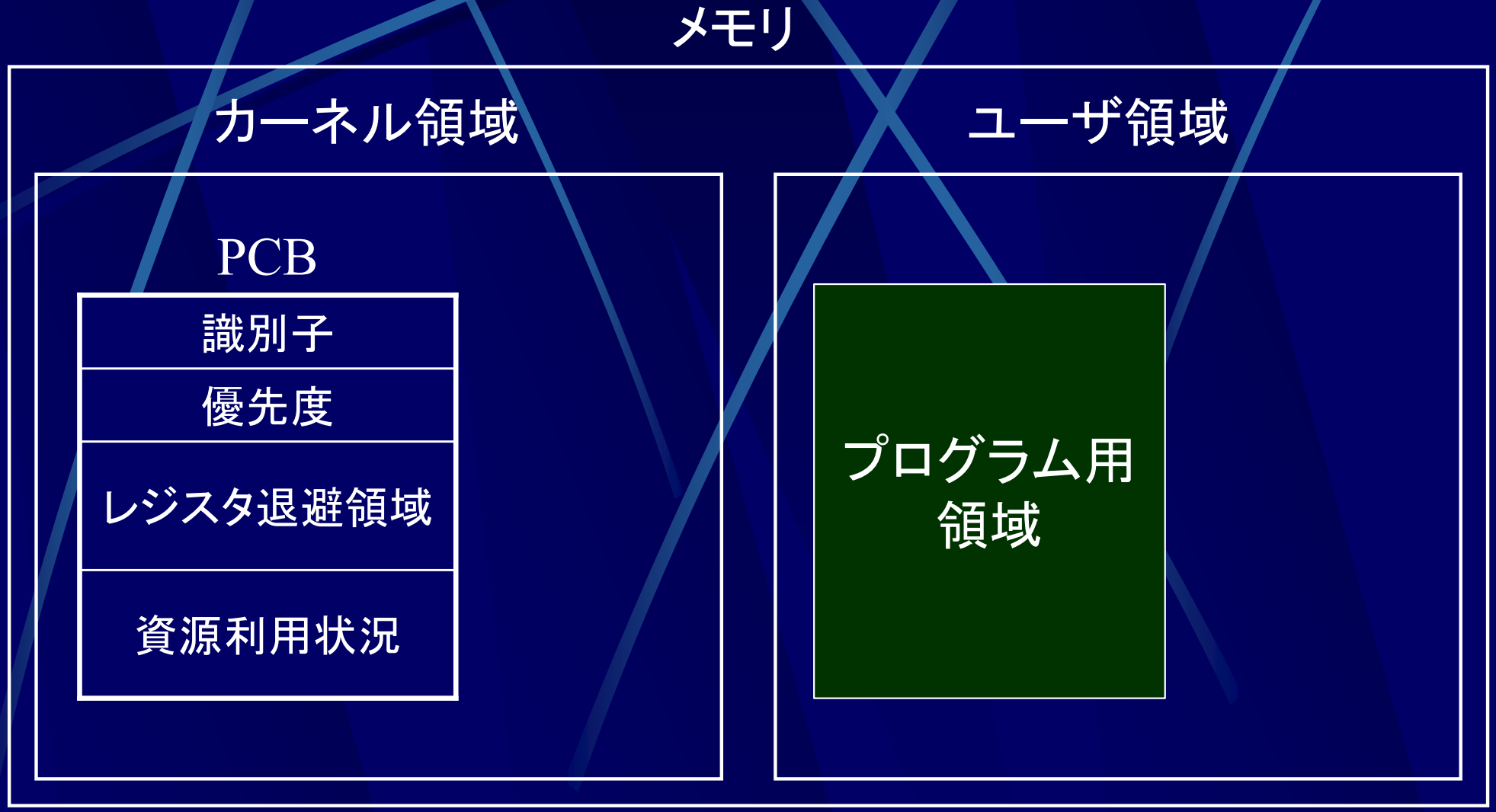
1. PCB領域の確保(カーネル領域)
2. コード, データ領域等の確保(ユーザ領域)
3. 名前付け,優先度決定
4. 資源割付
5. PCBの設定
6. PCBを実行可能キューへ

プロセスの生成



メモリのカーネル領域, ユーザ領域に必要な分を確保する

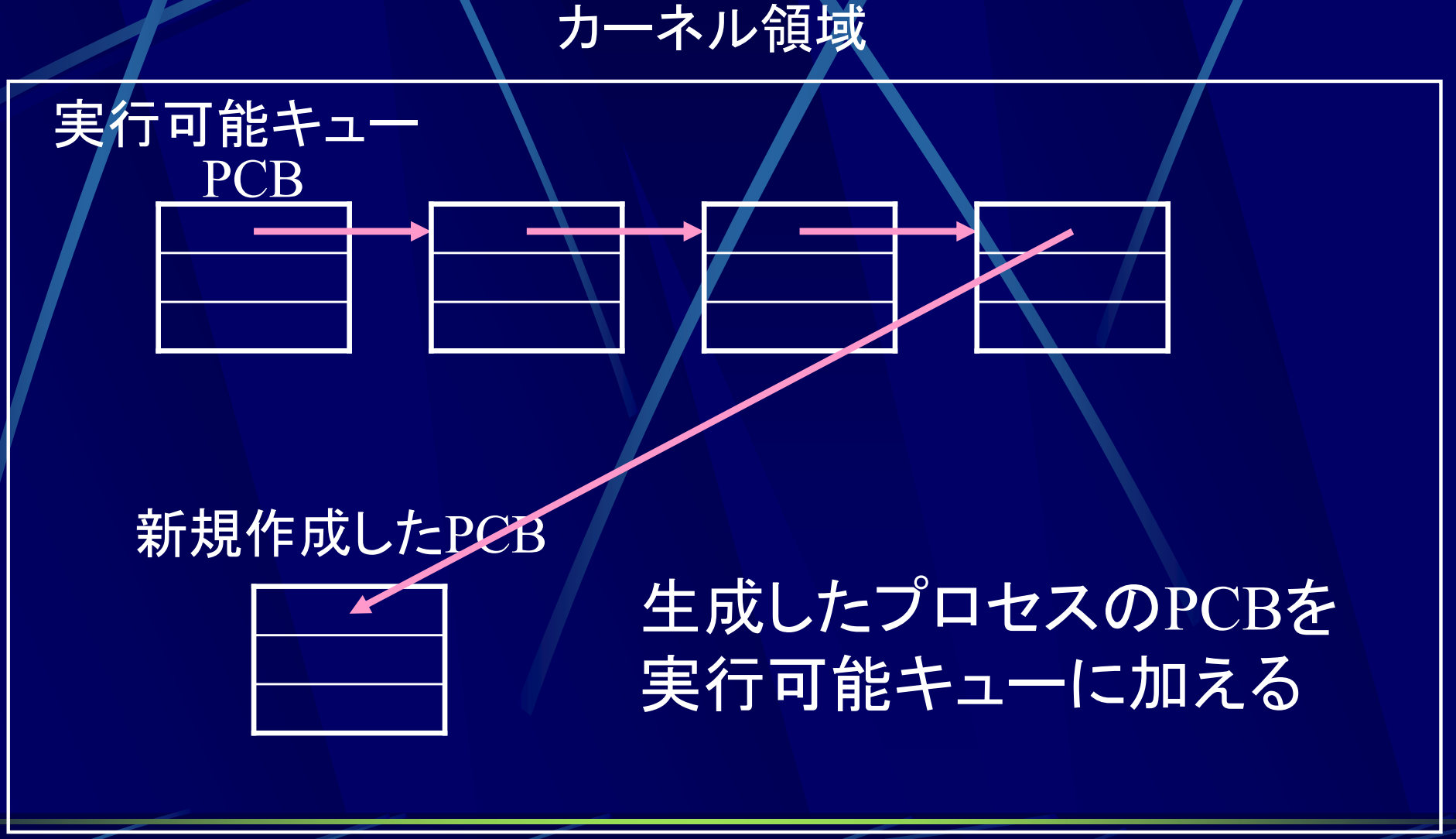
プロセスの生成



メモリのカーネル領域, ユーザ領域に必要な分を確保する

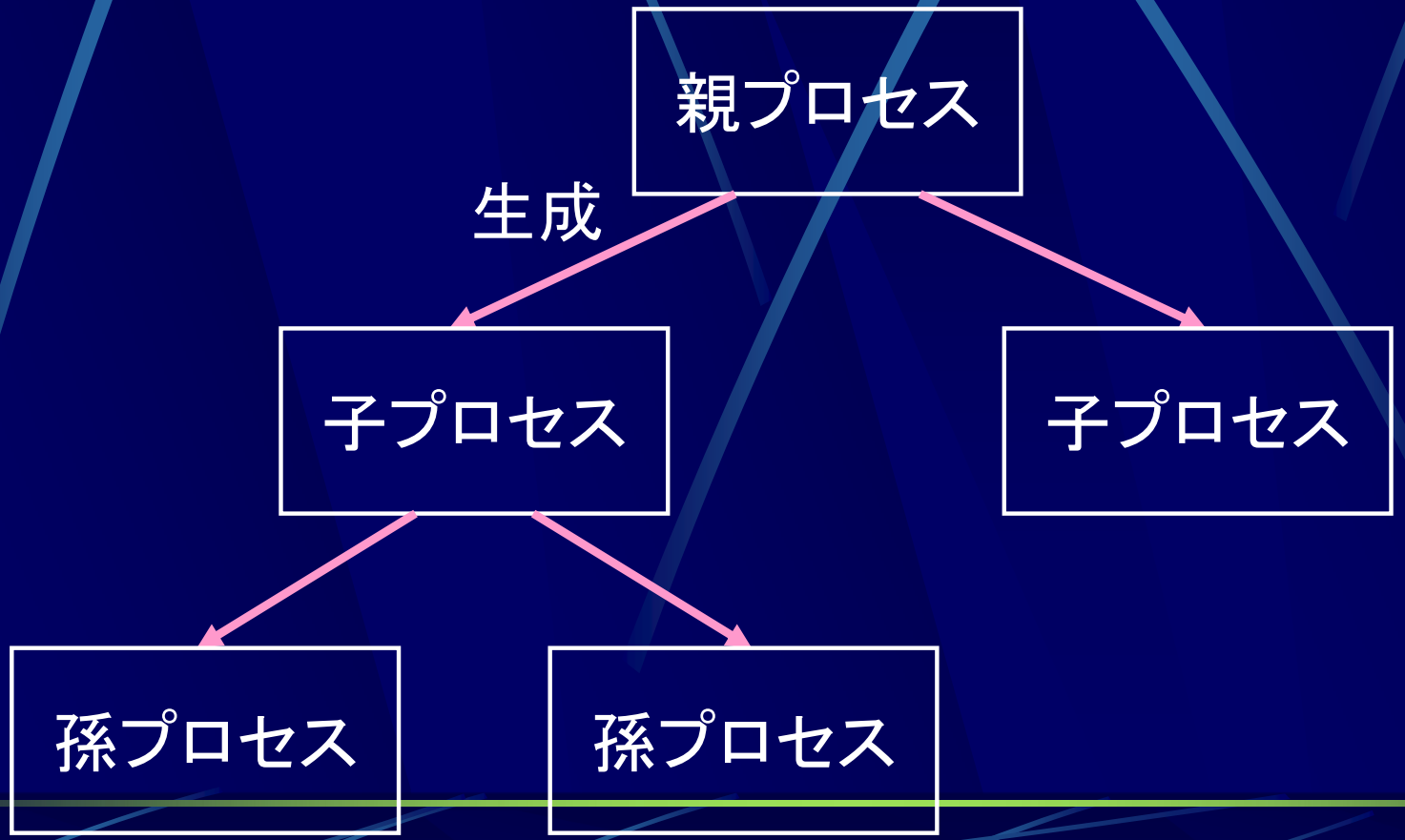
PCBを設定する

プロセスの生成



プロセスの階層

- プロセスの生成は階層的に行われる

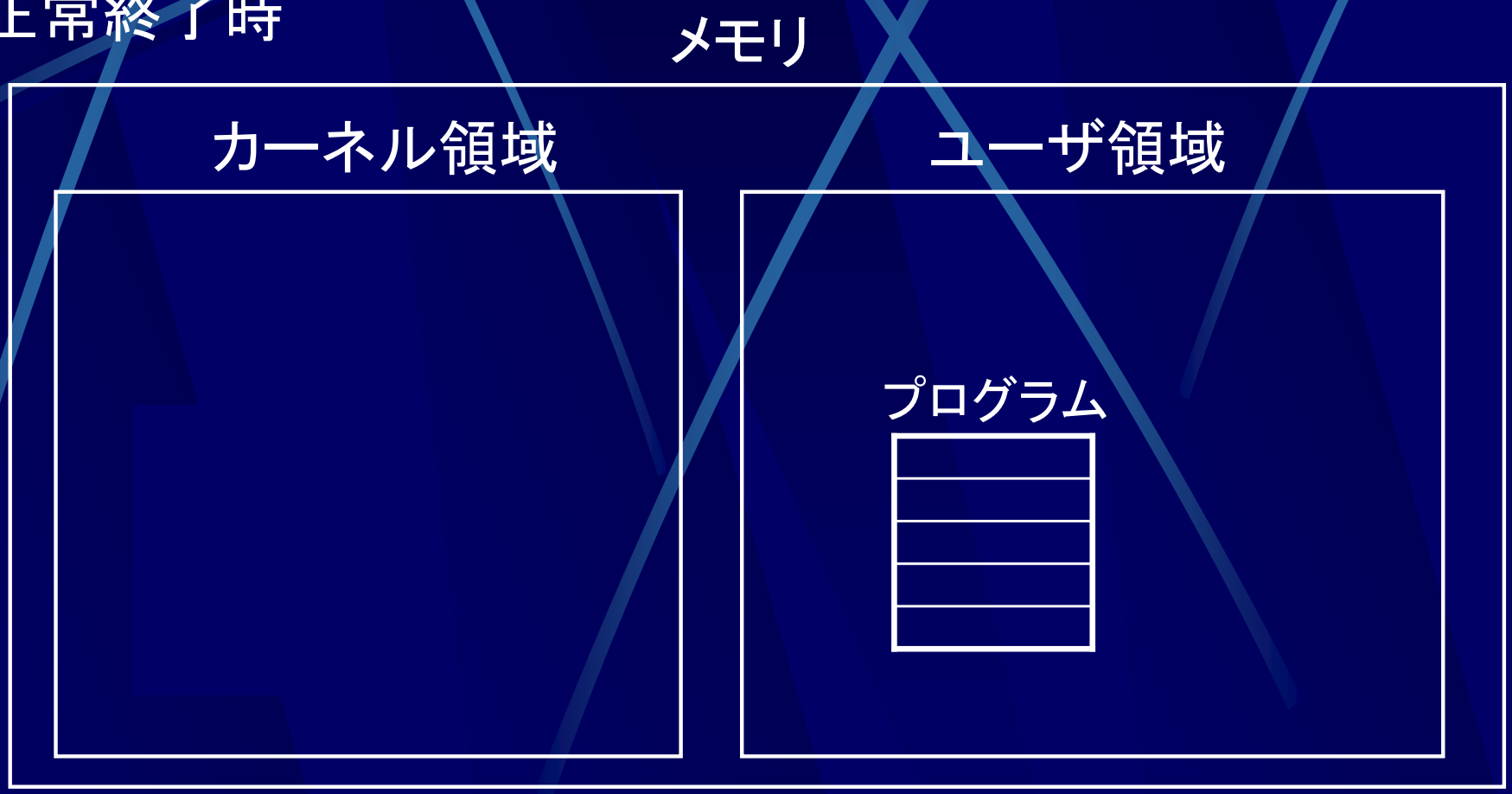


プロセスの消滅(destroy)

- プロセスの消滅
  - 正常終了
    - PCB領域の解放
  - 異常終了
    - 種々の後始末、コアダンプ等が必要

プロセスの消滅

正常終了時

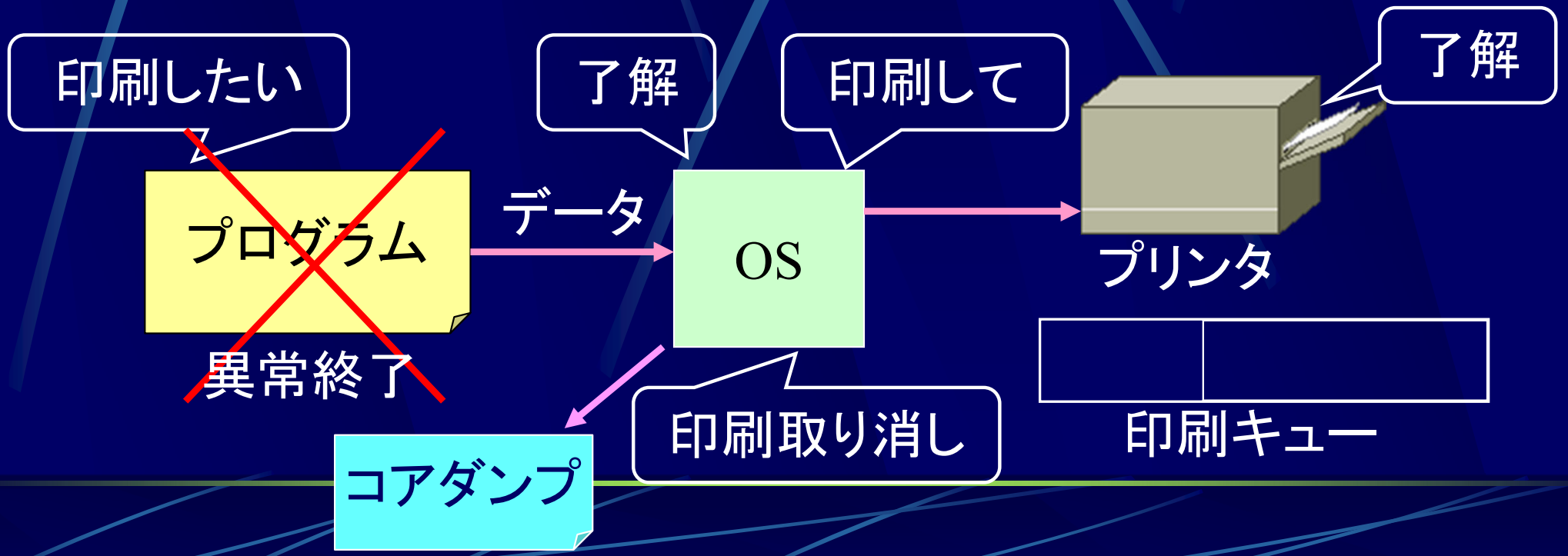


正常終了時はカーネル領域からPCBを削除する

(ユーザ領域のプログラムはそのまま)

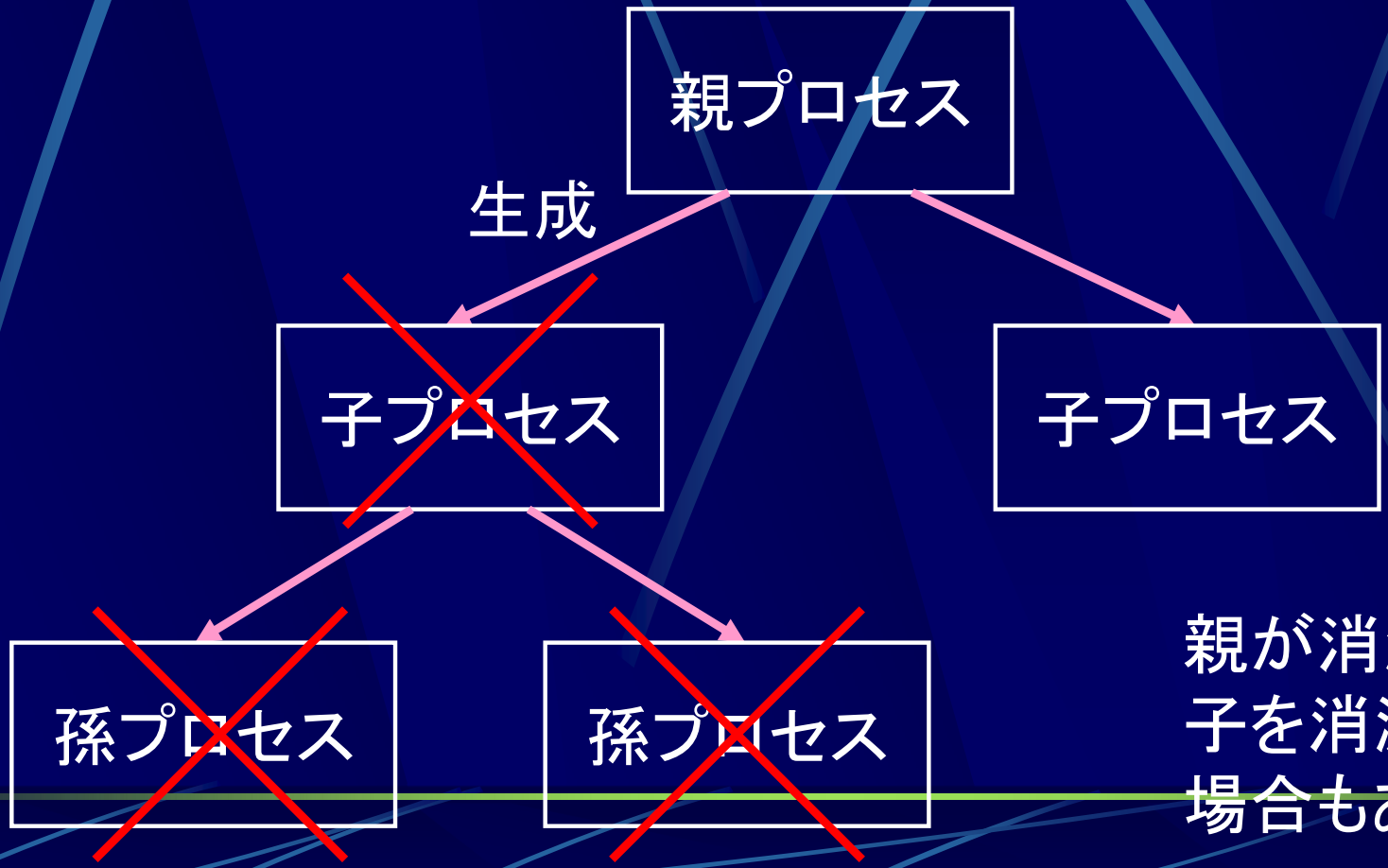
プロセスの消滅

- 異常終了時
  1. 使用中の資源をシステムに返却
     - 資源に対応した各種キュー, テーブルから削除
  2. コアダンプ(メモリ情報)出力
  3. PCBをシステムに返却



階層構造プロセスの消滅

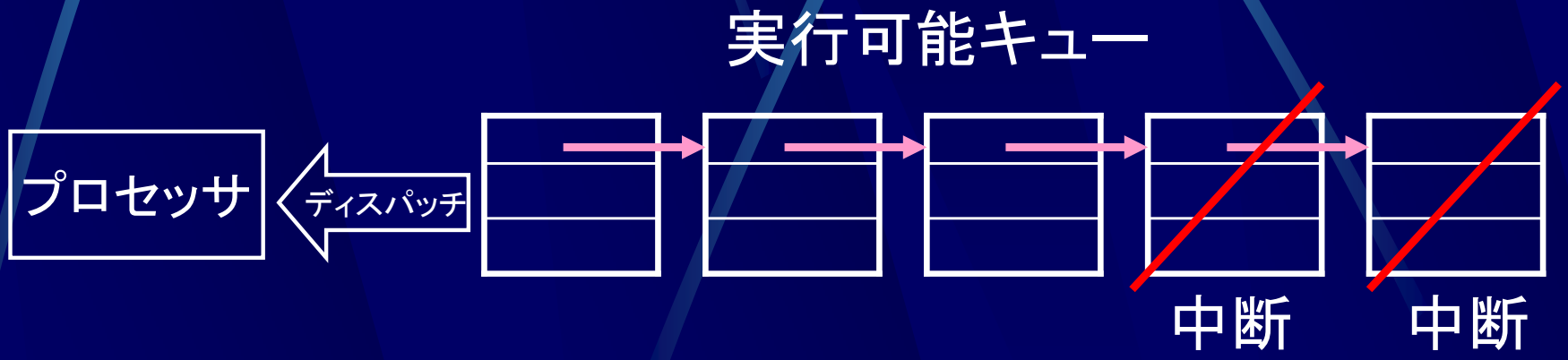
- 親プロセスが消滅したときに子プロセスも消滅



親が消滅しても
子を消滅させない
場合もある

プロセスの中断，再開
(suspend, resume)

- プロセスの中断(suspend)
  - システムの負荷が高くなったときに一時的に特定のプロセスを実行可能状態から除く



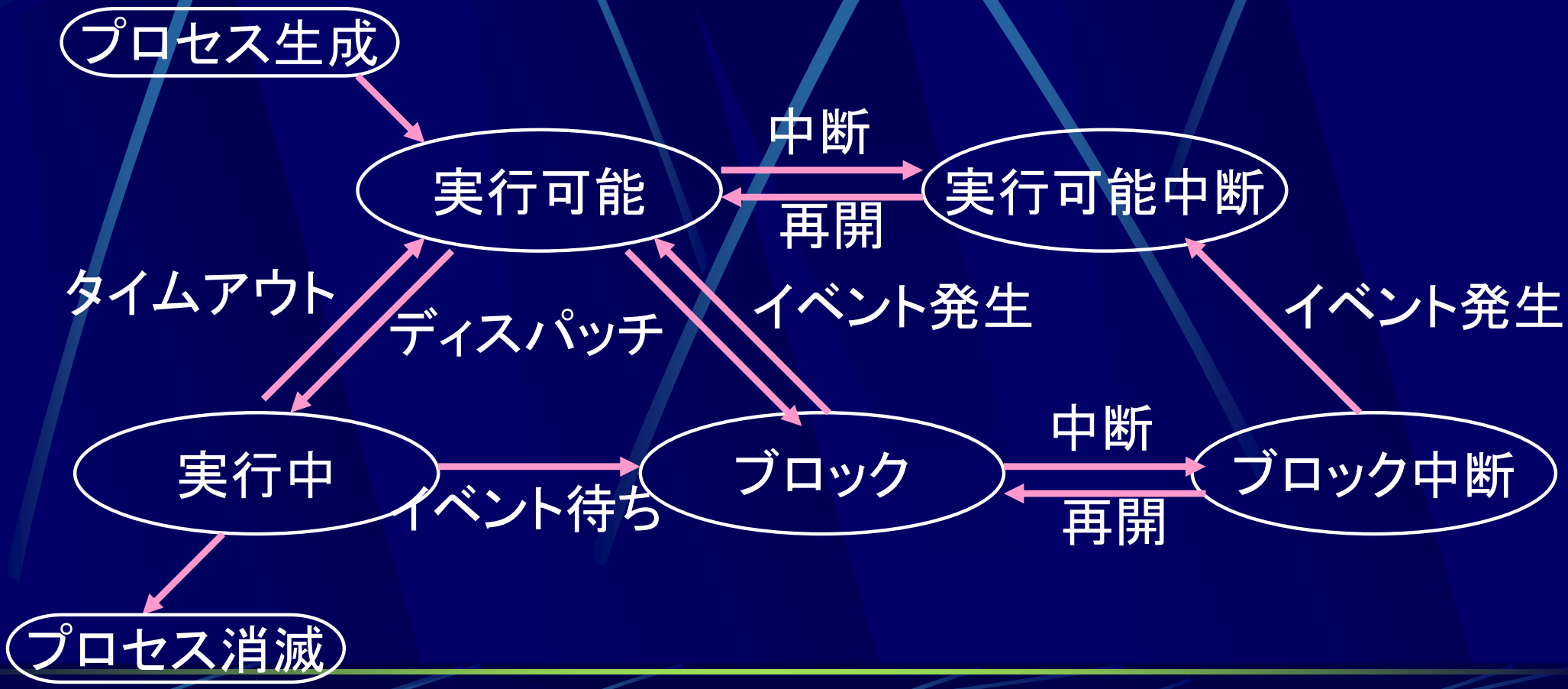
実行可能プロセスが多すぎるとシステムに負荷がかかる

いくつかのプロセスを中断する

プロセスの中断

- システムが高負荷
  - システムの負荷が十分小さくなるまで中断
- システム障害発生時
  - 障害回復するまで中断
- デバグ時
  - プロセスが正しく働いているかをユーザが確認するまで中断

プロセスの状態遷移


プロセス生成
実行可能
中断
再開
実行可能中断
タイムアウト
ディスパッチ
イベント発生
イベント発生
実行中
イベント待ち
ブロック
中断
再開
ブロック中断
プロセス消滅

プロセスの重量化

計算機システムの高性能化
(仮想記憶, ネットワーク機能, 並列処理等)

プロセス処理のオーバヘッドの肥大化

プロセスの重量化

プロセス実行を高速化する必要

プロセス実行の高速化

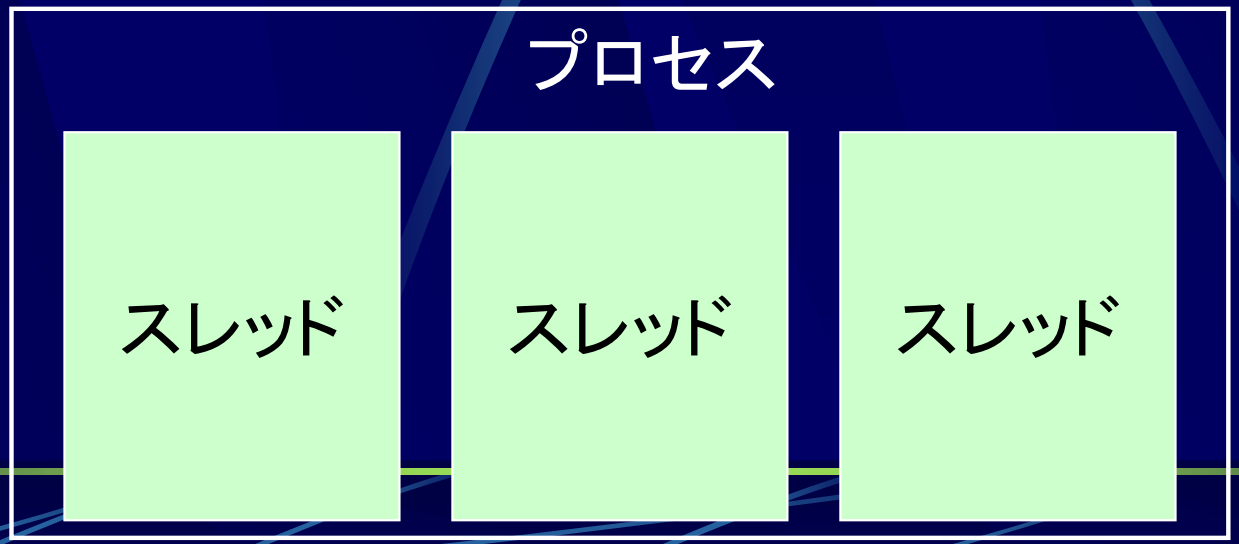
- プロセス実行の高速化
  - プロセス自体の高速化
  - プロセス生成, 消滅, 切り替え等の高速化

この部分を“軽く”する

スレッド(thread)を用いる

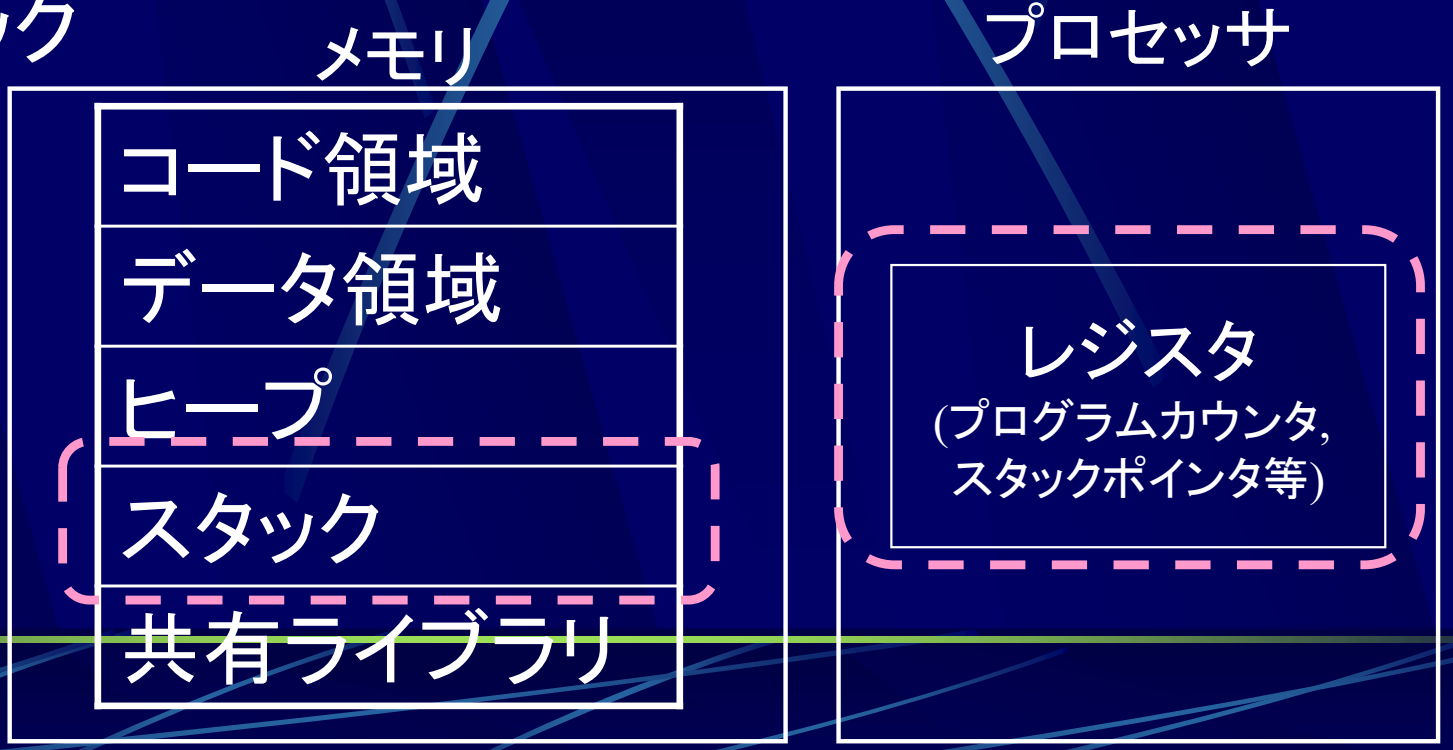
スレッド(thread)

- スレッド(thread)
  軽量プロセス(light weight process)
  - プロセスの拡張
  - 制御の流れ, プロセス中で実行できる実体

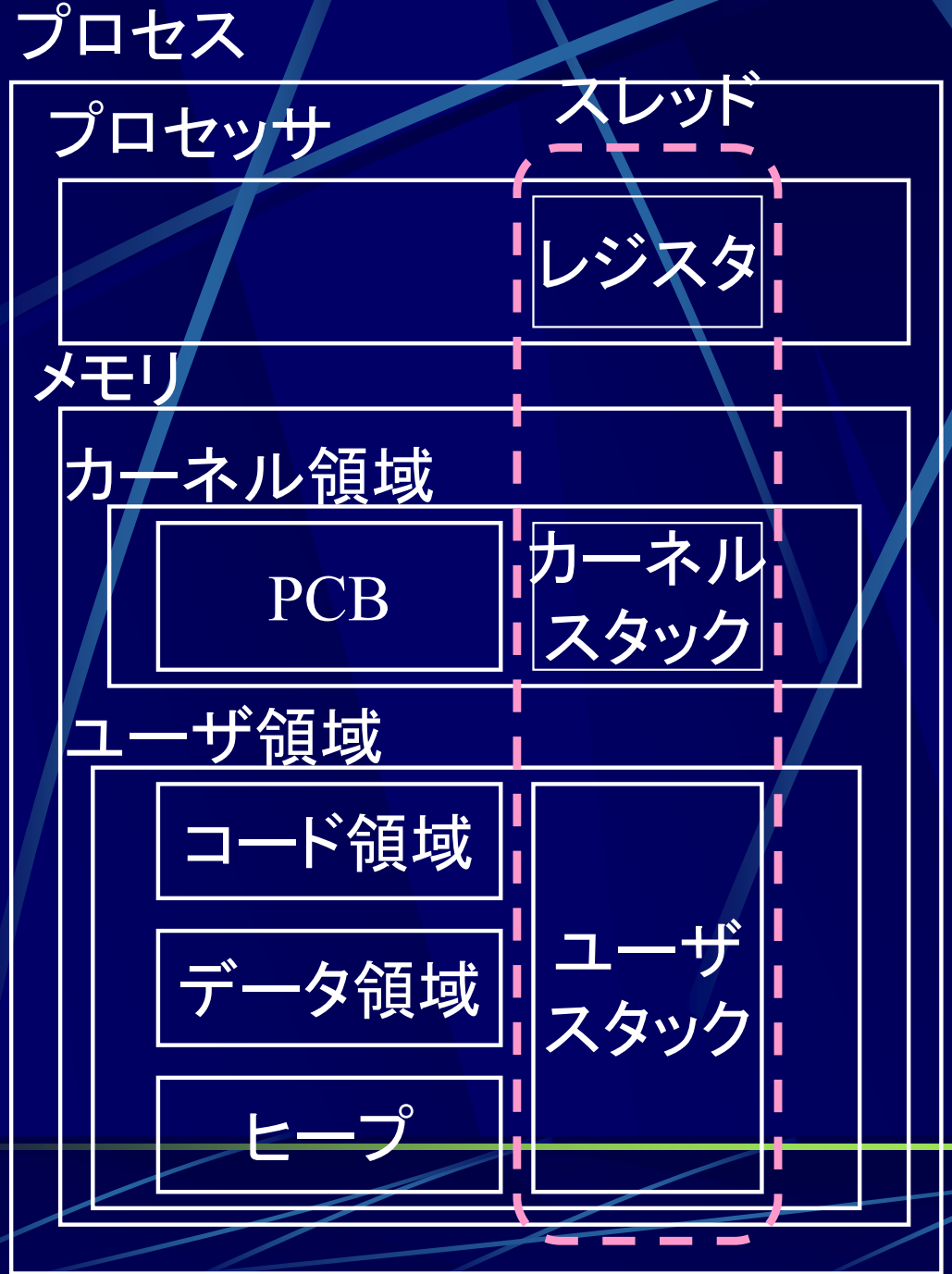


スレッド

- スレッドの要素
  - レジスタセット
    - (プログラムカウンタ, スタックポインタ等)
  - スタック



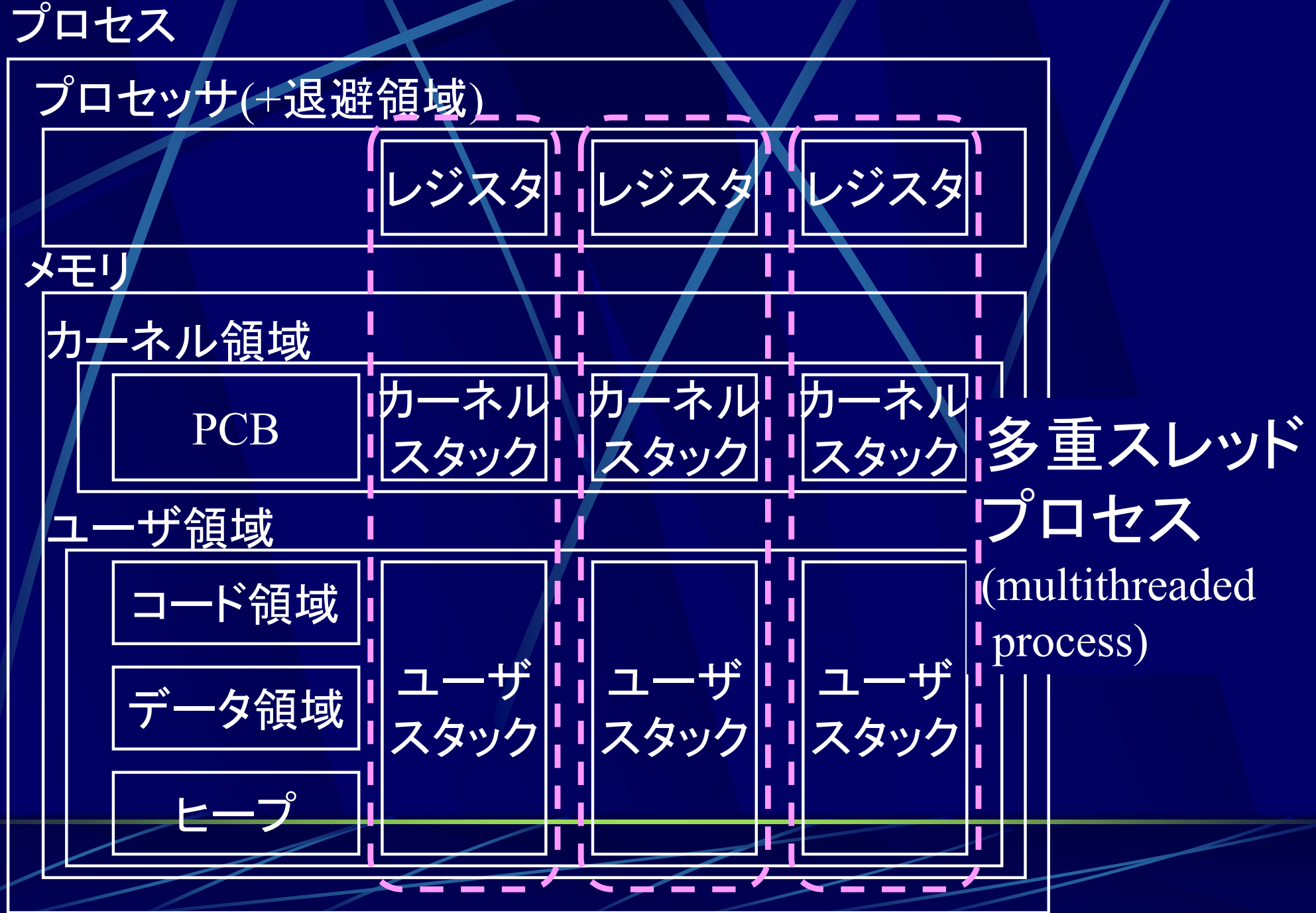
単一スレッド



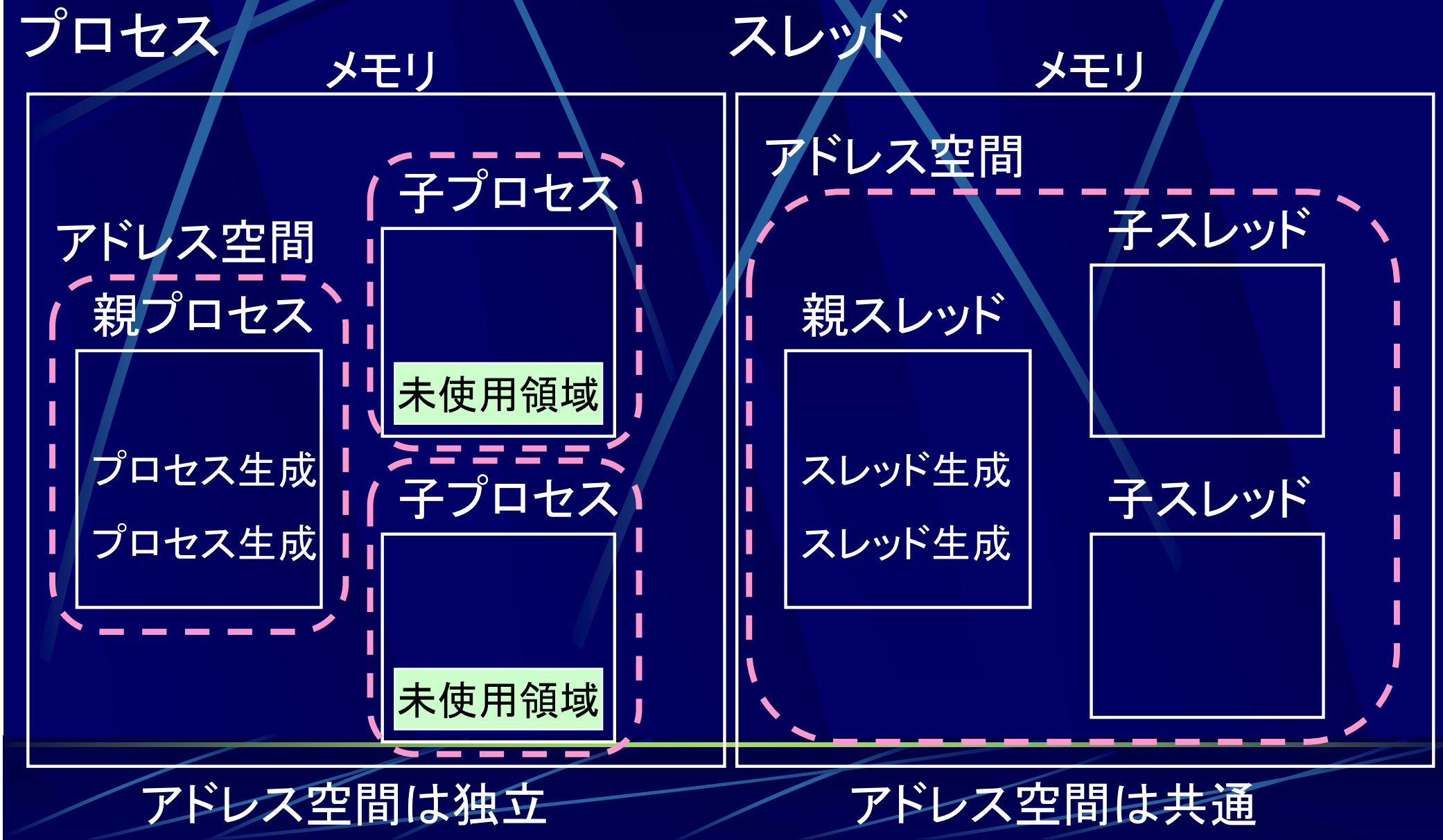
スレッドの要素
レジスタセット, スタック

単一スレッドプロセス
(single threaded process)
1プロセス中に1スレッド

多重スレッド


プロセス
プロセッサ(+退避領域)
レジスタ
レジスタ
レジスタ
メモリ
カーネル領域
PCB
カーネル
スタック
カーネル
スタック
カーネル
スタック
ユーザ領域
コード領域
データ領域
ヒープ
ユーザ
スタック
ユーザ
スタック
ユーザ
スタック
多重スレッド
プロセス
(multithreaded
process)

プロセスとスレッド


プロセス
メモリ
アドレス空間
親プロセス
プロセス生成
プロセス生成
子プロセス
未使用領域
子プロセス
未使用領域
アドレス空間は独立
スレッド
メモリ
アドレス空間
親スレッド
スレッド生成
スレッド生成
子スレッド
子スレッド
アドレス空間は共通

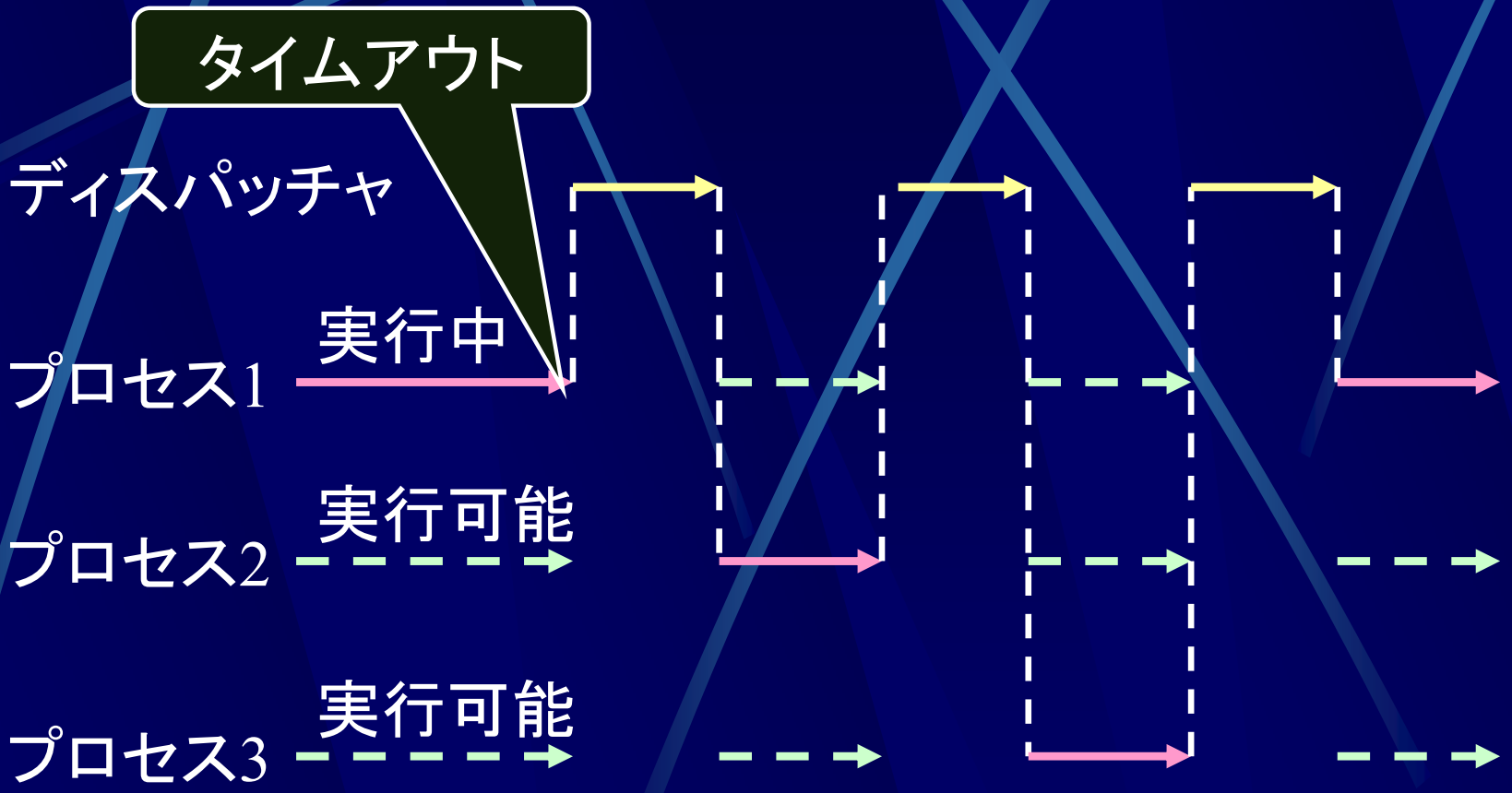
プロセスとスレッド

	プロセス	スレッド
コード領域	独立 or 共有	共有
データ領域	独立 or 共有	共有
スタック	独立	独立
計算機資源	独立	共有

コード領域・データ領域・計算機資源が共有
⇒ 切り替えの手間が少ない

コード領域が共通
⇒ 同一のプログラムしか使えない

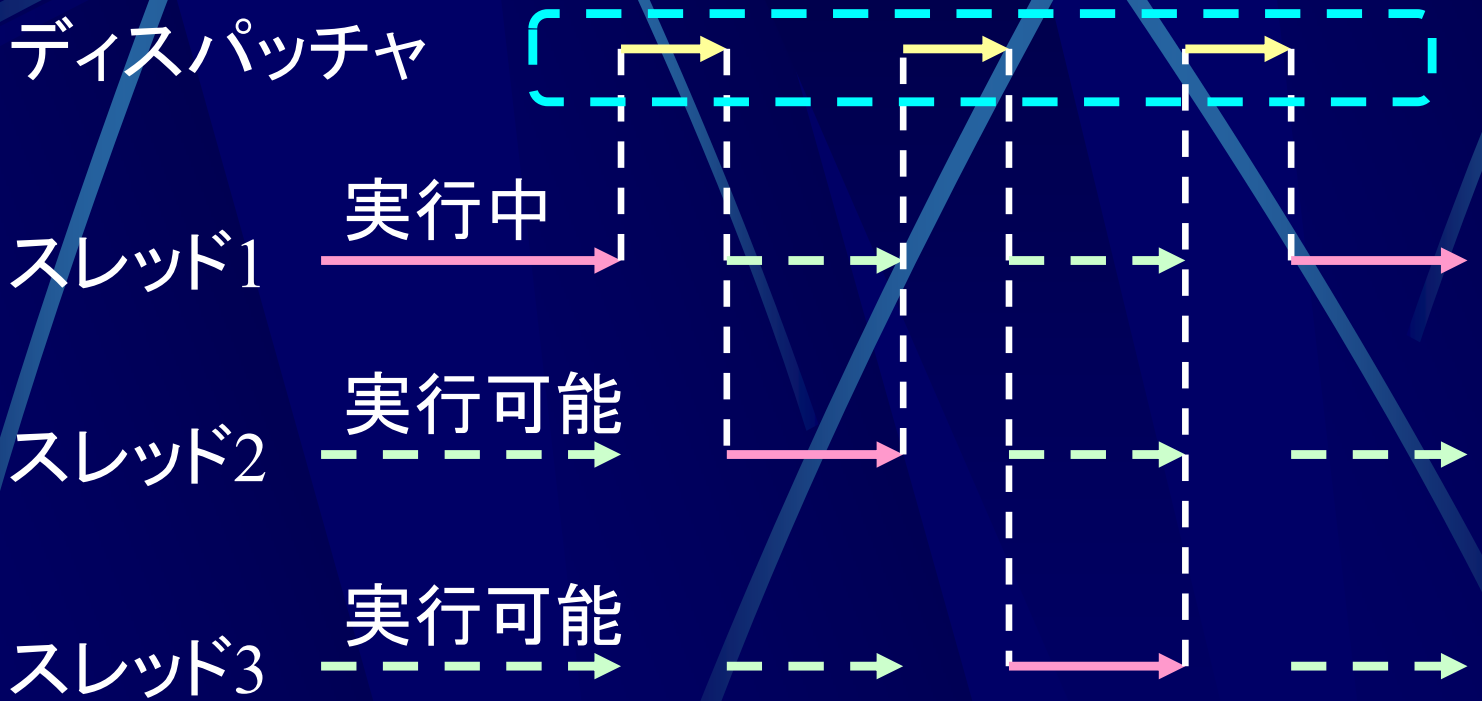
プロセスの状態遷移



ディスパッチャによるプロセス切り替え時間が
実行時間全体の無視できない割合を占める

(およそ6%)

スレッドの状態遷移



切り替えにかかる時間が
プロセスよりも短い

スレッドの実現法

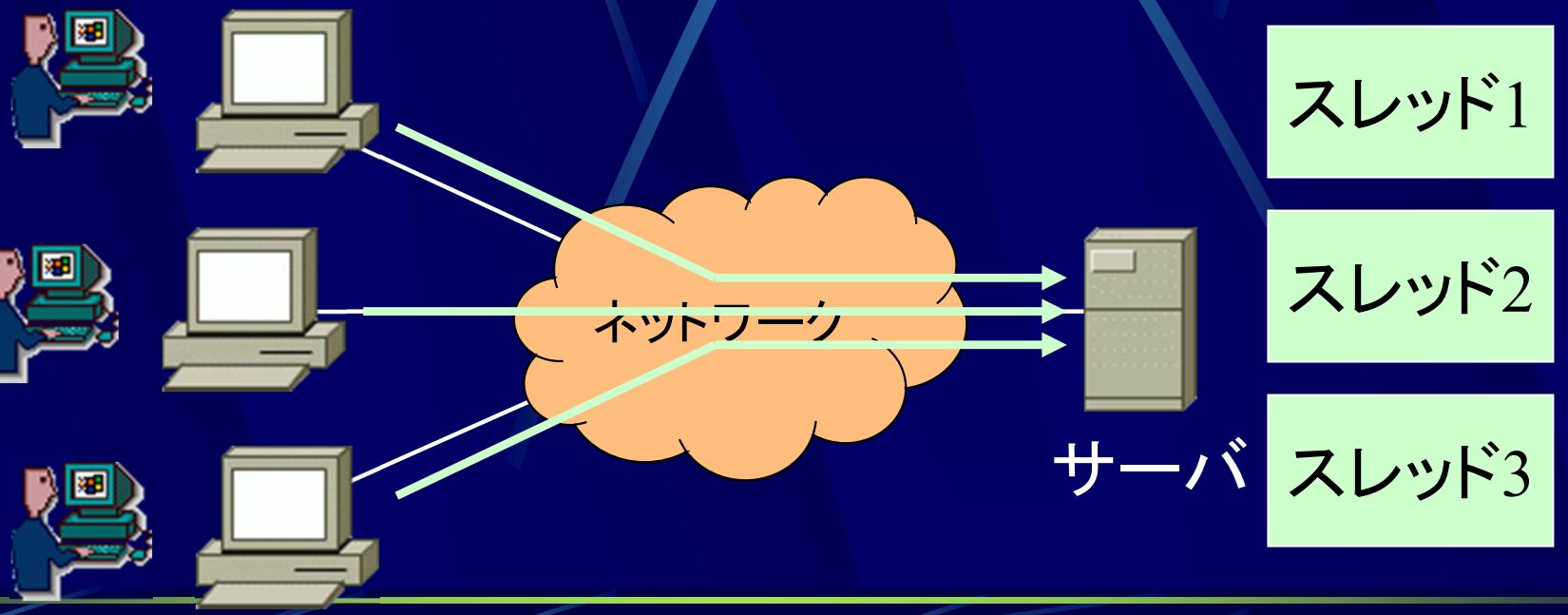
- カーネルレベルによる実現
  - カーネルがスレッドを管理
  - システムコールプリミティブを利用(プロセスと同様)
  - オーバヘッドが大きく重い
- ユーザレベルによる実現
  - コルーチンを使用してユーザが管理
  - スレッドライブラリプリミティブを利用
  - スレッドを軽くでき、多数のスレッドを作れる
  - スケジューリングをユーザが制御可能
  - 横取り、入出力の独立が難しい

スレッドの利用

- サーバプログラムの応答性の向上
- CPU処理と入出力処理のオーバラップ
- 並列アルゴリズムの実現

スレッドの利用

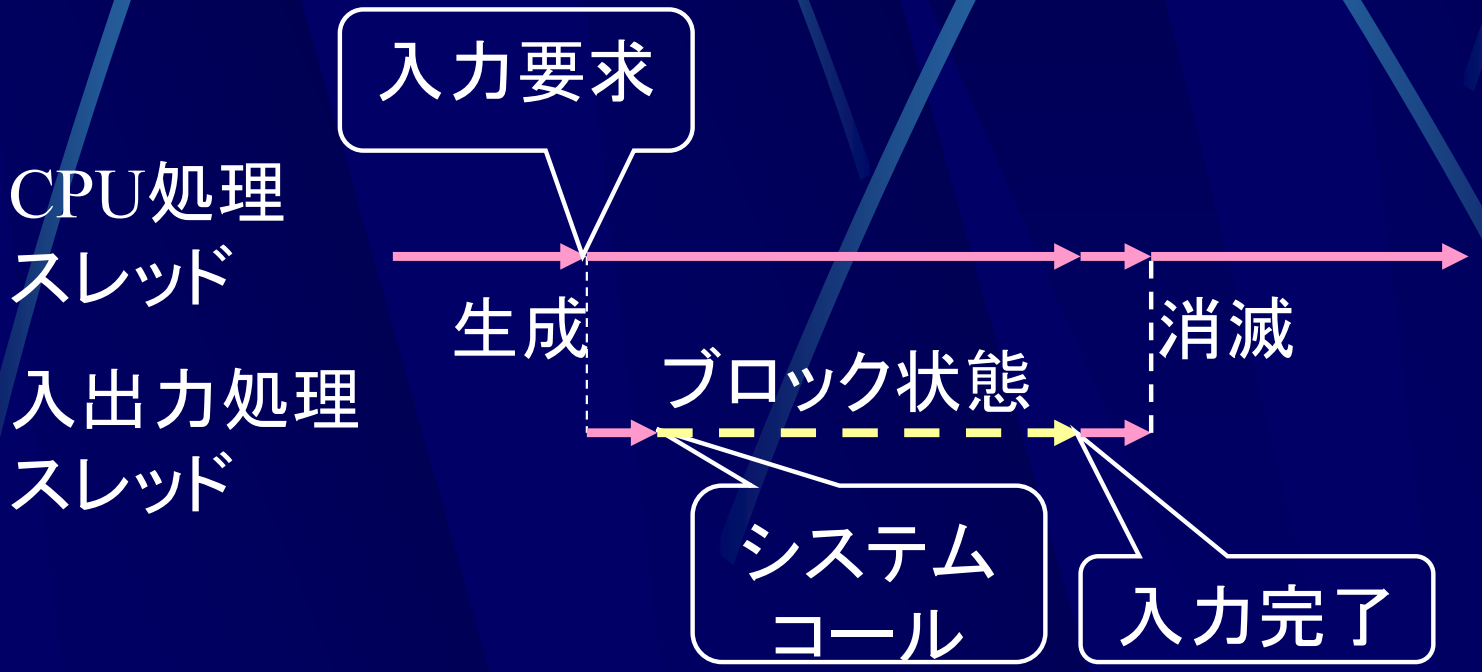
- サーバプログラムの応答性の向上
  - サーバプログラムを多重スレッド化



同時に複数のユーザがサーバを利用できる

スレッドの利用

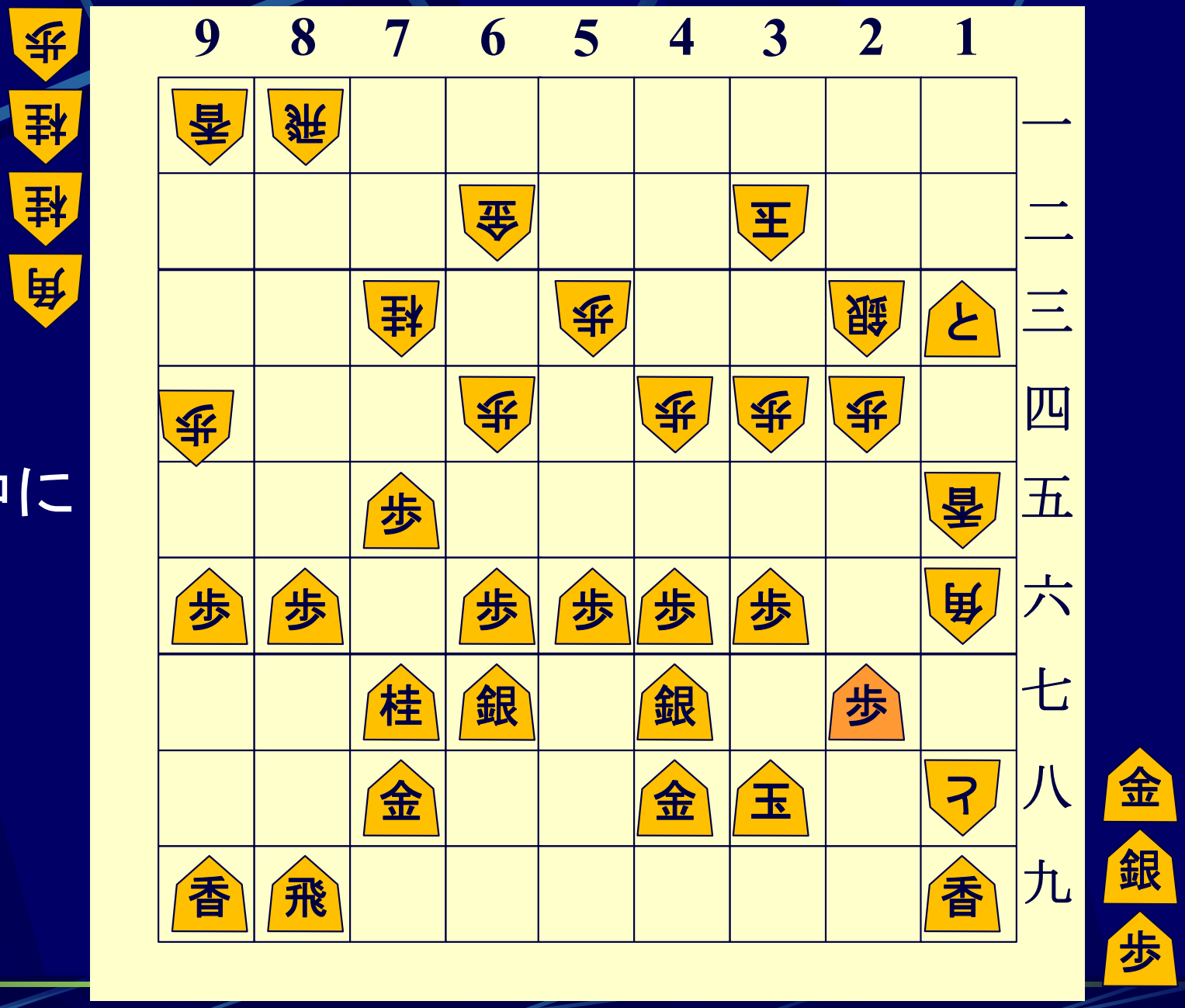
- CPU処理と入出力のオーバラップ
  - CPU処理と入出力処理を別のスレッドで実行



入力待ちの間もCPU処理をすることができる

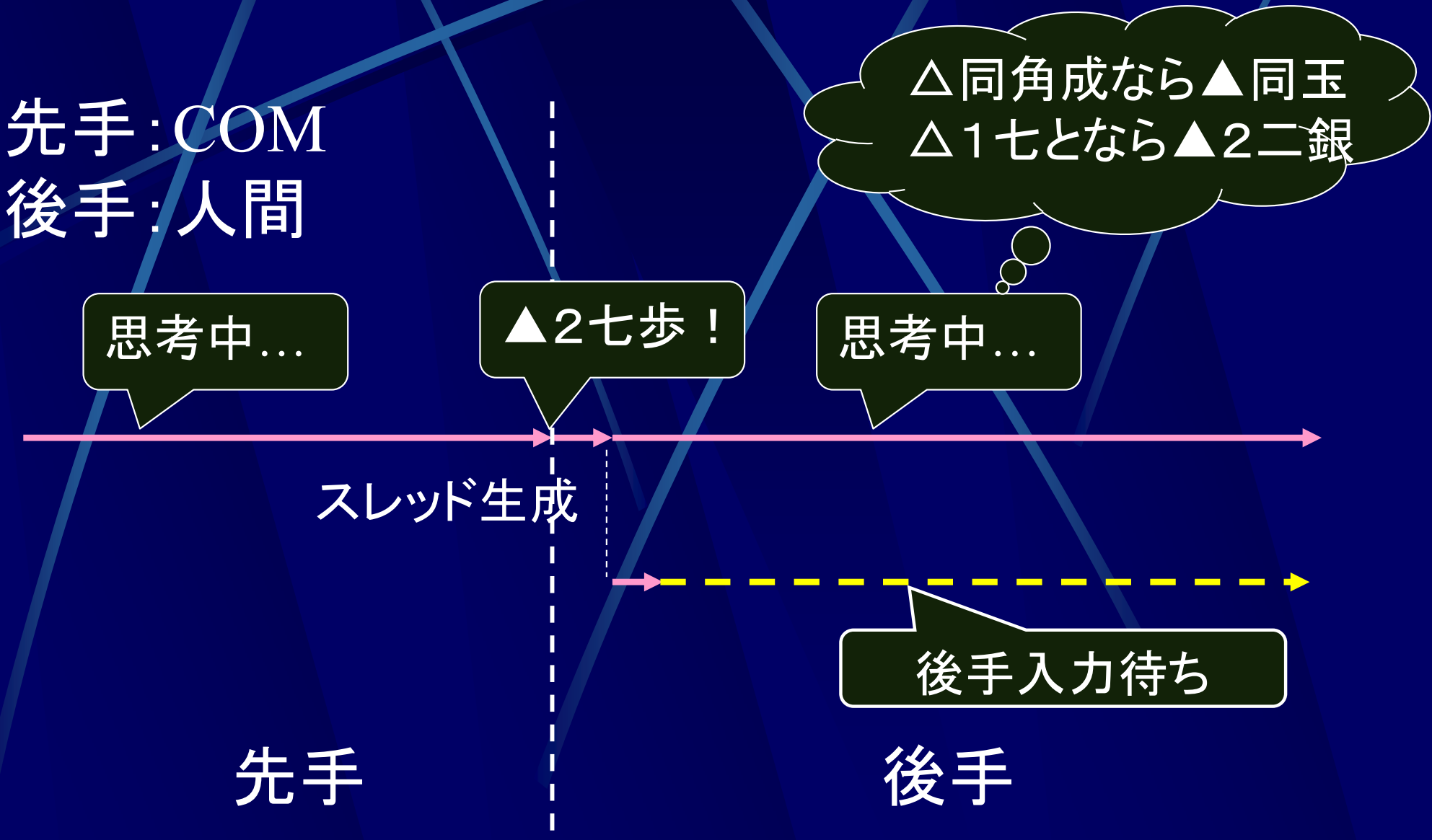
スレッドの利用

将棋AI
相手の思考中に
自分も考える



▲2七歩まで

スレッドの利用



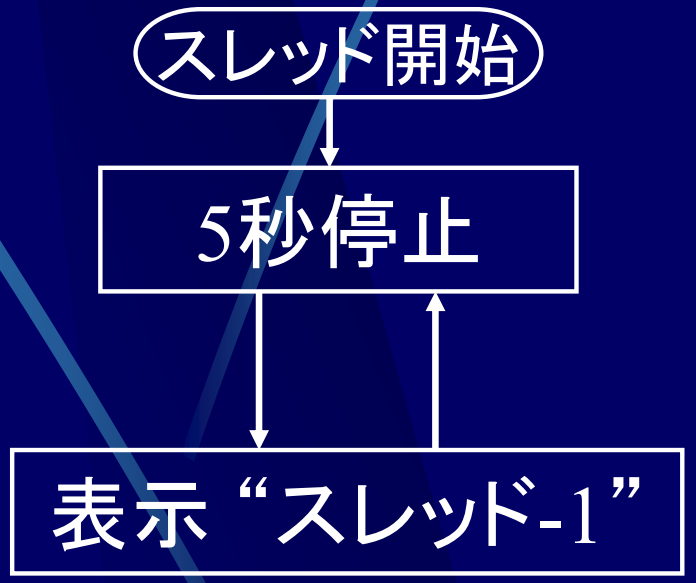
後手が考えている間に先手も考えられる

スレッドのメリットとデメリット

- スレッドのメリット
  - 切り替え時間が短い
  - 切り替えをユーザが制御できる
  - 必要なメモリが少ない
- スレッドのデメリット
  - 同一のプログラムしか使えない
  - 切り替えをユーザが制御しなければならない

参考：スレッドプログラム(java)

- StartThread.java
  - 以下の動作を繰り返す
    1. 引数で指定した時間停止
    2. スレッド名の表示
- ThreadCreate.java
  - 停止時間が5秒, 7秒, 4秒の3つのスレッドを生成, 実行する



スレッド1の実行

http://www.info.kindai.ac.jp/OS
からダウンロードし、各自実行してみること

参考：スレッドプログラム(java)

StartThread.java(前半)

```
class StartThread extends Thread { /* Threadクラスを拡張 */
   int threadNum;          /* スレッド番号 */
   long latency;           /* 停止する時間(ミリ秒) */

   /* コンストラクタ */
   StartThread (int threadNum, long latency) {
      this.threadNum = threadNum;
      this.latency = latency;
   }
```

後半に続く

参考：スレッドプログラム(java)

StartThread.java(後半)

```
  public void run() {
     while (true) { /* 永久に繰り返す */
        try {
            sleep (latency); /* 指定したミリ秒の間停止 */
        } catch (InterruptedException error_report) {
            System.out.println (error_report);
            System.exit (1);
        }
        System.out.println (threadNum); /* スレッド番号表示 */
     }
  }
}
```

参考：スレッドプログラム(java)

ThreadCreation.java

```
class ThreadCreation {
   public static void main (String[] args) {
      Thread thread[] = new Thread[3]; /* スレッド数3 */
      /* スレッド生成 */
      Thread thread[0] = new StartThread (0, 5000L);
      Thread thread[1] = new StartThread (1, 7000L);
      Thread thread[2] = new StartThread  (2, 4000L);
      /* スレッド実行開始 */
      for (Thread th : thread) /* 各スレッドをfor-each文で実行 */
         th.start(); /* “start()” はスレッドの実行開始命令 */
   }
}
```

参考：スレッドプログラム(java)

実行例

```
$ javac ThreadCreation.java
$ java ThreadCreation
                                        スレッド2 (4秒経過)
 スレッド0 (5秒経過)
                    スレッド1 (7秒経過)
                                        スレッド2 (4*2秒経過)
 スレッド0 (5*2秒経過)
                                        スレッド2 (4*3秒経過)
                    スレッド1 (7*2秒経過)
```

参考：スレッドプログラム(java)