オペレーティングシステム

資料　第5分冊(H30)

村田正幸（murata@ist.osaka-u.ac.jp)
○松田秀雄(matsuda@ist.osaka-u.ac.jp)

スケジューリングでの横取りなしとありの違い

- **横取りなし**スケジューリング
  - 実行中状態になったプロセスは、入出力等でブロックするか終了しない限り、別の状態に遷移しない
- **横取りあり**スケジューリング
  - 実行中状態になったプロセスが、ブロックや終了以外で、実行可能状態に遷移することがある（プロセスの**プロセッサへの割り付けを「横取り」する**）
- **横取りありスケジューリングの例**
  - ラウンドロビン
  - SRT（最小残余時間順）、優先度順

ラウンドロビン

- 横取りがあることを除けば、FCFSと同じ
- 実行中状態のプロセスを、一定時間ごとに横取り（図2.18）
  - 実行中のプロセスは実行可能キューの末尾へつながれる
  - 横取りが起こるまでの「一定時間」を、タイムスライス(time slice)という

タイムスライスの決め方

- タイムスライスの時間を長くすると、
  - プロセススイッチがまれにしか起こらなくなり、**プロセッサ利用率**が向上する
  - 長くし過ぎるとFCFSとの違いが小さくなり、応答時間が増えてしまう（ターンアラウンド時間が増加する）
- タイムスライスの時間を短くすると、
  - 実行可能キューにいるプロセスを順次実行することで、応答時間が減る（**ターンアラウンド時間**が短縮する）
  - 短くし過ぎると、プロセススイッチが多発することで、プロセッサ利用率が低下してしまう
- プロセスによって適切なタイムスライスの値が異なる
  - 対話的な処理では短くして応答時間を削減（**応答性能重視**）
  - 計算主体の処理では長くしてプロセススイッチを抑える
  - UNIXでは、10ms程度の基準時間(timer tick)ごとにプロセスの実行状況を監視して、タイムスライスの値を調整している

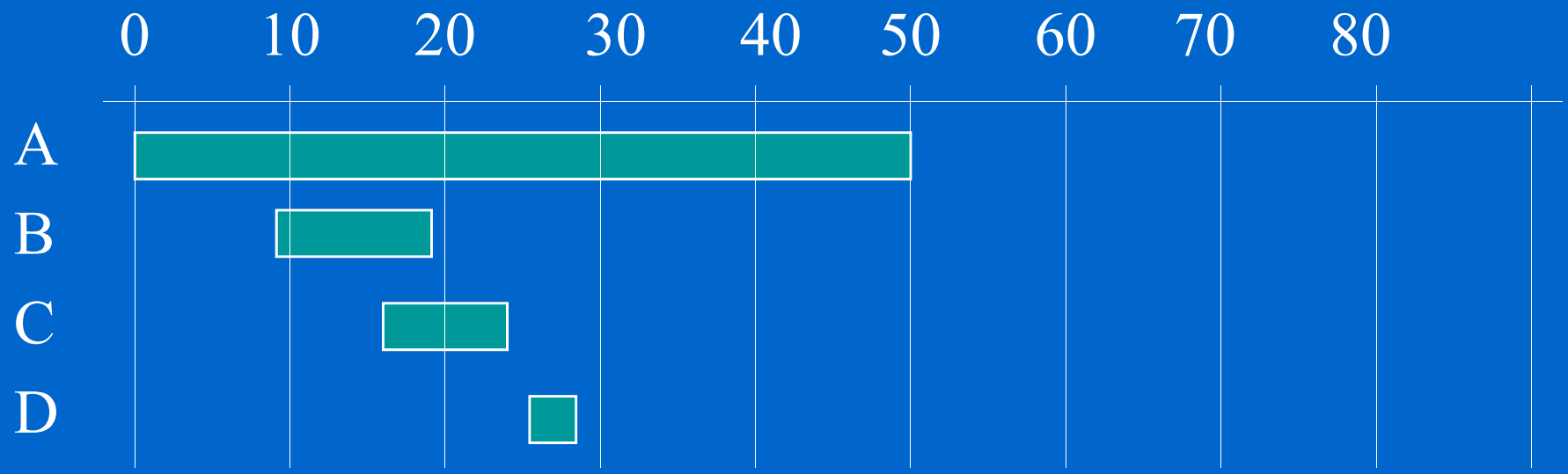
SRT（最小残余時間順）

- shortest remaining time firstの略
- 残りの処理時間が短いプロセスの順
  - 横取りがなければ、SJF（最短要求時間順）と同じ
  - 横取りが起こる（可能性がある）のは、新しいプロセスが生成（実行可能キューに到着）したとき
  - 現在実行中のプロセスの残余時間と比較し、新規プロセスの方が短ければプロセススイッチ（**横取り発生**）

プロセススケジューリングの例題(1)

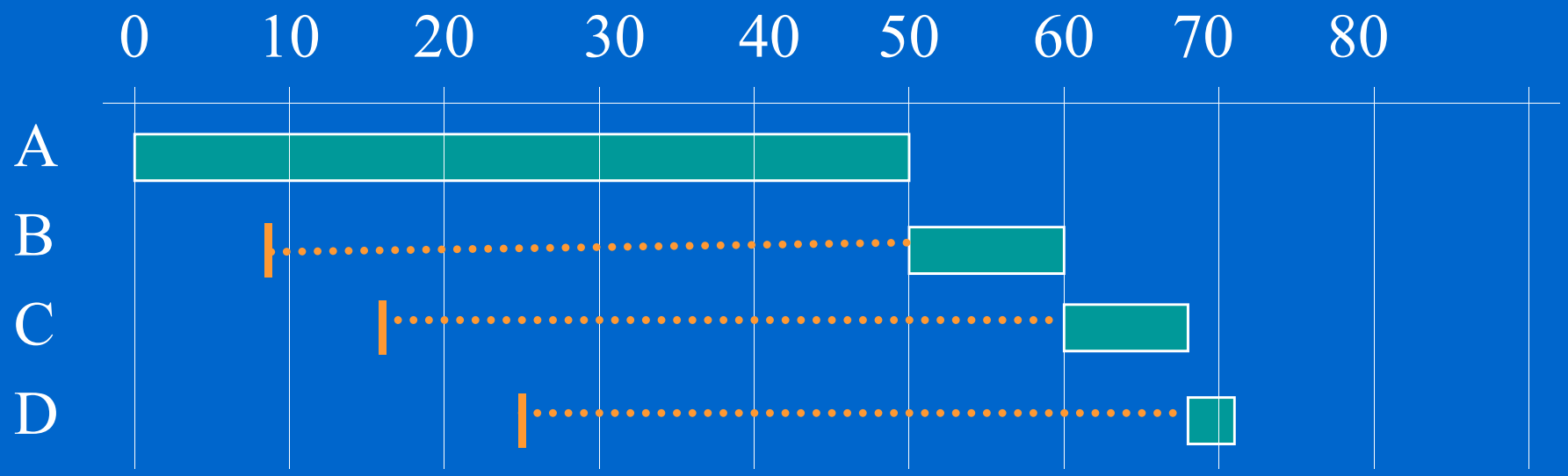
- 4つのプロセス
  A: 到着時刻　0,　処理時間 50
  B: 到着時刻　9,　処理時間 10
  C: 到着時刻16,　処理時間　8
  D: 到着時刻 26,　処理時間　3
- 処理はどのように進むか
  FCFS, SJF, SRT, ラウンドロビン（タイムスライス=10）
- それぞれの場合について答えよ

プロセススケジューリングの例題(2)



到着時刻と処理時間

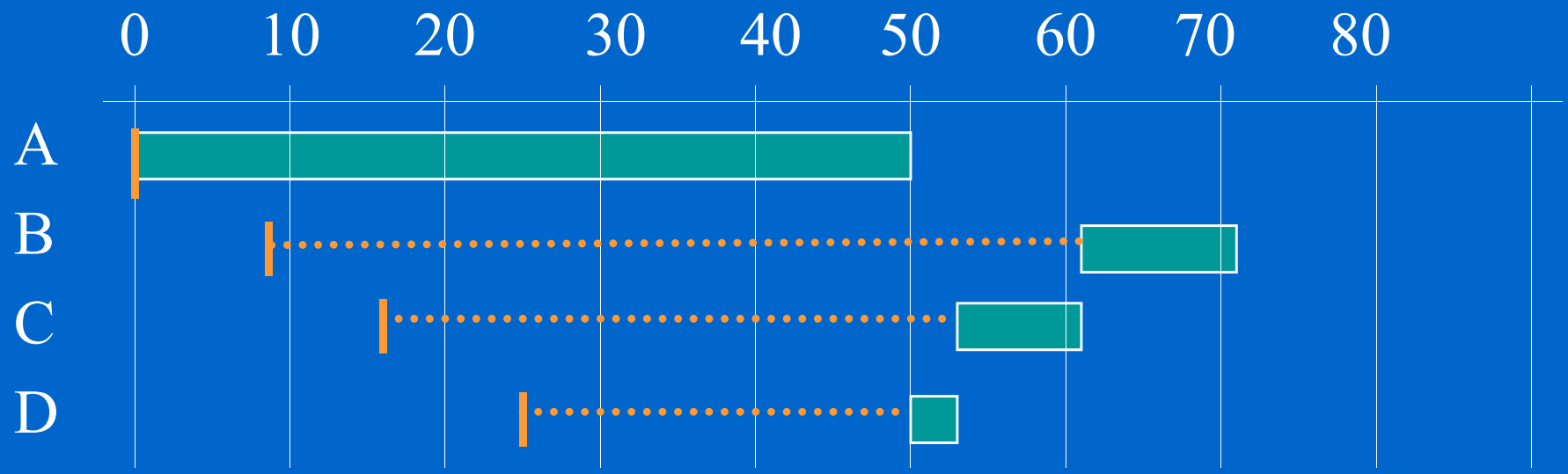
プロセススケジューリングの例題(3)
FCFS



到着順

プロセススケジューリングの例題(4)

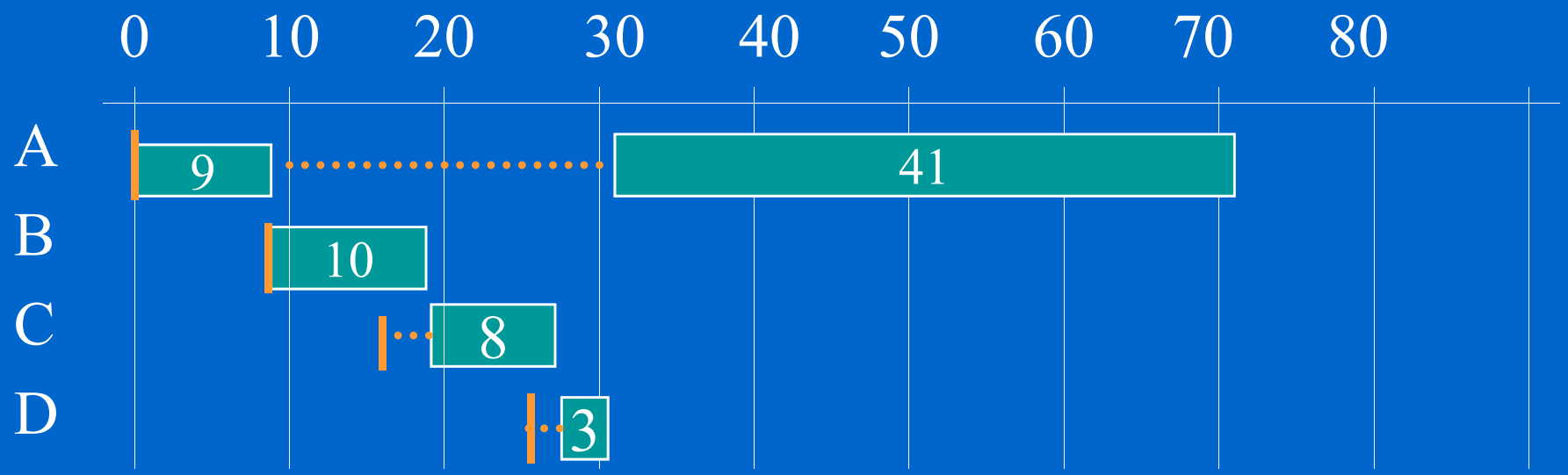
SJF



処理時間が短い順
（SJFでは横取りが起こらないことに注意）

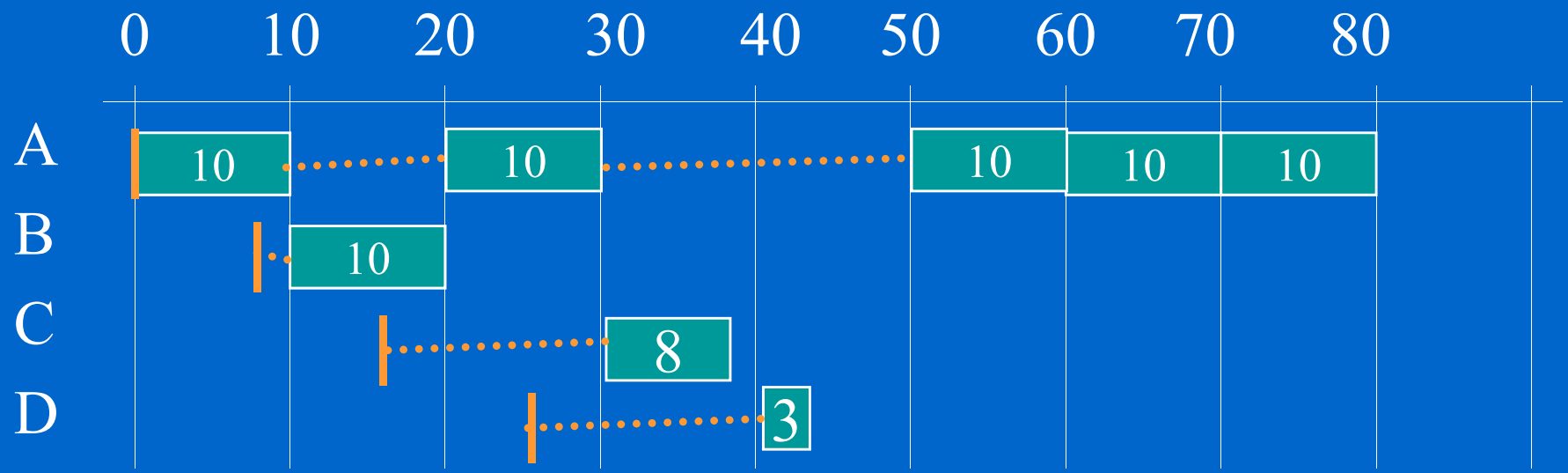
プロセススケジューリングの例題(5)

SRT



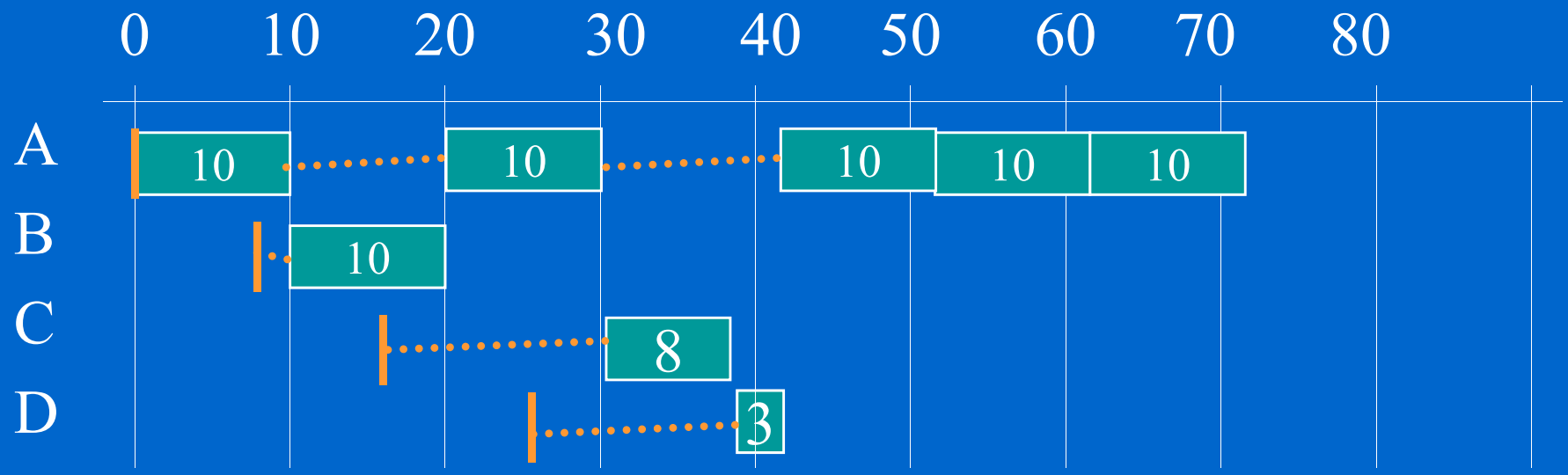
プロセス到着時に残余処理時間が短い順
（SRTでは横取りが起こる）

プロセススケジューリングの例題(6)
ラウンドロビン



- ラウンドロビン（タイムスライス=10）で、プロセスが終了した後に、次のタイムスライスまで待って別のプロセスをディスパッチする場合

プロセススケジューリングの例題(7)
ラウンドロビン（別の条件）



- ラウンドロビン（タイムスライス=10）で、プロセスが終了した後に、すぐに別のプロセスをディスパッチする場合

優先度順

- プロセスに優先度をつけ、実行可能キューにあるプロセスのうち優先度の高いものからディスパッチ

注意点

- 無限のブロック(infinite blocking)
  - いったん付与した優先度が固定されて変更されない場合に、優先度が低いプロセスがいつまで待っても実行されないことが起こり得る
  - この状態を、無限のブロックまたは**飢餓**(starvation)という
- 解決策:**エージング**(aging)
  - 長時間システムに滞在しているプロセスの優先度を時間経過につれて徐々に高くしていく

多重レベルスケジューリング(1)

- 優先度毎に実行可能キューを作る
- 優先度が高いキューが空のとき
  - 次に優先度の高いキューのプロセスをディスパッチ
- 多重レベルフィードバックスケジューリング
  - プロセスの実行可能キュー間の移動を許す
  - 新しく到着したプロセスは最大優先度のキューへ
  - タイムスライスを使い切ったら一つ低い優先度へ
- 飢餓状態を起こす可能性がある

多重レベルスケジューリング(2)

- 飢餓状態を避けるために
  - エージング（待ち時間が長いプロセスは優先度を上げる）
  - プロセッサを多く消費するプロセスは優先度を下げる

    UNIXの実装
    - 優先度値=基本優先度　＋　最近のプロセッサ消費量
    - 「優先度値」が小さいほど、プロセスの優先度が高い
    - niceコマンド: 指定できる優先度値の範囲は、-20（優先度最高）から19（優先度最低）
    - 負の値を指定することができるのはスーパーユーザのみ

プロセスのスケジューリングのまとめ

キューへのつなぎ方	横取りなし	横取りあり
到着順	FCFS	ラウンドロビン
処理時間の短い順	SJF（到着時の処理時間）	SRT（残りの処理時間）
（処理時間以外も含めた）優先度順		優先度順 多重レベル

2.1.6 スレッド

- プロセスの中の「マシン命令の実行の流れ（制御フロー）」で、プロセッサにおける「動的な実行単位」をスレッド（thread）という
- 最近のOSの多くはスレッドに対応している（カーネルが複数スレッドで実行される）
- スレッドを考慮するプロセスでは、1プロセスで複数の実行の状態を持つことができる（並行実行が可能）

なぜスレッドが必要か？

- 1プロセス内での並行処理が記述できる
  - 例：GUIシステム
    - 個々のボタンやスライダの処理をそれぞれ別々のスレッドで実行
- マルチプロセッサ・マルチコアへの対応
  - 従来のプロセス
    - 1プロセスにプロセッサ（またはコア）1個のみ対応
  - スレッド
    - 複数のスレッドを生成することで、複数のプロセッサ（またはコア）を使用可能

スレッドの特徴

- プロセスよりも生成や消滅が簡単にできる（理由は後述）

→プロセスよりも有効時間（存在している時間、ライフタイムともいう）が短い

- プロセスと比べて、コンテクスト（実行に必要な情報）が小さいため、空間サイズは小さい

→スレッドの切り替え（スレッドスイッチという）は、プロセススイッチよりも軽快で速い

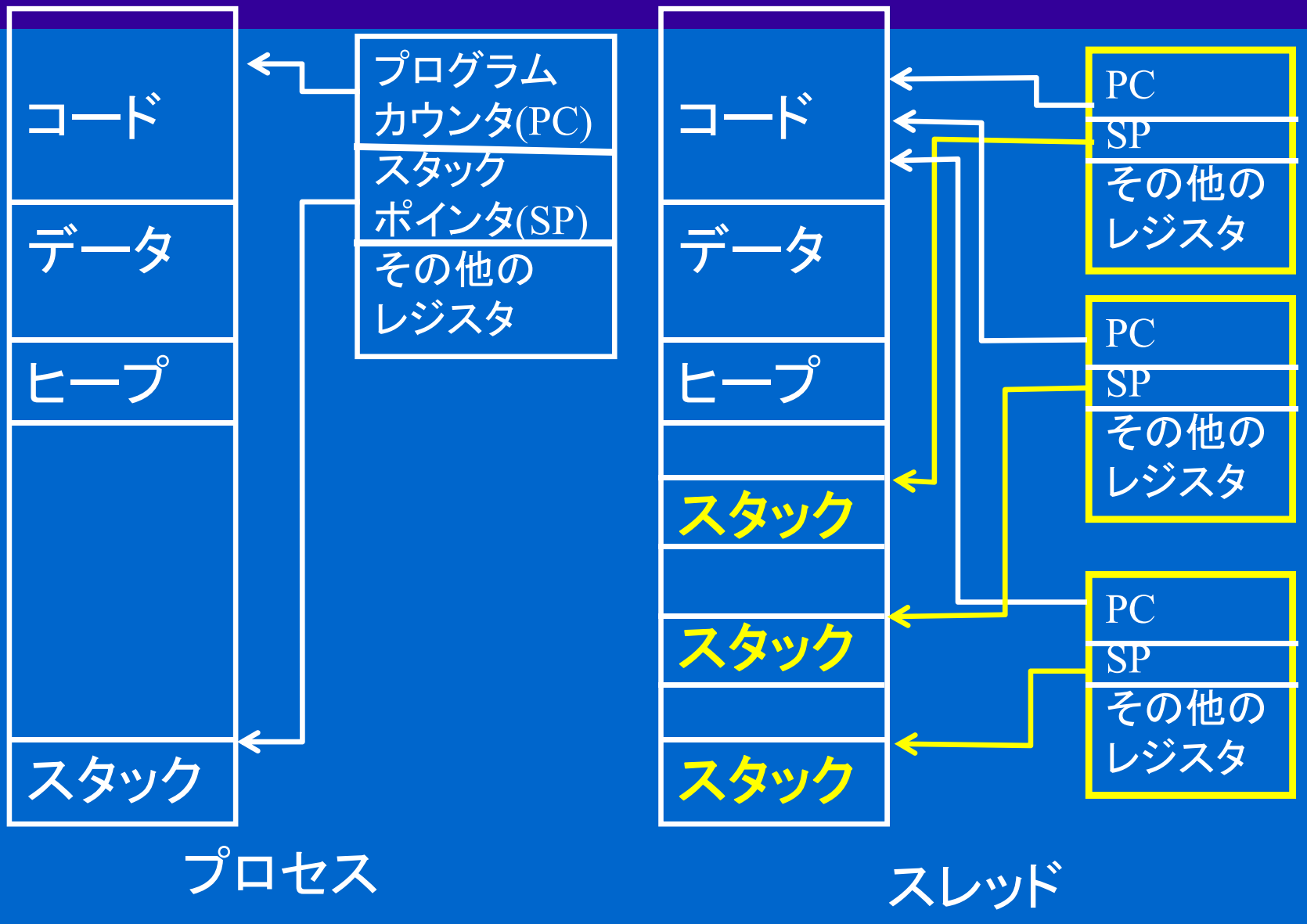
- 最近のOSでの取り扱い
  - プロセスは、「プロセッサ以外のハードウェア資源（メモリなど）に割り付ける単位」
  - スレッドは、「プロセッサへ割り付ける単位」

スレッドコンテクスト

- プロセス制御ブロック(プロセスコンテクスト)の中で、
  - プロセッサの状態、コンディション(プロセッサのフラグ類)
  - プログラムカウンタ、汎用レジスタ
  - プロセス領域のうちのスタックフレームと、スレッド生成後に作られたヒープ

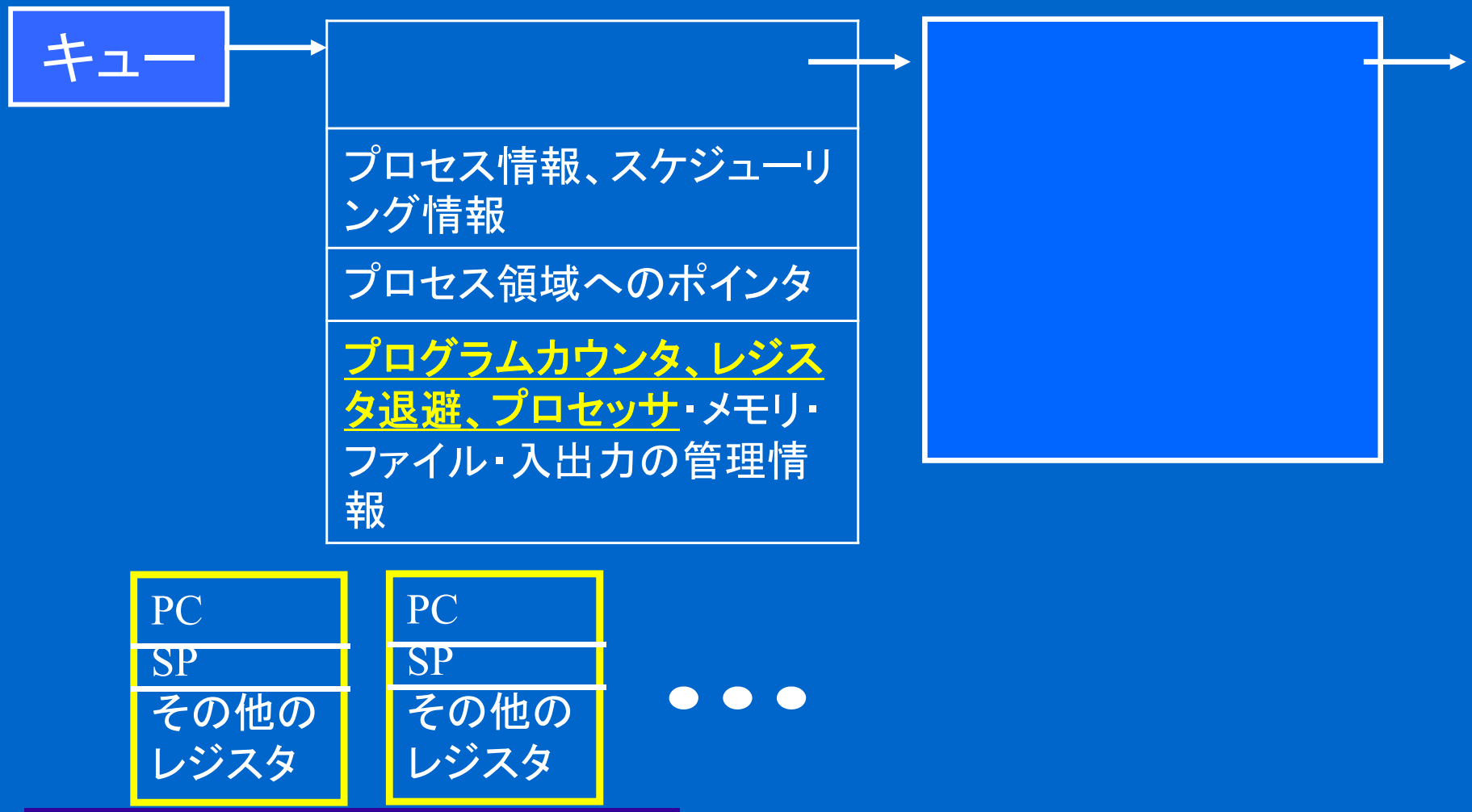
をスレッドごとに持つ(表2.2)

スレッドのメモリへの割り付け



プロセス制御ブロックとの関係

- PCBのうち、プロセッサに関係した部分だけが、スレッドごとに別々に作成される



スレッド間の通信

- スレッドは資源を共有
- 通信は簡単
  1. メモリに、共有データの置き場所を作る
  2. 送信側スレッドはデータを置く
  3. 受信側スレッドはデータを読み出す

スレッドとカーネル

- プロセスがシステムコールを発行
  - 事象待ちのある処理だった場合、プロセスはブロックする
- あるスレッドがシステムコールを発行
  - カーネルが1スレッドでしか動いていないと、事象待ちのある処理だった場合、プロセスがブロックする
  - → 実質的にそのプロセスの全スレッドがブロックする
- カーネルの処理も複数のスレッドにするカーネルスレッドが必要

ユーザスレッドとカーネルスレッド

- 2種類のスレッドを設定
  - ユーザスレッド：ユーザプログラムで制御するスレッド
  - カーネルスレッド：カーネルが制御するスレッド
- ユーザスレッドは、プログラムの中での並行に処理可能な部分の実行に対応する
  - 実際に実行するにはプロセッサの割り付けが必要
- カーネルスレッドにはプロセッサが割り付けられる
  - ユーザスレッドが並行実行するには、カーネルスレッドを通して、プロセッサを割り付けなければならない（理由は後述）

ユーザスレッド

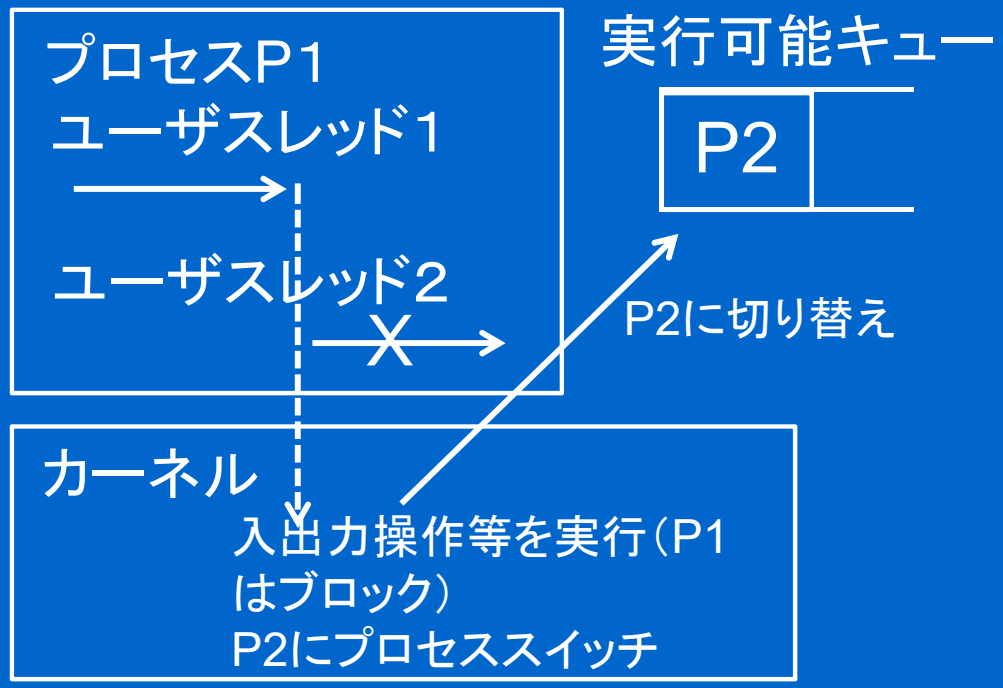
- ユーザがスレッドライブラリを用いて生成し、制御する(とユーザプログラムで記述する)
  - 新たなスレッドの生成や、別のスレッドへの切替えは、スレッドライブラリの呼出しで行う
  - カーネルスレッドがなければ、新たなスレッドを生成しても、プロセッサへの割り付けはない

カーネルスレッド

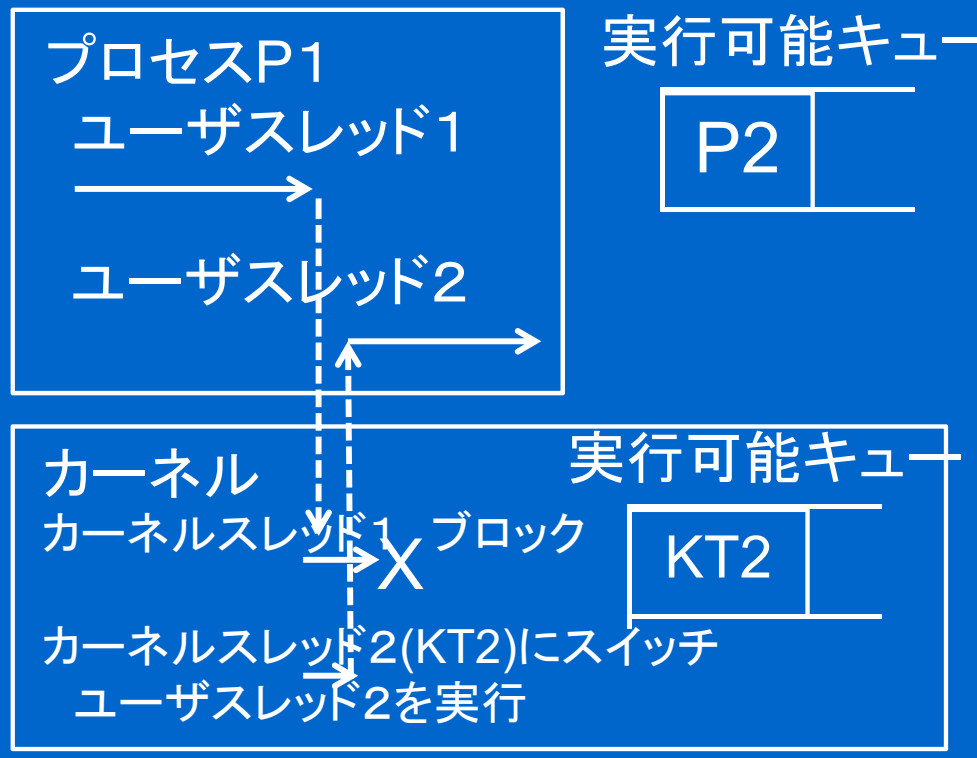
- カーネルレベルのスレッド
- カーネル空間で制御される
  - スレッドごとにプロセッサを割り付け可能
  - スケジューリングは原則としてカーネルが行う
    - ユーザプログラムでは制御できない
- カーネルがスイッチを行う
  - スレッドの数が多いとマルチプロセスに近くなる(スレッドの処理にプロセスの処理と同程度の負荷がかかる)

カーネルスレッドの必要性

カーネルスレッドは、あるユーザスレッドがシステムコールを実行したとき、別のユーザスレッドに切り替えるのに必要



カーネルでスレッドが実行できないプロセスP1をブロックし別プロセスP2に切り替える（**ユーザスレッド2は実行されない**）

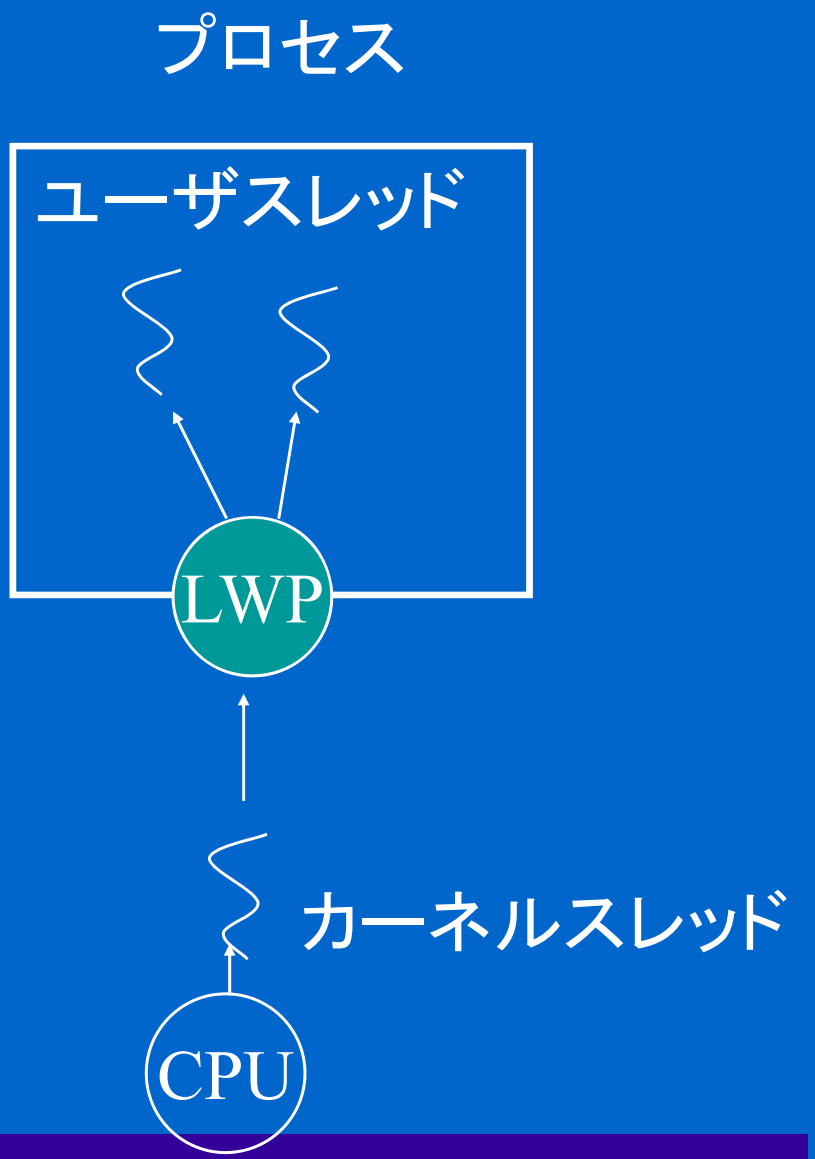


カーネル内にスレッドの実行可能キュー
カーネルのスレッドスイッチにより**カーネルスレッド2を起動しユーザスレッド2を実行**

スレッドの利用例

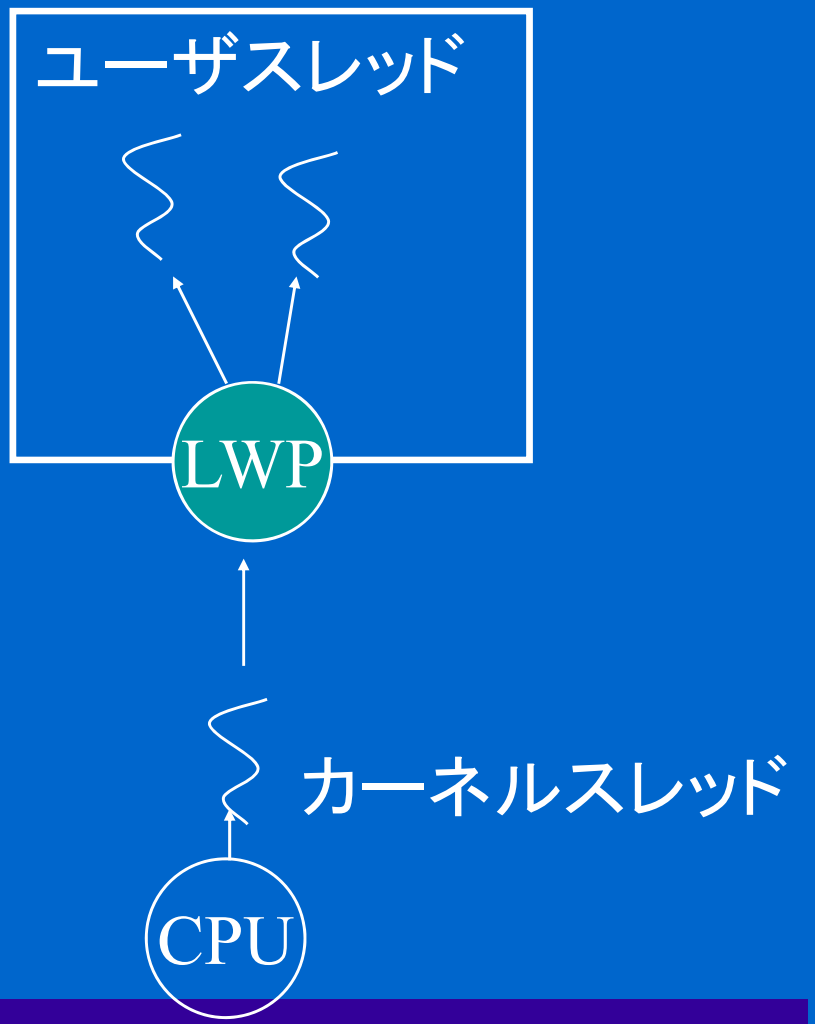
- IEEEで標準仕様が定められている
  - POSIX thread (pthread)と呼ばれる
  - 最も広く採用されている
- OSごとに少しずつ違った実装がある
- Solaris（UNIX系OS）のスレッドを例に説明

Solarisでのスレッド



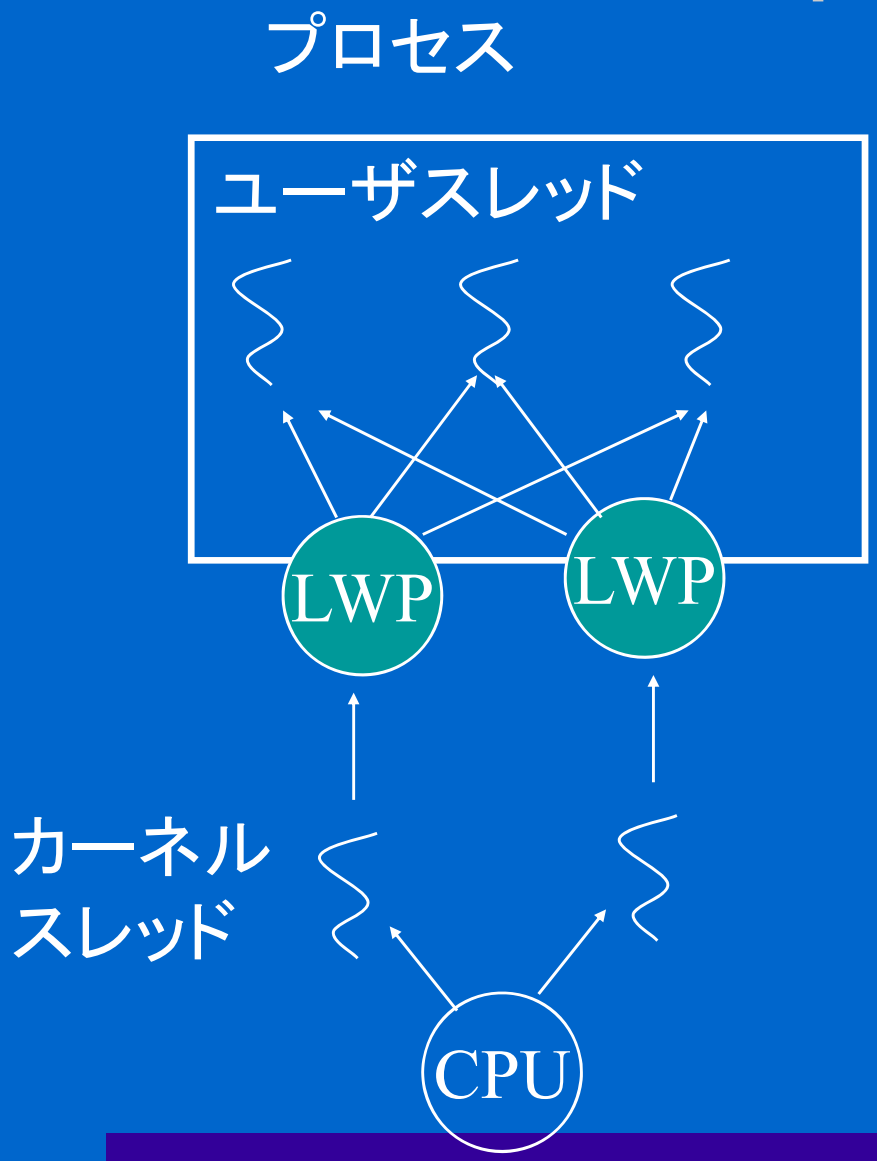
- ユーザスレッドとカーネルスレッドの間を対応付けるものとして、**LWP（軽量プロセス）**を導入（対応付けのためだけの存在）

Solarisでの例1（カーネルスレッドを1個に固定）



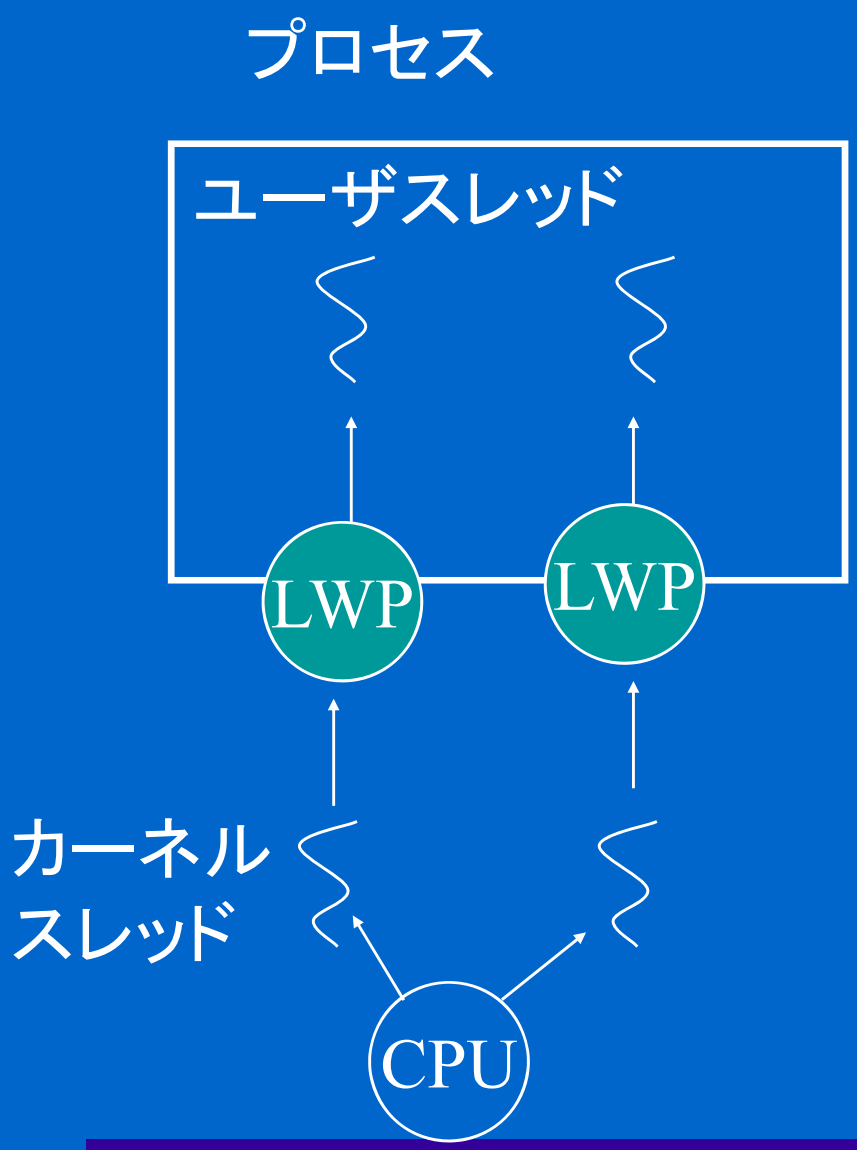
- 複数のユーザスレッド
- カーネルスレッドは1個
  - ある瞬間に動くのは高々1スレッド
- ユーザスレッドはシステムコールなどにより、まとめてブロックする

Solarisでの例2（ユーザスレッド数より小さい複数のカーネルスレッド）



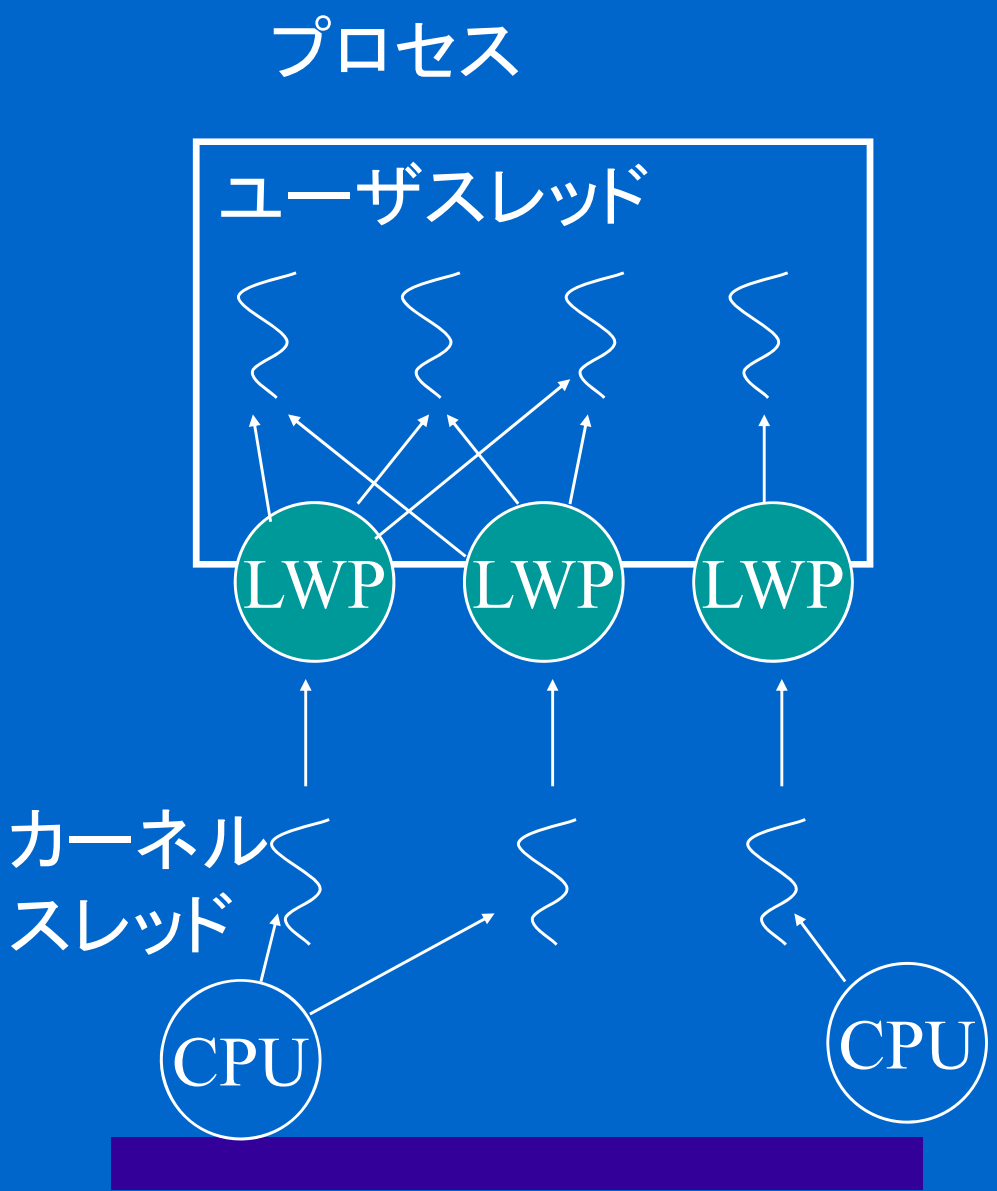
- ユーザスレッドの数＞カーネルスレッドの数≧2
  - カーネルスレッドの数だけ並行に動く
- ユーザスレッドが1個だけブロックしても、他のユーザスレッドは動く
- カーネルスレッドの数と同じ数（左図では2個）のユーザスレッドがブロックすると、残りのユーザスレッドもブロックする

Solarisでの例3
（ユーザスレッド数=カーネルスレッド数）



- ユーザスレッドの数
  ＝　カーネルスレッドの数
  - 全部のユーザスレッドが並行に動く
- ユーザスレッドが生成されるごとにカーネルスレッドも生成される
- Windows　NT系(2000,XP,Vista, 7, 8, 10)は、この方式

Solarisでの例4（プロセッサまたはコアが複数個）



- プロセッサ（またはコア）が複数ある場合は、カーネルスレッドとプロセッサ（コア）の対応を設定できる
- プロセッサ（コア）に割り付けられたカーネルスレッドが実行される

2.2 並行プロセス
ープロセスの管理の理論ー

- 同時に実行可能な複数のプロセスを**並行プロセス（concurrent process）**という
  - 「同時に実行可能」とは、「真に同時に実行した結果（それぞれ別のプロセッサで実行した結果）」と「任意の逐次順序で実行した結果」がすべて同じである場合をいう
- 同時に実行不可能で、ある一意に定められた順序だけで逐次実行する複数プロセスを**逐次プロセス**という

並行プロセスの生成

- 並行プロセスの生成を指定する実例には、次のようなものがある（図2.21）
- コルーチン
  - ユーザプログラムレベルでの並行プロセスの一つの実行概念であり、OSで実装されている例は少ない
- フォークアンドジョイン
  - **UNIXで標準的に実装**されている並行プロセスの生成方法
- 並行文
  - フォークアンドジョインの多重版
  - 並行プロセスの教科書の一つで、並行プロセスの生成方法として紹介されている（現在のOSでの実装例は少ない）