オペレーティングシステム

資料　第5分冊(H27)

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)またはクォンタム(quantum)という

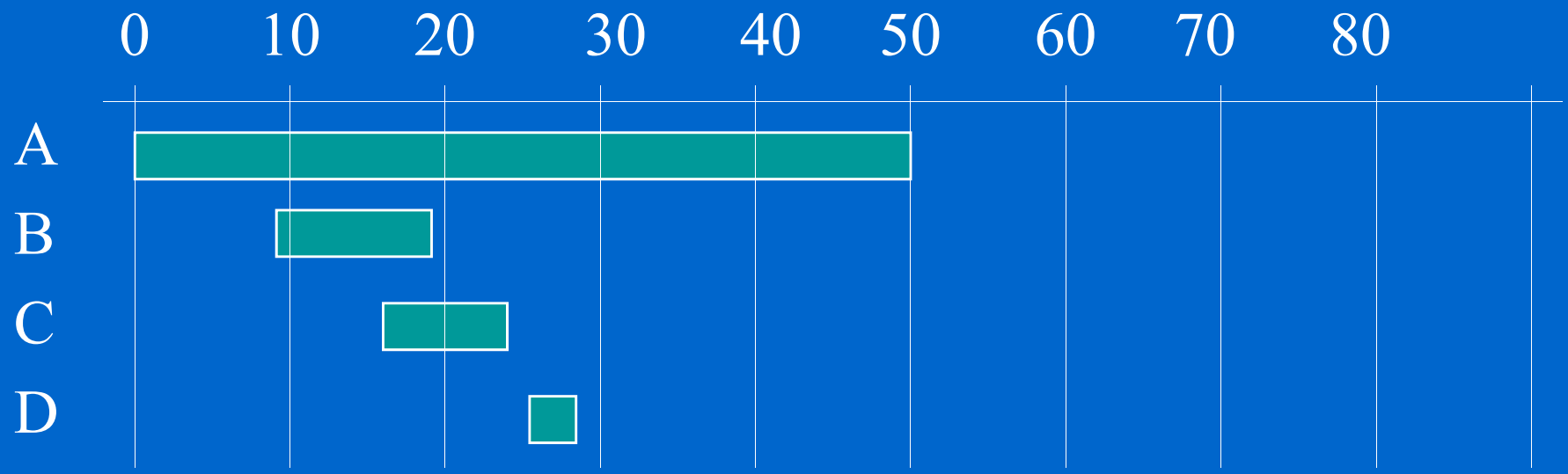
SRT（最小残余時間順）

- 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

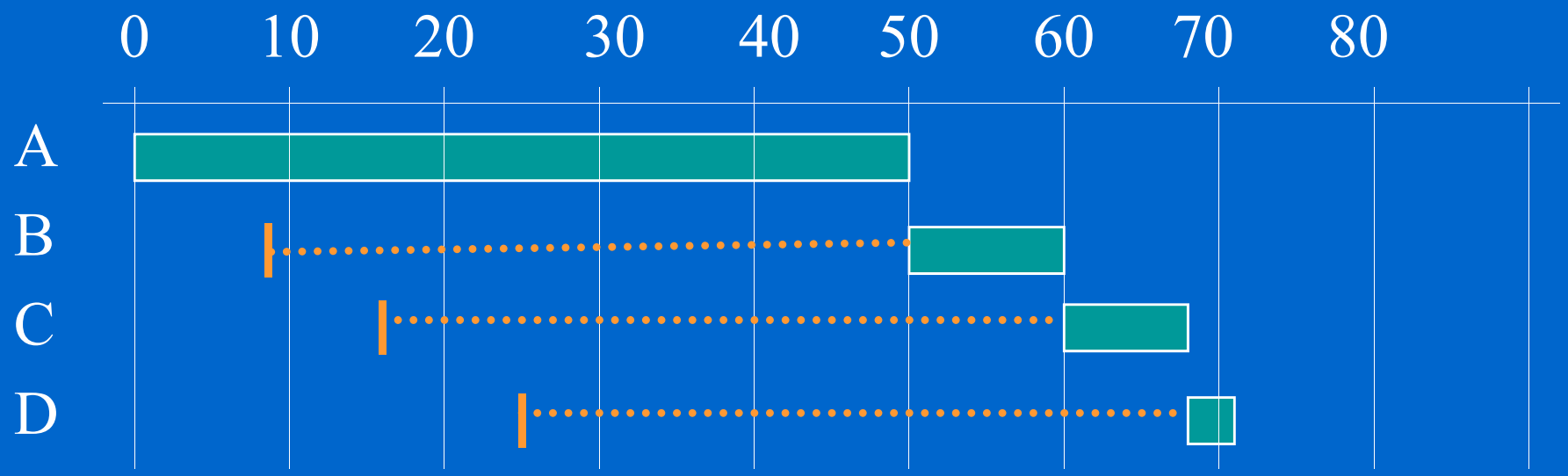
プロセススケジューリングの例題(2)



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)

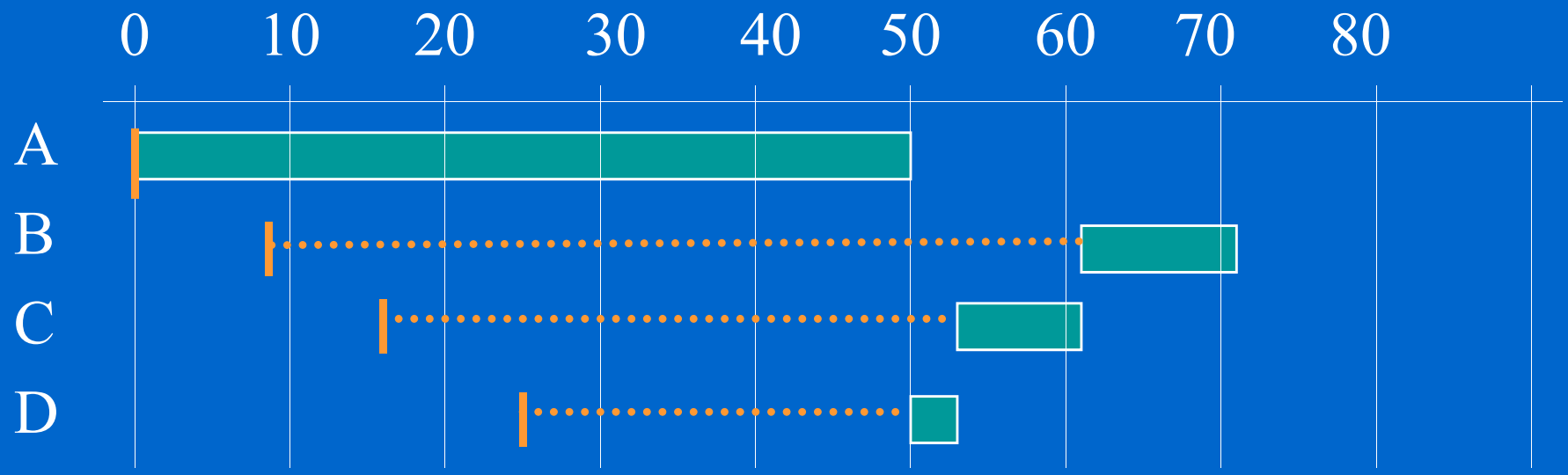
FCFS



到着順

プロセススケジューリングの例題(4)

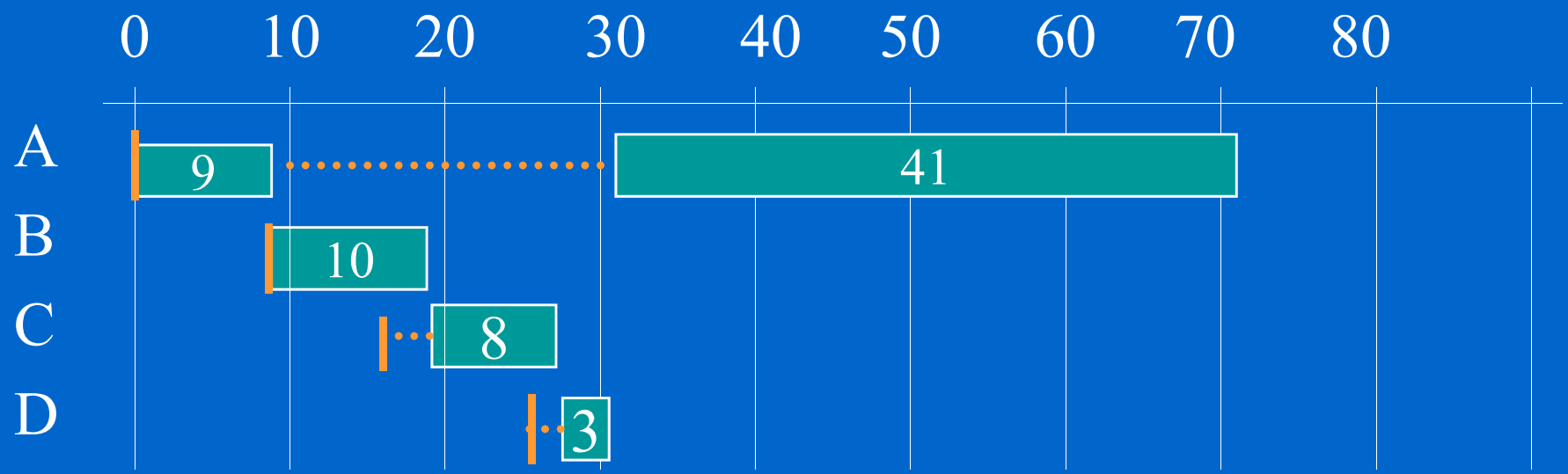
SJF



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）

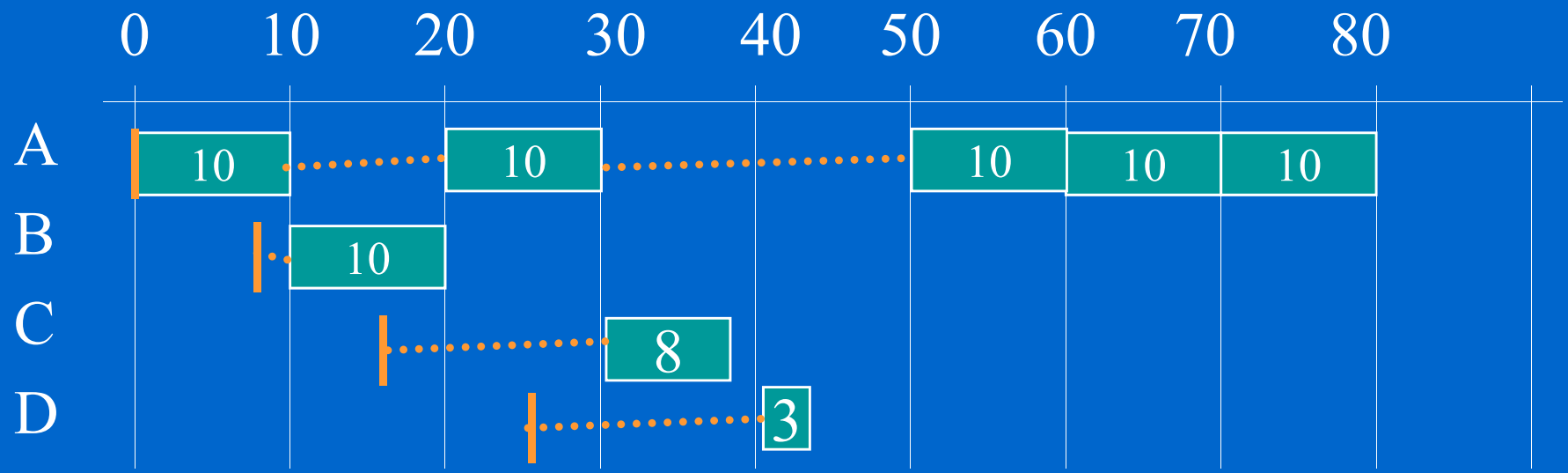
プロセススケジューリングの例題(5)

SRT



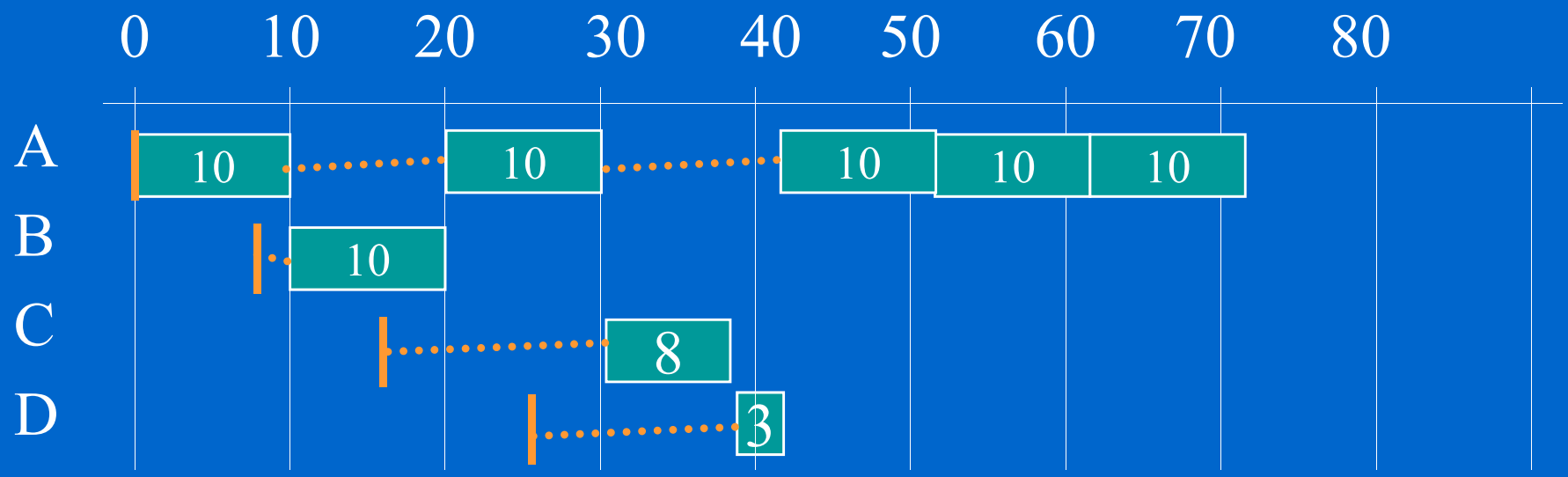
プロセス到着時に残余処理時間が短い順
（SRTでは横取りが起こる）

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
- 通常、タイムスライスは優先度の逆比例
- 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)

- バリエーションはいろいろ
- プロセスの優先度が決まっている
  - 新しく到着したプロセスもその優先度で決まるキューに入れる
- 飢餓状態を避けるために
  - 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の場合
    - 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

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システムで時間のかかる処理を実行中
    - 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）

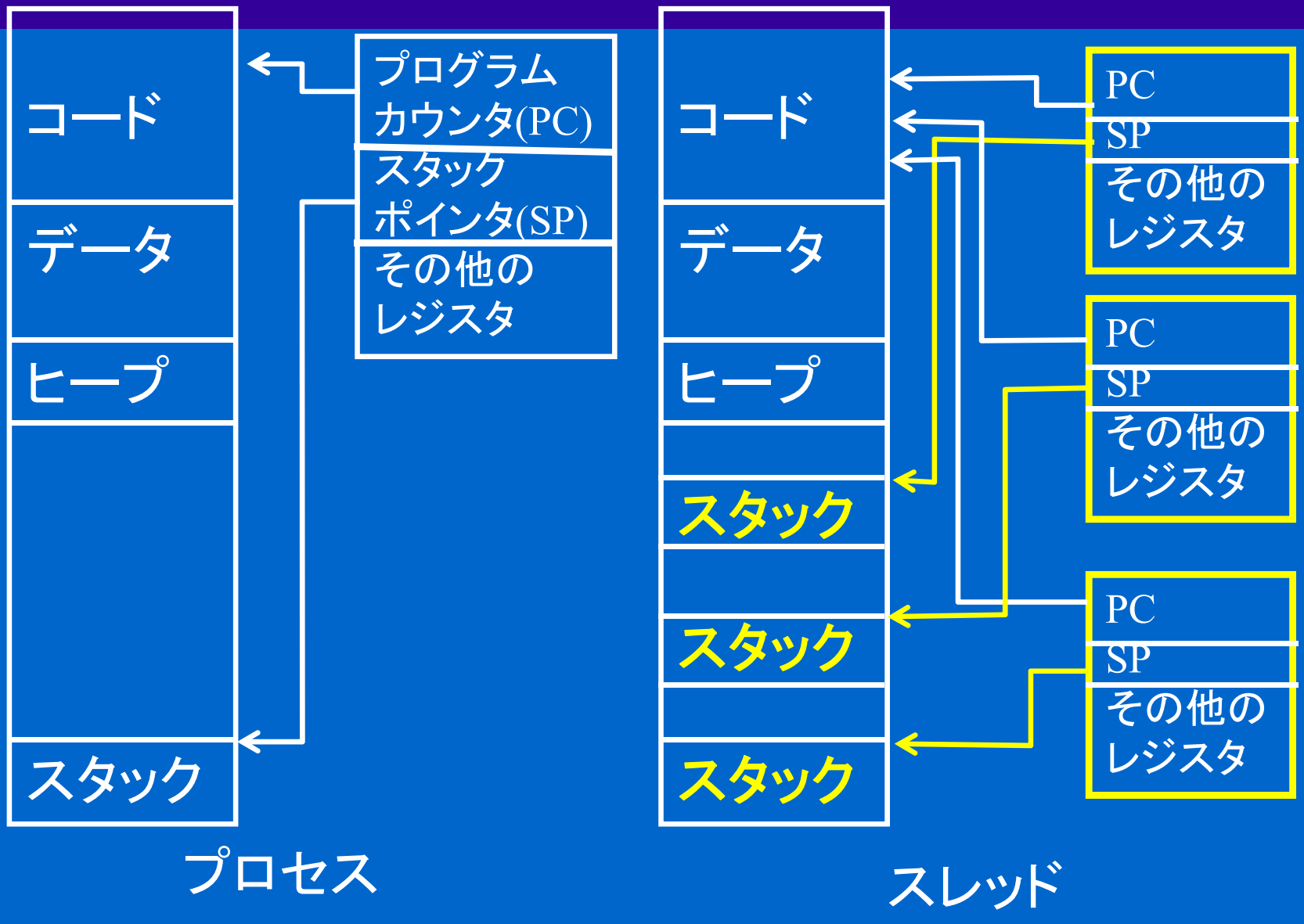
→プロセスよりも有効時間（存在している時間、ライフタイムともいう）が短い

- プロセスと比べて、コンテクスト（実行に必要な情報）が少ないため、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

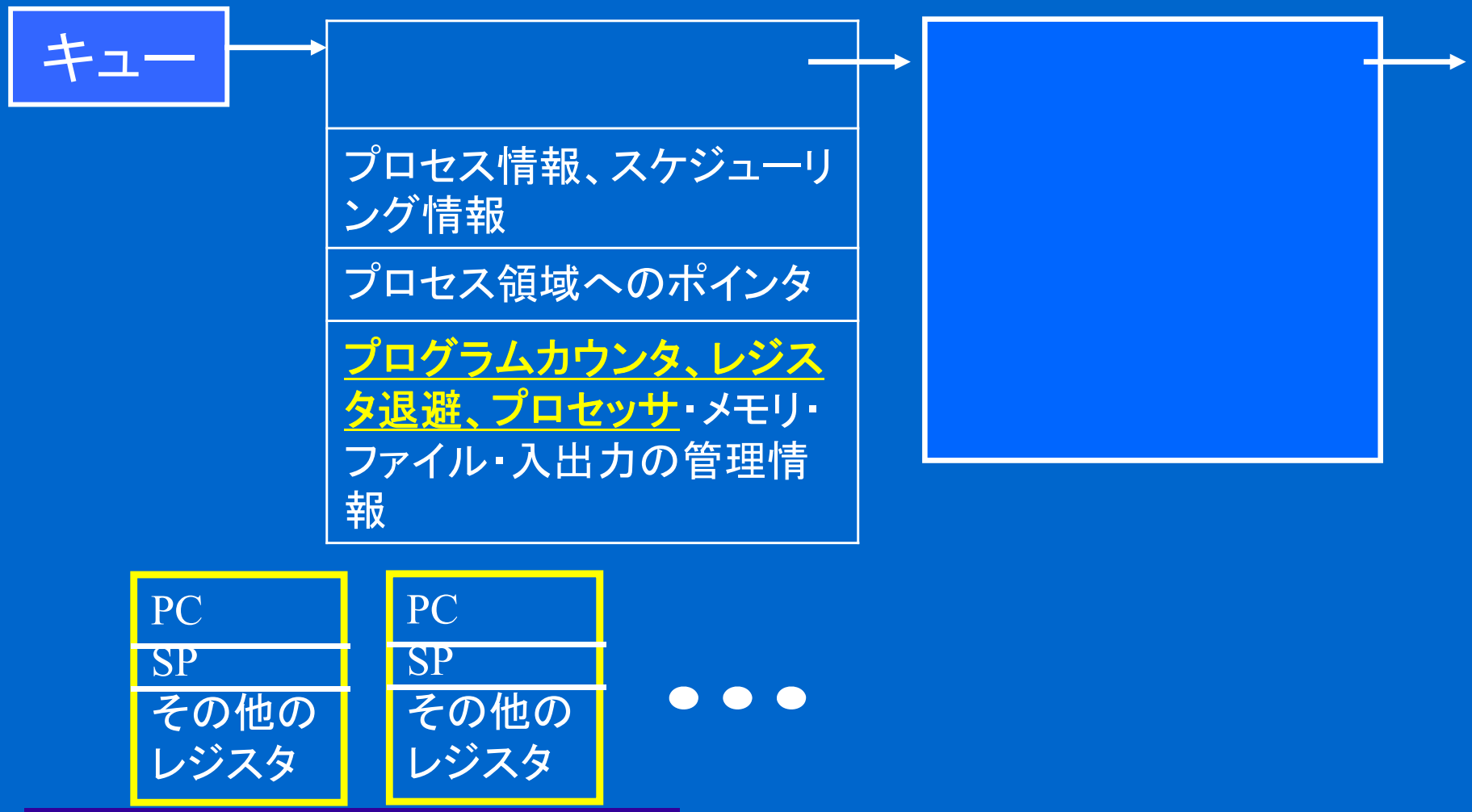
- プロセスは、「プロセッサ以外のハードウェア資源（メモリなど）に割り付ける単位」
- スレッドは、「プロセッサへ割り付ける単位」

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
  - すぐに終わらない処理だった場合、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、入出力処理など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

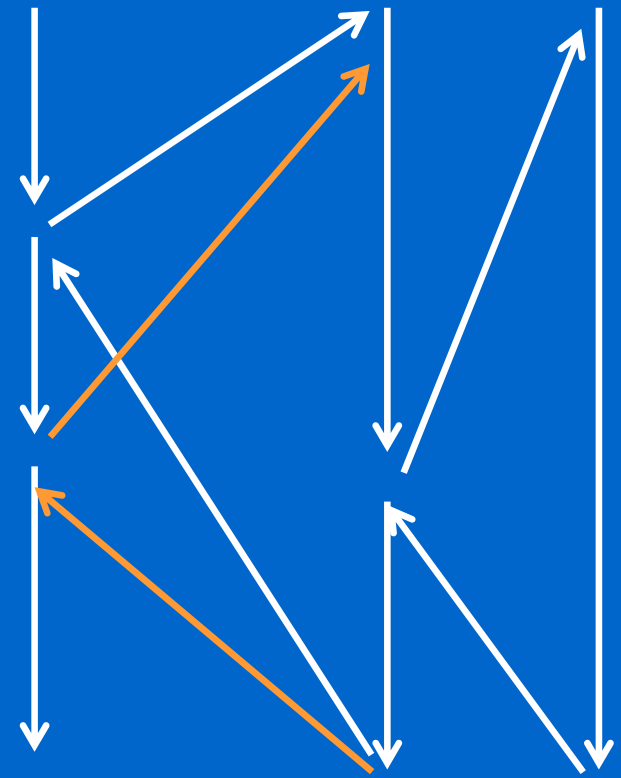
- 2種類のスレッドを設定
  - ユーザスレッド：ユーザプログラムで制御するスレッド
  - カーネルスレッド：カーネルが制御するスレッド
- ユーザスレッドは、プログラムの中での並行に処理可能な部分の実行に対応する
  - 実際に実行するにはプロセッサの割り付けが必要
- カーネルスレッドにはプロセッサが割り付けられる
  - ユーザスレッドが並行実行するには、カーネルスレッドを通して、プロセッサを割り付けなければならない

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)
- スレッドの切替えは、ライブラリの呼出しで行う
  - 一種のコルーチン(後述)
  - ユーザスレッドだけだと、あるスレッドがシステムコールを実行すると、全スレッドがブロックしてしまう

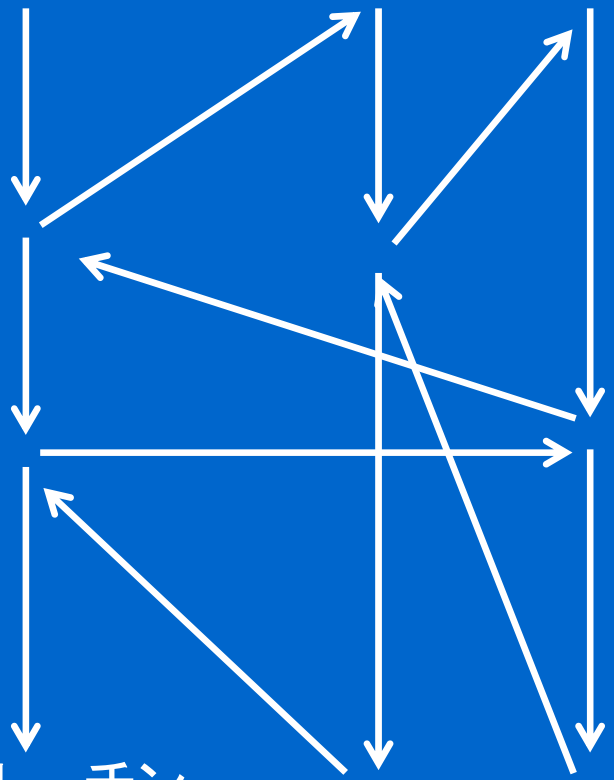
サブルーチンとコルーチン

プログラムA　B　C



サブルーチン
- callとreturnで呼出しと復帰
- 呼出し先では最初から実行し、呼出し元のcall直後から再開する（非対称）

プログラムA　B　C



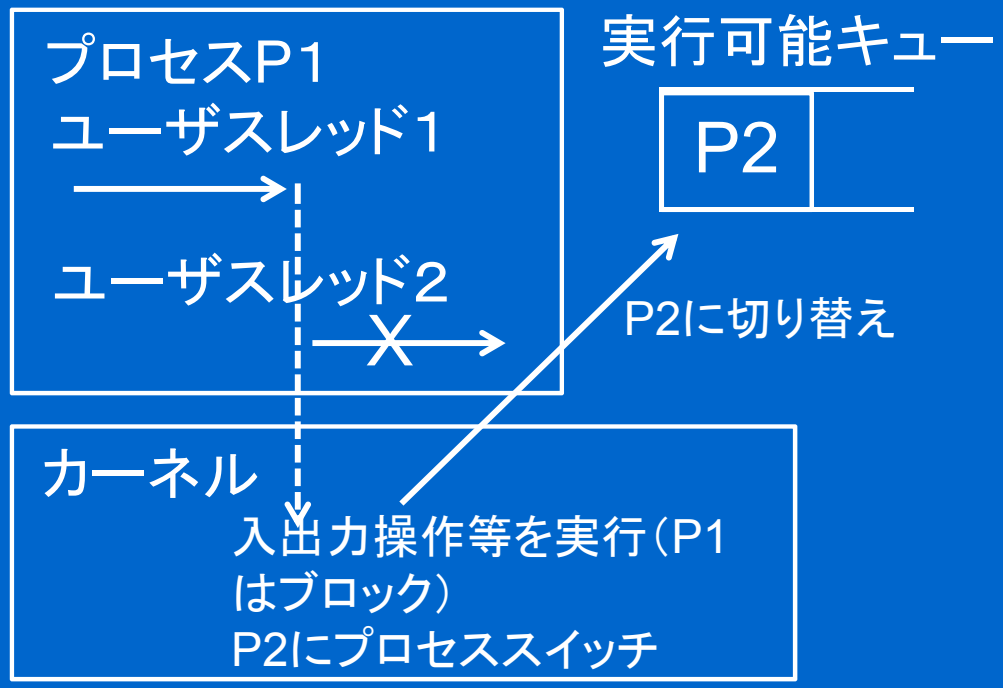
コルーチン
- resumeで呼出し（復帰命令はない）
- 呼出し先で他のプログラムをresumeした場合は、直後から再開する（対称）

カーネルスレッド

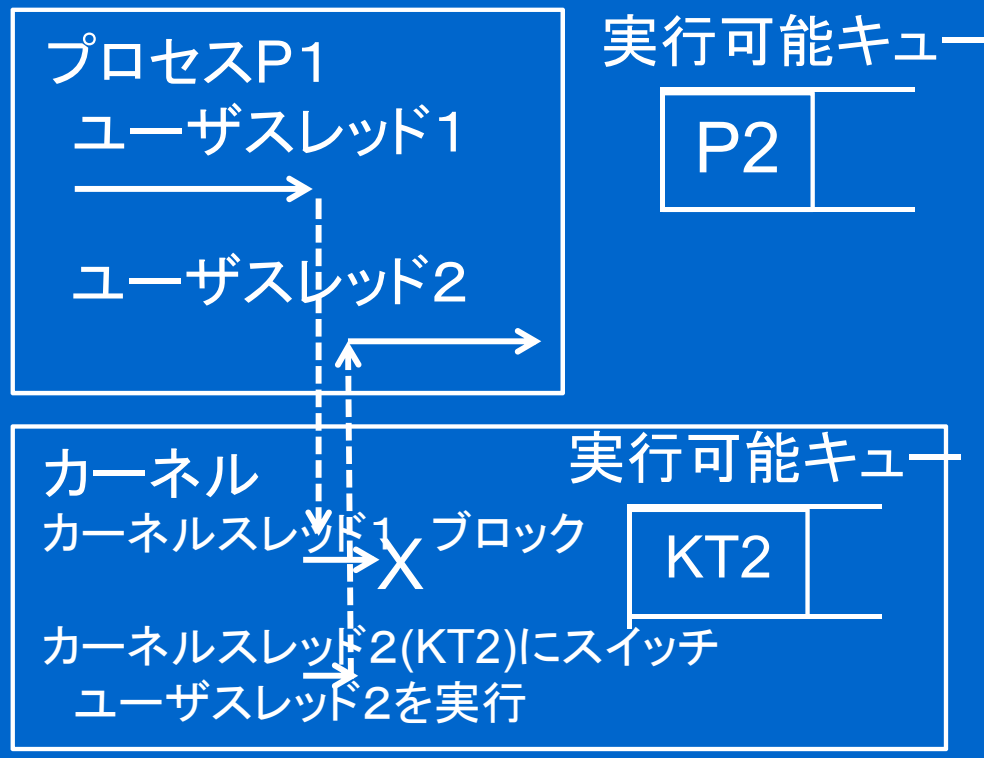
- カーネルレベルのスレッド
- カーネル空間で制御される
  - スレッドごとにプロセッサを割り付け可能
  - スケジューリングは原則としてカーネルが行う
    - ユーザプログラムでは制御できない
- カーネルがスイッチを行う
  - スレッドの数が多いとマルチプロセスに近くなる(スレッドの処理にプロセスの処理と同程度の負荷がかかる)

カーネルスレッドとは？

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**）

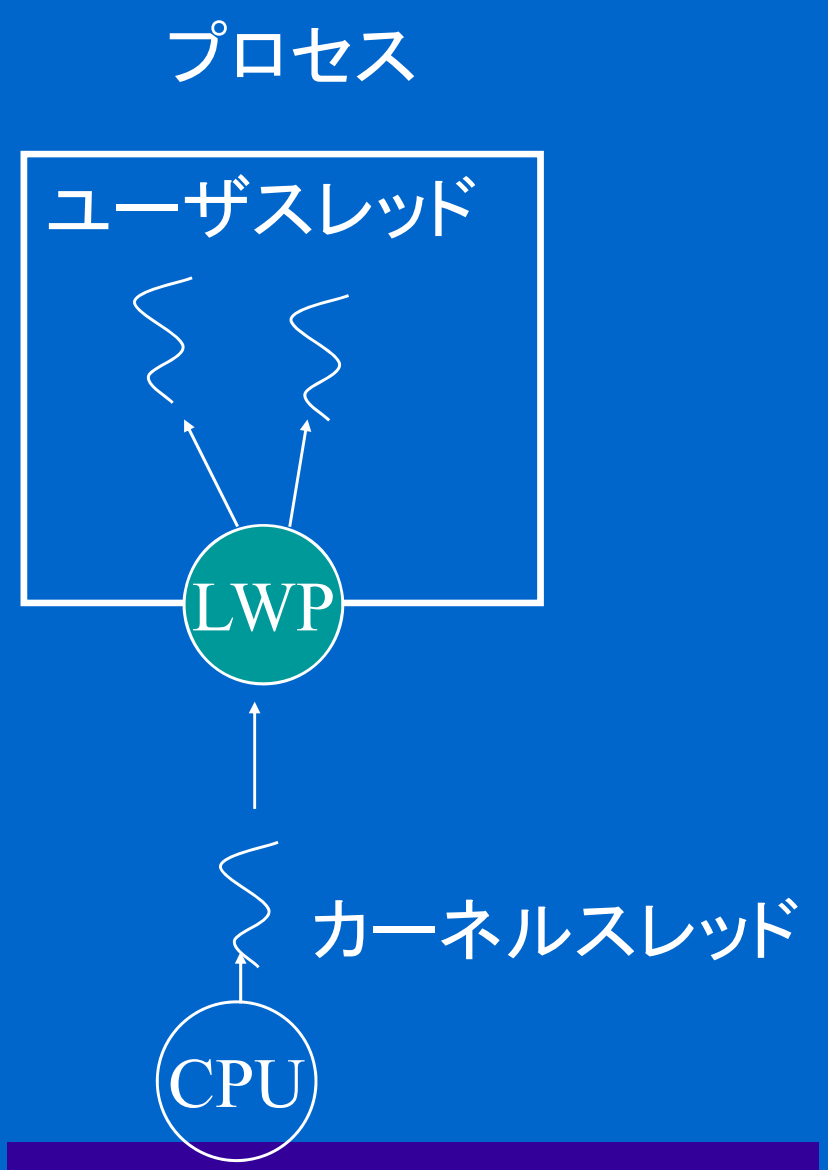


カーネル内にスレッドの実行可能キュー
カーネルのスレッドスイッチによりカーネルスレッド2が起動しユーザスレッド2を実行

スレッドの利用例

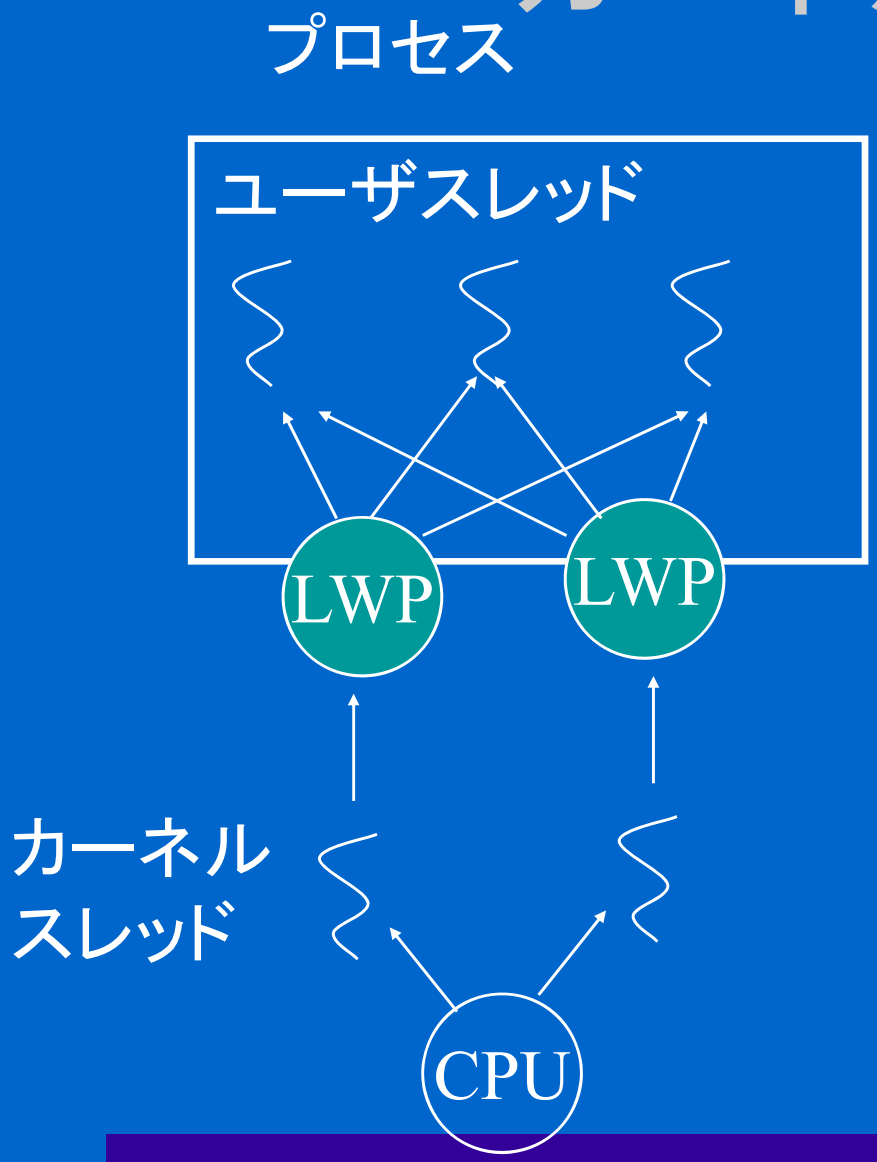
- IEEEで標準仕様が定められている
  - POSIX thread (pthread)と呼ばれる
  - 最も広く採用されている
- OSごとに少しずつ違った実装がある
- Solaris（UNIX系OS）のスレッドを例に説明

Solarisでの例1（カーネルスレッドを1個に固定）



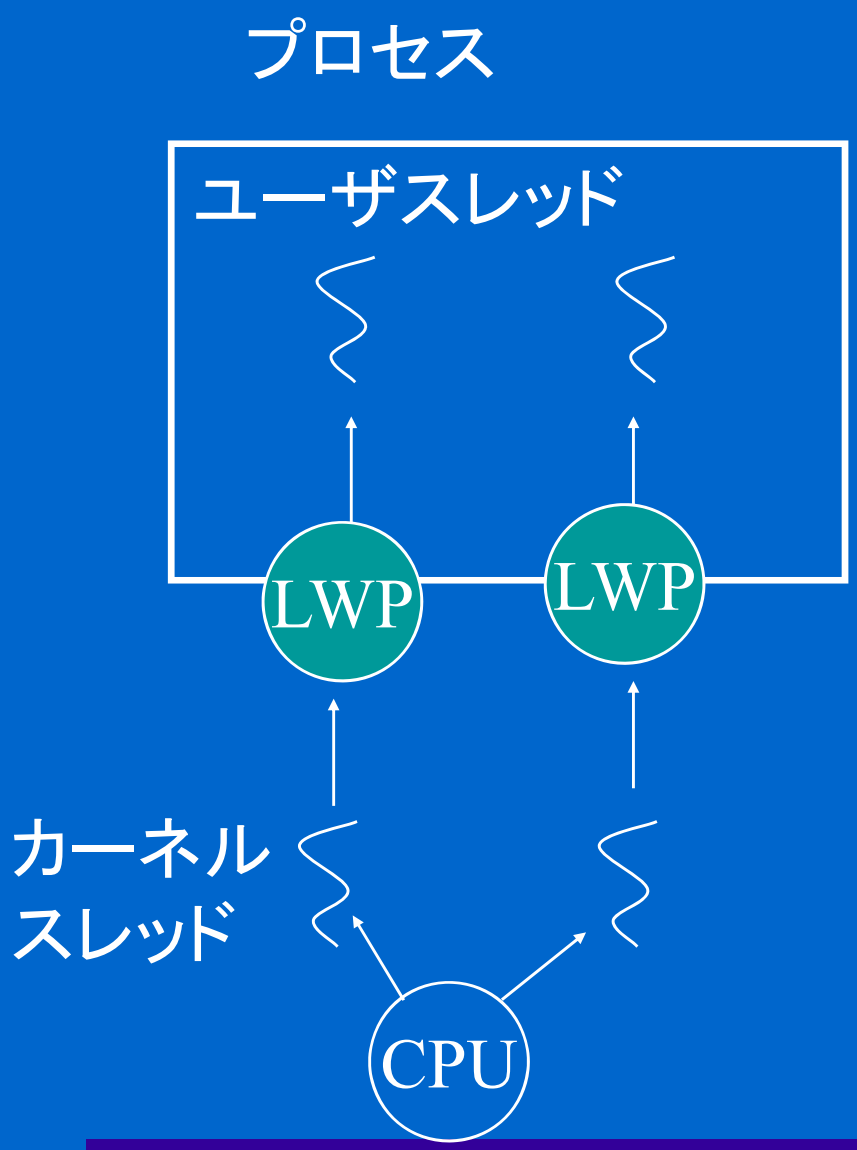
- 複数のユーザスレッド
- カーネルスレッドは1個
  - ある瞬間に動くのは高々1スレッド
- LWP（軽量プロセス）は、ユーザスレッドとカーネルスレッドを対応付ける
- ユーザスレッドはシステムコールなどにより、まとめてブロックする
- コルーチンと同じ

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）



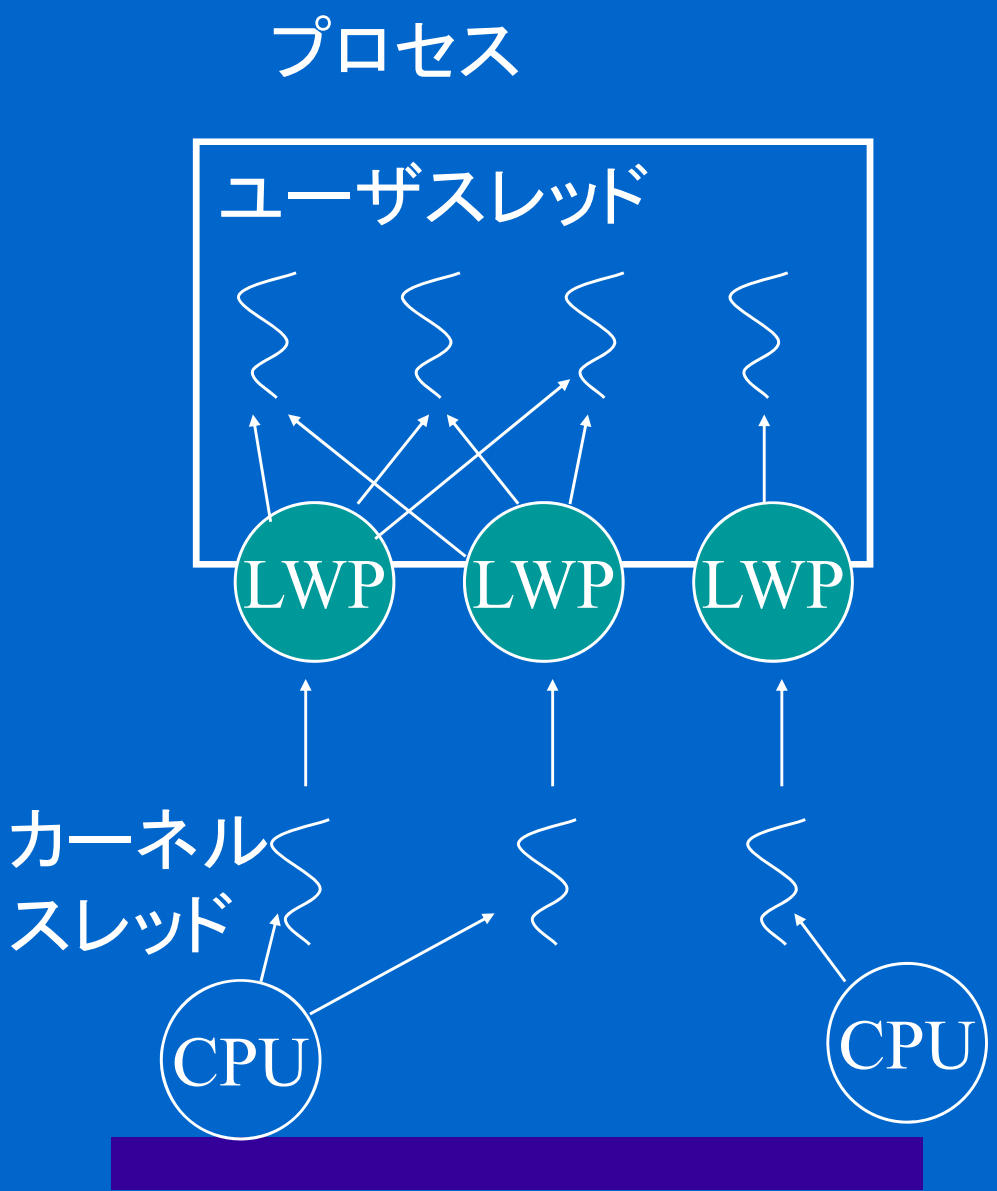
- ユーザスレッドの数
  ＞カーネルスレッドの数≧2
  - カーネルスレッドの数だけ並行に動く
- １個のユーザスレッドがブロックしても、他のユーザスレッドは動く
- カーネルスレッドの数と同じ数（左図では2個）のユーザスレッドがブロックすると、残りのユーザスレッドもブロックする

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 (2000,XP,Vista, 7)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
- プロセッサが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
―プロセスの管理の理論―

- 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）

2.2.2 プロセスの同期と相互排除

- 並行プロセスでは、次のような場合について並行プロセスの制御、すなわち**同期**が必要になる（図2.23）
  - 共有する資源（プログラム中での特定の領域やハードウェア装置など）の使用要求が競合する
  - プロセス間で通信する（プロセス間通信）
- ある1個の共有資源を複数プロセスが同時に使用しないように制御する同期機能を、**排他制御**または**相互排除**制御という（図2.24）

相互排除が必要な例

- 配列でスタックを実現したとする
- 共有メモリ上に置いて複数のプロセスで利用
- プロセス1:データをプッシュ
  - top = top ＋1;
  - stack(top) = item;
- プロセス2:データをポップ
  - item=stack(top)
  - top = top ー 1

プロセス切り替えの例

プロセス1

top = top + 1

stack(top)=item

時間の流れ→

プロセス2

item=stack(top);

top = top - 1;

横取りにより、プロセス1のスタック操作が、プロセス2の操作により分断される可能性がある

クリティカルセクション

- スタックのpush, pop操作
  1. 分割して実行されてはならない
  2. 実行するプロセスは一度には1個だけでないといけない
  - プログラム中で1., 2.の条件が必要な部分を、**クリティカルセクション**という（**臨界領域**、または**際どい部分**ということもある）
- **不可分操作**（atomic action）
  - 分割が許されない一連の実行操作
- **相互排除**（mutual exclusion）
  - 一つのプロセスしかクリティカルセクションには入れないようにする

不可分操作の例

- 操作がすべて完了するか、まったく行われないか、どちらかであることが保証されていること
  - 分割不可能な操作
- 例: mkdir
  - ディレクトリが作られているが . や ..が無いという状態にはならないことを保証

同期問題

- 資源を複数のプロセスが使用するためプロセスが待つ仕組みを作ること
- デッドロックや飢餓状態が起こらないように考慮する必要がある

デッドロックと飢餓状態

- **デッドロック (deadlock)**
  - 資源を持っているプロセスが他の資源を待ってブロックしたままになる
- **飢餓状態　(starvation)**
  - あるプロセスが待っている資源が、解放されるたびに、他のプロセスに取られてしまう
  - そのプロセスにはずっと割り当てられない

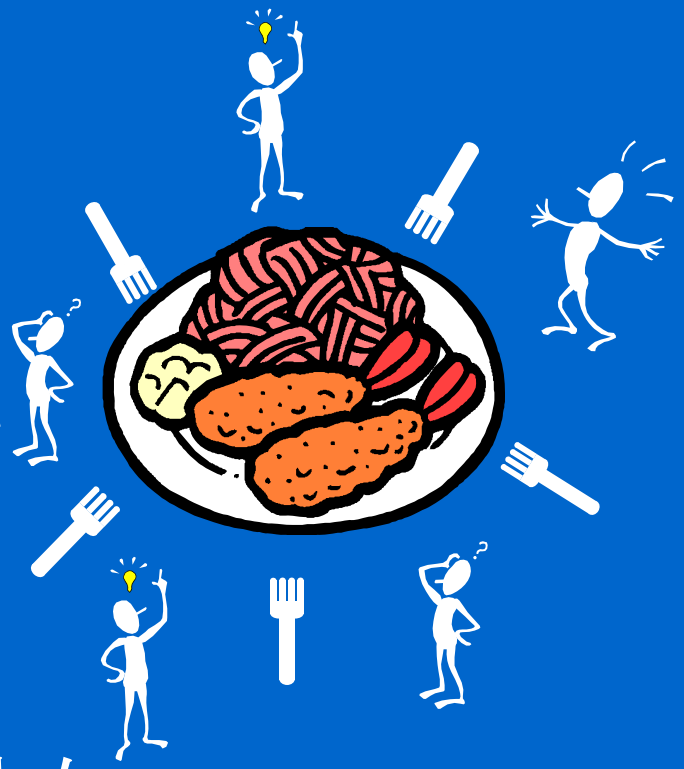
    →待ちっぱなし

哲学者の食事（dinning philosopher ） 問題

- 円卓に5人の哲学者
- 中央に食物、席と席の間にフォーク（最近の説明では「箸」）
- 食べる手順
  - 哲学者は片側のフォーク（または箸）を取る
  - 別の側のフォーク（または箸）を取る
  - 食べる
  - フォーク（または箸）を2本とも戻す
- 哲学者は行儀がよいのでフォーク（または箸）が2本揃わないと食事をしない（2本揃わないときは思索する）

哲学者の食事問題（つづき）

- 食べ終わるまでフォークは返さない
  →フォークを1本取ったら、
  　必ずもう1本を取って食べる
- ずっと食べ続けることはない
- ずっと思索し続けることもない
- 同期問題の例題
  – これでうまく食事できるか？
  – 問題があるとすればどういうことか



テストアンドセットによる相互排除

- テストアンドセット：1ビットのフラグによってクリティカルセクションに対する相互排除を実現する
- 不可分に行なうマシン命令（TS命令）を利用（プロセッサのクロックの不可分性を利用）
  - テスト：フラグの値が0（空き）か1（使用中）かチェック
  - セット：フラグを1（使用中）に設定

使用例:

```
LOCK:    TS ENTER
         BNZ   LOCK
         <critical section>
UNLOCK: MVI   ENTER, '00'
```

テストアンドセットの問題点

- busy waiting（プロセッサを割付けている状態で待つこと、spin lockともいう）になっている
- プロセッサ時間の無駄使い

- 長くは待たないことが明らかな場合に利用

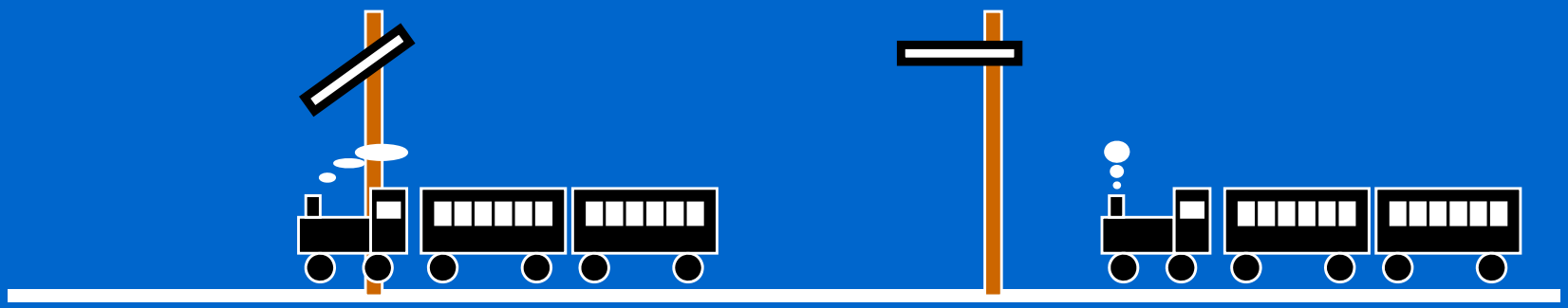
セマフォ

- 同期問題を解決する道具の一つ
  - 1968年E.W.Dijkstraが考案
- 整数型変数 とそれに対する手続き
- signal(semaphore)とwait(semaphore)操作
  - P(semaphore)とV(semaphore)操作と言う表記もある（教科書はこれで表記されている）
  - P: Paseren、V: Verhoog(オランダ語)
- セマフォ:　鉄道の腕木信号機

腕木信号機（図2.28）

進入可

進入不可



セマフォの種類

- 二進セマフォ(binary semaphore)
  - 0または1の値を取る
- 汎用セマフォ、計数セマフォ
  - 非負の値をとる

セマフォの操作（図2.29, 図2.30）

- wait(semaphore)
  - semaphoreの値が正なら値を1減らす
    - 次の文へ
  - 0なら、ブロック（待ち状態のキューにつながれる）
- signal(semaphore)
  - waitでブロックしているプロセスがあるとき
    - ブロックしているプロセスの実行を再開する（実際には、実行可能キューにつなぐ）
  - ないとき
    - semaphoreの値を1増やす

セマフォ:実際の操作

- signal(semaohore)
  - waitでブロックしているプロセスが居ても、この時点で切り替わるわけではない
- 動作可能になるだけ(単一プロセッサの場合)
  - 次に何かのタイミングでプロセスが切り替わったときに、動く
- マルチプロセッサで、複数のプロセッサの競合回避にセマフォを使っている場合
  - signalの時点で、ブロックしているプロセスが動くことがありえる

セマフォを用いたスタック操作

- semaphoreの初期値は1
- push操作

```
wait(semaphore);
top=top-1;
stack(top)=item;
signal(semaphore);
```

pop操作

```
wait(semaphore);
item=stack(top);
top=top+1;
signal(semaphore);
```

生産者消費者問題

- 生産者
  - データを生成しバッファに入れる
  - バッファがいっぱいなら
    - 消費されるのを待つ
- 消費者
  - バッファ内のデータを消費する(処理する)
  - データがなければ待つ
- バッファにはN個のデータが入る

生産者消費者問題のプログラミング例

```
/* プロセス1（生産者） */
for(;;){
product =送信データ;
wait(write);
put(buffer, product);
signal(read);
}
```

```
/* プロセス2（消費者） */
for(;;) {
wait(read);
get(buffer, product);
signal(write);
productを読み出す;
}
```

- writeの初期値：バッファのサイズ
- readの初期値：0

例題1

```
/* プロセス1（生産者）*/
product =送信データ;
wait(write);
buffer=product;
signal(read);
```

```
/* プロセス2（消費者）*/
wait(read);
product = buffer;
signal(write);
productを読み出す;
```

writeとreadはセマフォであり、bufferはプロセス1とプロセス2で共有されている変数とする（初期値は、write=1, read=0）

各文はどのような順番で実行されるか？（実行可能キューの先頭にプロセス2, その次にプロセス1がつながれていたとする）

例題1の解答例

/* プロセス1（生産者） */

②product =送信データ;

③wait(write);

④buffer=product;

⑤signal(read);

/* プロセス2（消費者） */

①wait(read);

⑥product = buffer;

⑦signal(write);

⑧productを読み出す;

- プロセス2の①を実行（readの値が0のため、wait操作によりプロセス2はブロック）。プロセス1に切り替わる
- プロセス1の②、③を実行（writeの値が1のため、writeを0にして実行継続）。さらに、④、⑤を実行（signal操作により、プロセス2を実行可能キューにつなぐ）。プロセス1は終了
- プロセス2の⑥から実行を再開。⑦を実行（writeの値を1にする）。さらに、⑧を実行して終了

例題2

```
/* プロセス1（生産者）*/
product =送信データ;
wait(read);
buffer=product;
signal(write);
```

```
/* プロセス2（消費者）*/
wait(read);
product = buffer;
signal(write);
productを読み出す;
```

例題1のプログラムで、プロセス1のreadとwriteを逆に書いたらどうなるか？

例題2の解答例

/* プロセス1（生産者） */
②product =送信データ;
③wait(**read**);
buffer=product;
signal(**write**);

/* プロセス2（消費者） */
①wait(read);
product = buffer;
signal(write);
productを読み出す;

- プロセス2の①を実行（readの値が0のため、プロセス2はブロック）。プロセス1に切り替わる
- プロセス1の②、③を実行（readの値が0のため、プロセス1はブロック）
- 両方のプロセスがブロックしてしまい、プロセスの実行は停止（**デッドロック**）

セマフォについて

- 基本的な同期機構
- 多くのOSに備わる
- signal/waitを正しく使わなければならない
  - さもないと相互排除失敗orデッドロック
- signalとwaitがプログラムの中に分散
  - 対応関係が正しいか検証が難しい
- より良い相互排除機構の模索
  - モニタ、メッセージによる同期

基本同期命令 (synchronization primitive)

- プロセス間で同期を取るための基本的な命令
- 不可分な操作である
  - 必ずしも1つの機械語命令ではない
- 相互排除の要件
  - 相互実行（mutual execution）を禁止する
  - デッドロック(deadlock)を禁止する

単一プロセッサでのセマフォ

- もっとも簡単な基本同期命令
- 単一プロセッサであれば、割り込みを禁止すれば、横取りは起こらない(プロセスの実行が中断されない)
- 割り込み禁止は手間がかからない
  - 通常1命令
- ユーザプロセスで割り込みを禁止するには問題がある
  - 誤って禁止した場合に、システムを保護できない
- 応用
  - 利用したい資源に関連する割り込みだけ禁止

マルチプロセッサとセマフォ

- UNIXの相互排除
  - 相互排除操作をシステムコールで実現
- 初期のUNIXのマルチプロセッサ対応
  - システムコールを実行できるプロセッサを1個だけに限定（カーネルを実行できるプロセスを同時には1個だけにすることで、相互排除操作が不可分であることを実現）
- マルチプロセッサ対応UNIXでは、複数のプロセッサが同時にシステムコールを実行可能
  - カーネルの機能だけでは不可分な操作であることが保証できない（原則としてハードウェアによりサポート）