オペレーティングシステム

資料　第5分冊(2021)

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

2

プロセスのスケジューリング

- **横取りなし**スケジューリング（前回の講義で説明）
  - 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
  - 例： FCFS（到着順）、SJF（最短要求時間順）
- **横取りあり**スケジューリング（今回の講義で説明）
  - 実行中状態のプロセスが、ブロックや終了以外で、実行可能状態に遷移する（プロセスの**プロセッサへの割り付けを「横取り」する**）ことがある
  - 例： ラウンドロビン、SRT（最小残余時間順）、優先度順

今回の講義では、前回に引き続き、プロセスのスケジューリングについて説明します。

スケジューリングには、「横取りなし」と「横取りあり」の2種類があります。

横取りなしスケジューリングとは、前回の講義で説明したように、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、実行が終了しない限り、別の状態に遷移しないというものです。例としてはFCFSやSJFがありました。

横取りありスケジューリングでは、今回の講義で詳しく説明しますが、実行中状態のプロセスが、ブロックや終了以外で実行可能状態に遷移する（プロセスの**プロセッサへの割り付けを「横取り」する**）ことがあります。

例としては、ラウンドロビン、SRT（最小残余時間順）、優先度順があります。

3

ラウンドロビン

- 横取りがあることを除けば、FCFSと同じ
- 実行中のプロセスを、一定時間ごとに横取り（図2.18）
  - 横取りが起こると、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
  - 例えば、プロセス1、プロセス2、プロセス3がこの順番に生成されたとすると、それらの実行は一定時間ごとに、1, 2, 3, 1, 2, 3, …と繰り返されていきます
  - 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

ラウンドロビンについて説明します。

ラウンドロビンは、横取りがあることを除けば、FCFS（到着順）のスケジューリングと同じです。

プロセスは、到着順に実行可能キューにつながれ、ディスパッチにより実行中の状態に遷移して実行されます。

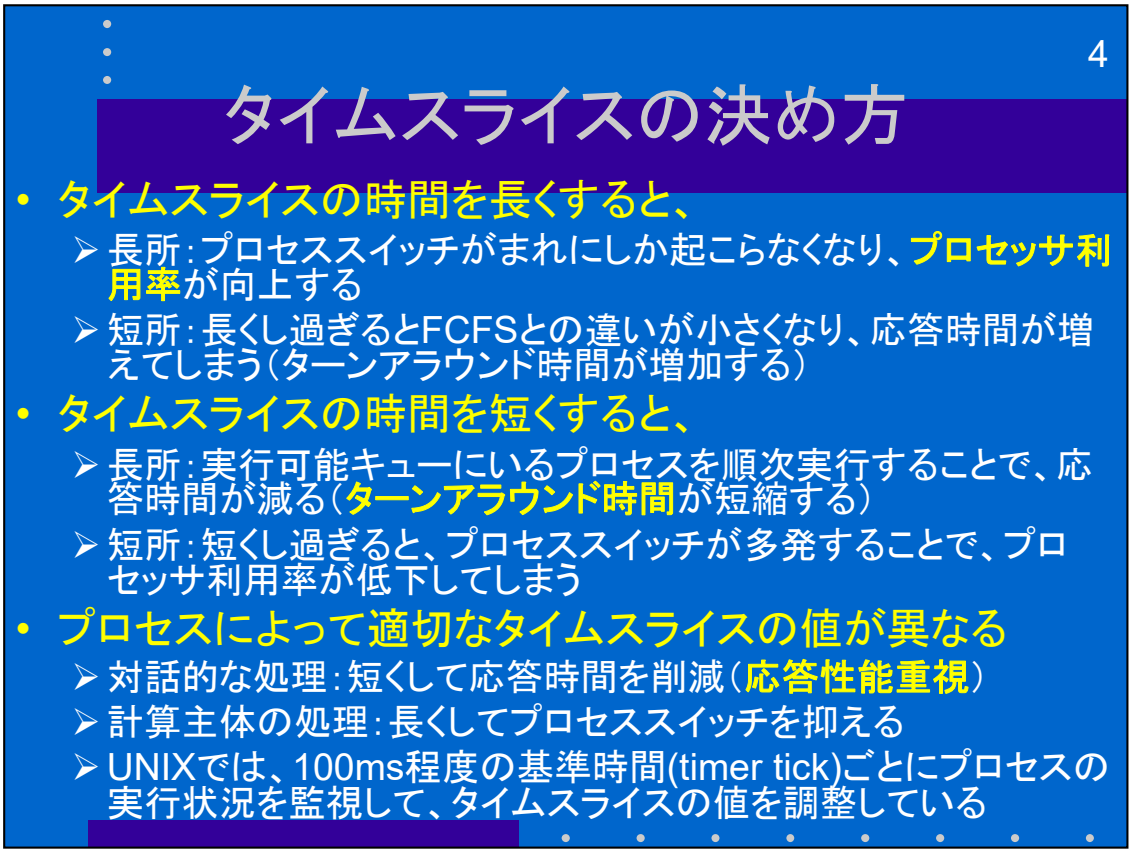
ここまではFCFSと同じですが、実行中のプロセスに対して、一定時間ごとに横取りが起こるところがFCFSと異なります。

横取りが起こると、その時点で実行中のプロセスは、実行可能キューの末尾へつながれます。

例えば、プロセス1、プロセス2、プロセス3がこの順番に生成され、実行可能キューに到着したとすると、それらの実行は一定時間ごとに、1, 2, 3, 1, 2, 3, …と繰り返されていきます。

このように順番に巡回することからラウンドロビンと呼ばれます。

このとき、横取りが起こるまでの「一定時間」のことをタイムスライスと呼びます。



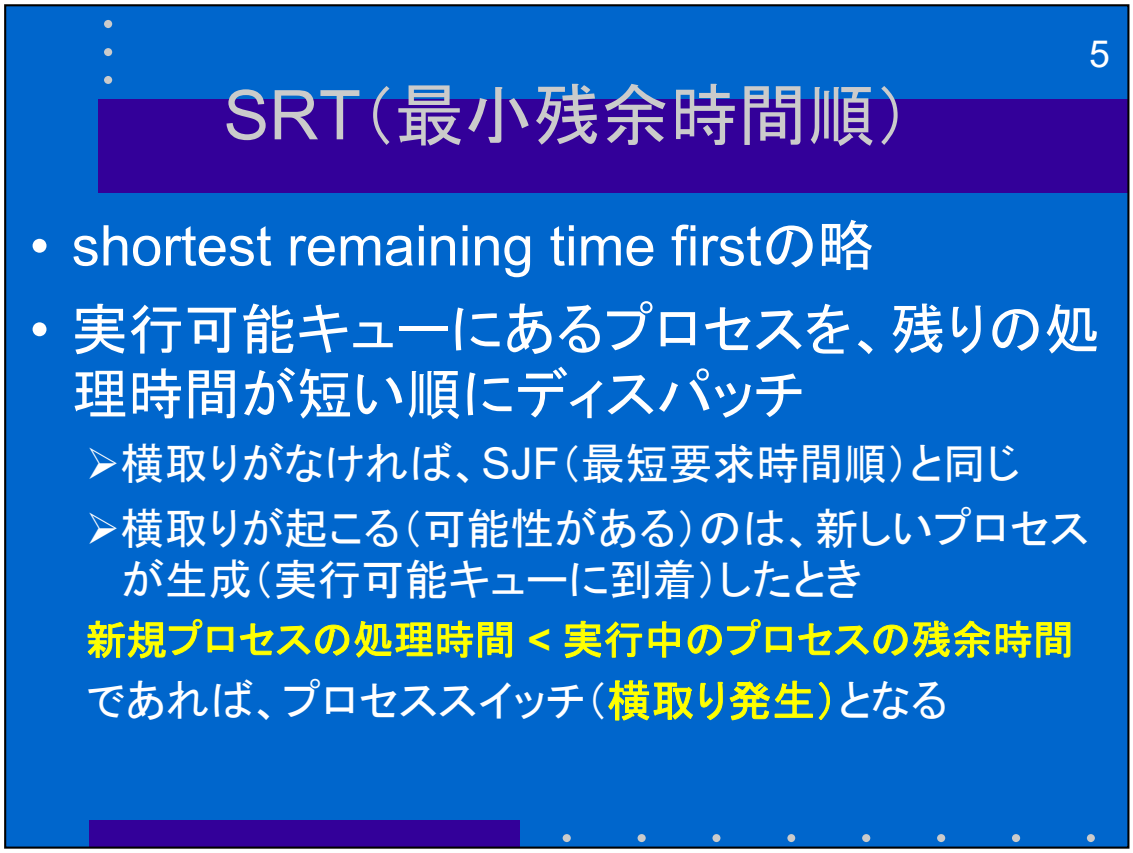
ラウンドロビンのスケジューリングは、タイムスライスの長さによって影響を受けます。

タイムスライスの時間を長くする長所としては、プロセススイッチがまれにしか起こらなくなり、削減されたプロセススイッチにかかる時間分だけ、プロセッサ利用率（全体の時間の中でプロセスの実行が占める時間の割合）が向上します。短所としては、長くし過ぎるとFCFSとの違いが小さくなり、応答時間が増える（ターンアラウンド時間が増加する）ことがあります。

タイムスライスの時間を短くする長所としては、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）ことです。短所としては、短くし過ぎると、プロセススイッチが多発することで、プロセススイッチに時間が取られることになる、プロセッサ利用率が低下してしまうことです。

結論としては、プロセス内の処理の種類によって適切なタイムスライスの値が異なるということです。対話的な処理の多いプロセスでは、タイムスライスを短くして応答時間を削減し、応答性能を重視することが有効です。計算主体の処理の多いプロセスでは、タイムスライスを長くしてプロセススイッチを抑えることが望ましいです。

参考までに、UNIXでは、100ms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ます



次に、SRTのスケジューリングについて説明します。

これは、shortest remaining time first（最小残余時間順）の略です。

つまり、実行可能キューにあるプロセスを、残りの処理時間が短い順にディスパッチするというものです。

SRTは、横取りが起こらなければSJF（最短要求時間順）と同じスケジューリングとなります。

ではどういう時に、横取りが起こるかというと、新しいプロセスが生成された、つまり、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（終了までの残り時間）と比較して、新規プロセスの方が処理時間が短ければ、横取りが発生し、実行中のプロセスは実行可能キューに残余時間順につながれ、新規プロセスに実行が切り替わります。

6

プロセススケジューリングの例題(1)

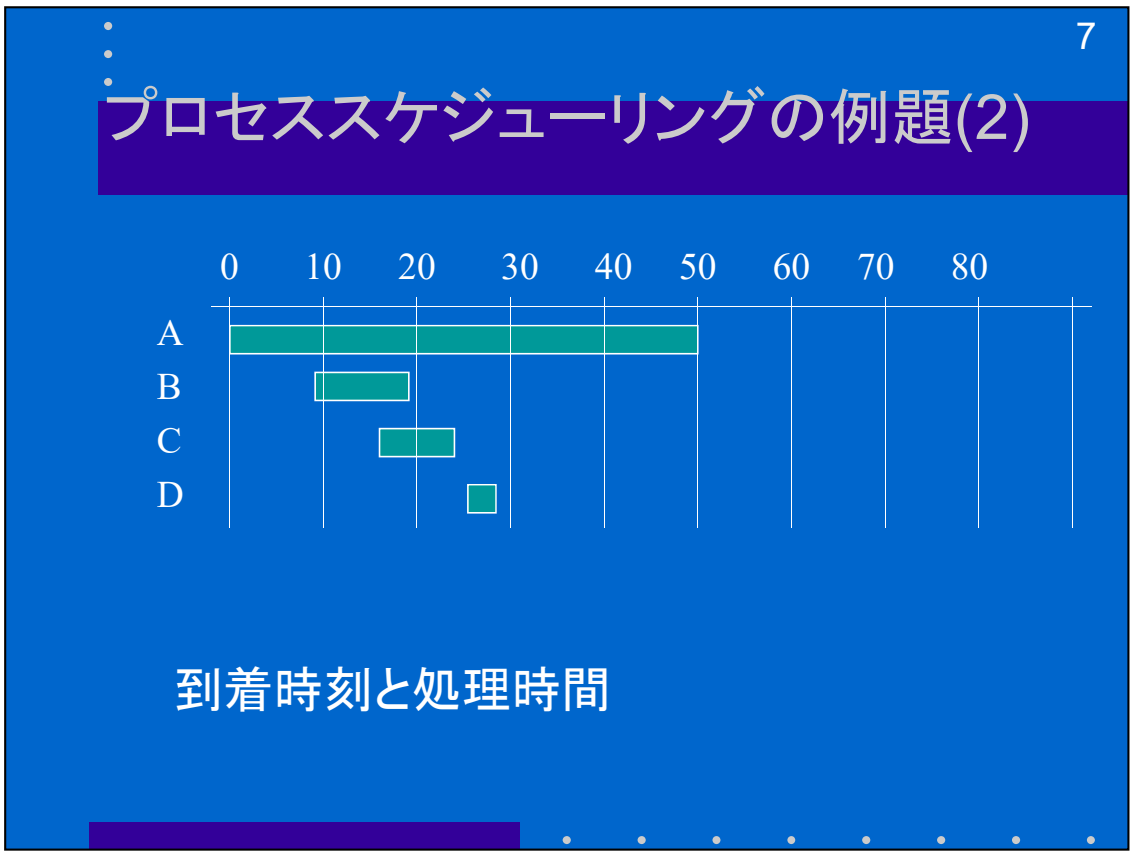
- 4つのプロセス
  A: 到着時刻 0, 処理時間 50
  B: 到着時刻 9, 処理時間 10
  C: 到着時刻16, 処理時間 8
  D: 到着時刻 26, 処理時間 3
- 処理はどのように進むか（プロセッサは1個だけ）
  FCFS, SJF, SRT, ラウンドロビン（タイムスライス=10）
- それぞれの場合について考える

これまでに説明した、FCFS SJF, SRT, ラウンドロビンのスケジューリングでのプロセスの実行例をそれぞれ見て行きます。

ただし、プロセッサは1個だけとします。

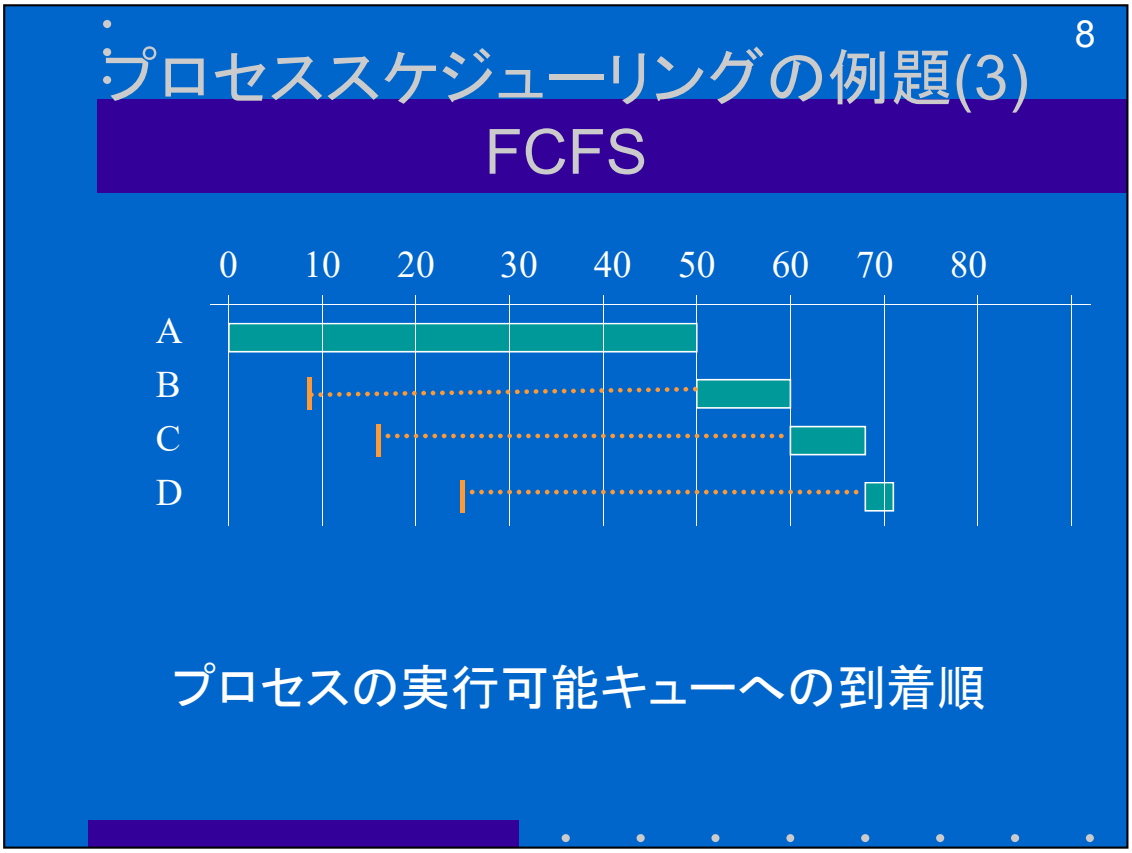
ラウンドロビンは、タイムスライスの値によってスケジューリングが変わりますので、この例ではタイムスライスの値を10とします。

そこで、到着時刻と処理時間の異なる4個のプロセスについて、これらのスケジューリングで実行するとどうなるか考えてみましょう。



この図は、4個のプロセスの到着時刻と処理時間を単純に時間軸上に置いたものです。

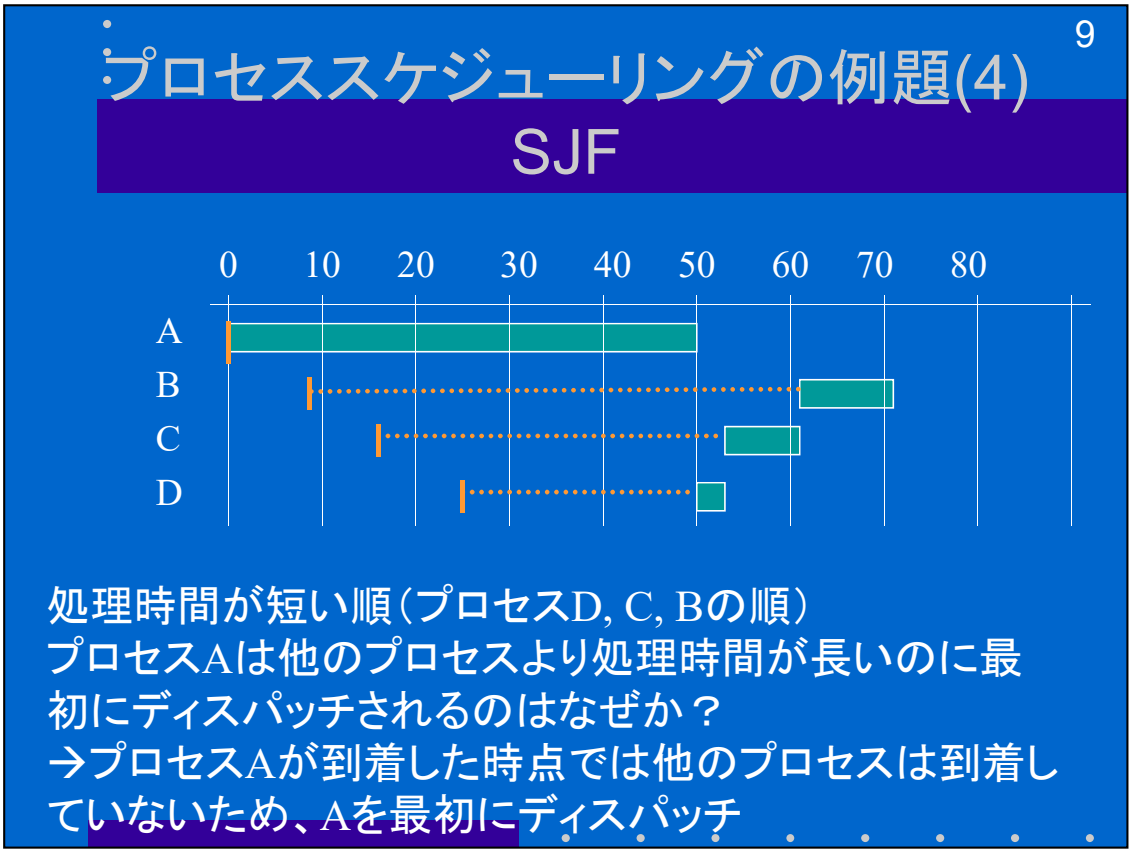
プロセッサが4個以上あれば、このままで実行できますが、ここでは、プロセッサが1個しかないときですので、処理時間を重ねることができません。



FCFS（到着順）では、プロセスが実行可能キューに到着した順に単純に実行されます。

横取りは全く発生しません。

この例のように、処理時間が長いプロセスが最初にあると、以降に到着したプロセスは、最初のプロセスの実行が終わるまで待たされることになります。

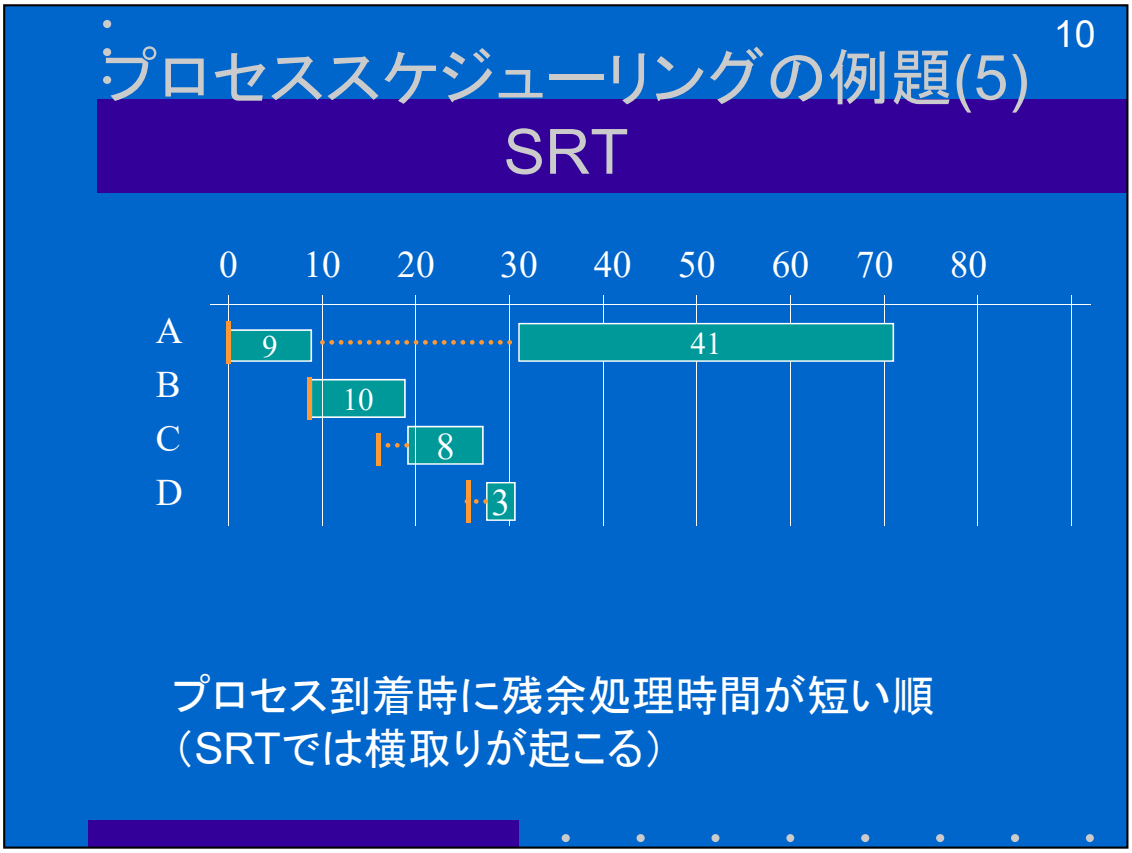


SJFでは、実行可能キューで処理時間が短い順にプロセスが実行されます。

プロセスAは、他のプロセスより処理時間が長いのに最初にディスパッチされることに注意してください。

これは、プロセスAが到着した時点では他のプロセスが到着していないためです。

プロセスプロセスB, C, Dについては、処理時間の短いD, C, Bの順にディスパッチされます。



SRTは、SJFの横取りあり版となります。

SJFと同様に、最初はプロセスAが実行されます。

プロセスBが到着した時点で、「プロセスBの処理時間(10) ＜ プロセスAの残余時間(41)」であるため、横取りが発生し、プロセスAは実行可能キューに移され、プロセスBがディスパッチされます。

プロセスCが到着した時点で、「プロセスCの処理時間(8) ＞ プロセスBの残余時間(3)」であるため、横取りは発生しません。

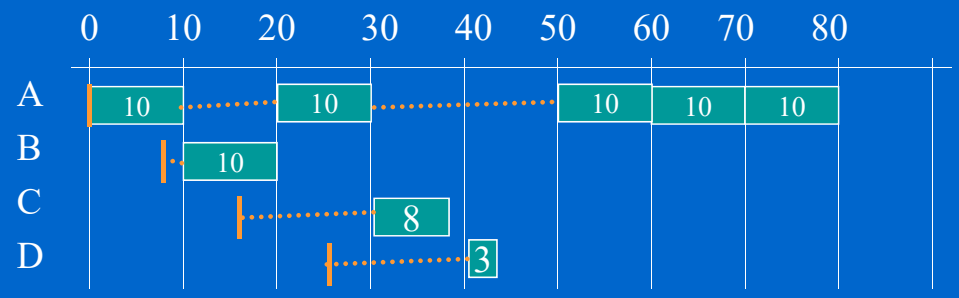
プロセスBが終了した時点で、実行可能キューにはプロセスA（残余時間41）、プロセスC（処理時間8）があり、残余時間の短いプロセスCがディスパッチされます。

プロセスDが到着した時点で、「プロセスDの処理時間(3) ＞ プロセスCの残余時間(1)」であるため、横取りは発生しません。

プロセスCが終了した時点で、実行可能キューにはプロセスA（残余時間41）しかいないので、プロセスAがディスパッチされ、継続して実行されます。

11

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
- 時刻20でプロセスCではなくプロセスAをディスパッチすることに注意

→時刻10で横取りが発生した時点で、まだプロセスCが到着していないため、時刻20で実行可能キューの先頭はプロセスAであるため

ラウンドロビンのスケジューリングについて示します。

ラウンドロビンでは、FCFSの横取りあり版であるので、まずは到着順でプロセスAが最初にディスパッチされます。

ここでタイムスライスを10としているので、時刻10で横取りが発生し、プロセスAは実行可能キューにつながれ、プロセスBが代わりにディスパッチされます。

時刻20で次のタイムスライスが来て、また横取りが発生しますが、このときディスパッチされるのはプロセスCではなくプロセスAとなることに注意します。

これは、時刻10ではまだプロセスCが到着していないため、プロセスAが実行可能キューの先頭となり、その後に到着したプロセスCはその後につなげられるためです。

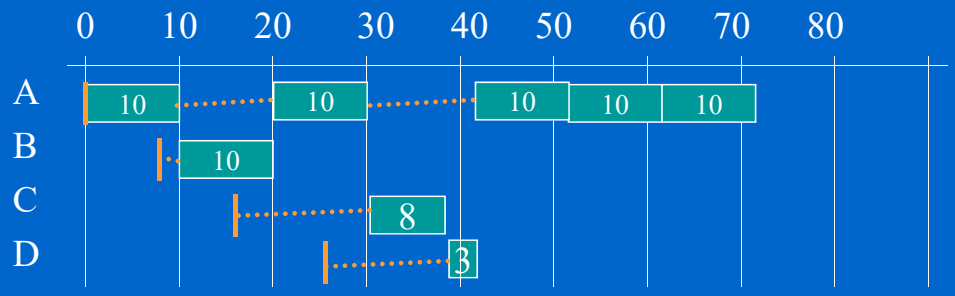
時刻30でも横取りが発生し、今度はプロセスCがディスパッチされます。

時刻40でも横取りが発生し、プロセスDがディスパッチされます（プロセスAは時刻30で実行可能キューにつながれますが、プロセスDは時刻30より前に到着しているため、プロセスDがディスパッチされます）

時刻50以降でも、タイムスライスごとに実行可能キューがチェックされますが、キューにつながれているプロセスがないため、横取りは起こらずにそのままプロセスAがディスパッチされます。

12

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、すぐに次のプロセスをディスパッチする場合

前のスライドで示したラウンドロビンでのスケジューリングでは、タイムスライスの値の倍数の時刻ごとに横取りが発生していました。

しかし、プロセスCとプロセスDは処理時間が10より小さいため、実行されると、次のタイムスライスまで待たずに実行が終了してしまいます。

このとき、次のタイムスライスの時刻が来るまで、プロセスが実行されず、プロセッサが空いてしまうことになります。

これを避けるには、プロセスが終了した時には、次のタイムスライスの時刻までまたずに、すぐに実行可能キューの先頭のプロセスをディスパッチすることが考えられます。

このときには、タイムスライスは、次のプロセスをディスパッチしてから新たに10ずつ数えられることになります。

13

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
  - SJFやSRTは、それぞれ処理時間、残余時間の短いプロセスに高い優先度をつけた優先度順とみなせる

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、特定のプロセスが永久に実行されないことを指す
  - 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)
  - 実行可能キューに長時間つながれているプロセスについては、優先度を時間経過につれて徐々に高くしていく

プロセスのスケジューリングでは、実行可能キューにあるプロセスに優先度をつけて、優先度の高いものからディスパッチしたい場合があります。

これが優先度順のスケジューリングです。

これまでに説明した、SJFやSRTは、それぞれ処理時間、残余時間の短いプロセスに高い優先度をつけた優先度順とみなすことができます。

ここで注意したいのが、優先度順は公平なスケジューリングではないため、実行順に偏りが生じるということです。

偏りの極端な例が、無限のブロックです。

これは、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、特定のプロセスが永久に実行されないことを指します。

例えば、たまたまあるプロセスに非常に低い優先度を付けてしまった後で、それ以降で生成されるプロセスがすべて最初のプロセスよりも優先度が高ければ、最初のプロセスは永久に実行されないことが起こりえます。

これが無限のブロックまたは飢餓と呼ばれる状態です。

これを解決する方法には、エージングと呼ばれるものがあります。

これは、実行可能キューに長時間つながれているプロセスについて、優先度を時間経過につれて徐々に高くしていくというものです。

これにより、永久にブロックされることはなく、いずれはディスパッチされて実行されることになります。

14

多重レベルスケジューリング(1)

- 優先度ごと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
  - 長時間実行されるプロセスは優先度を低くする
- 飢餓状態を起こす可能性がある

優先度順を拡張したのが、多重レベルスケジューリングです。

このスケジューリングでは、優先度ごとに実行可能キューを作成します。

プロセスのディスパッチでは、まず最大優先度のキューをみて、その先頭のプロセスをディスパッチし、なければ次の優先度のキューを見に行き、そこにもなければその次の優先度という形で進みます。

これをさらに発展させてものが、多重レベルフィードバックスケジューリングです。

このスケジューリングでは、プロセスの実行可能キュー間の移動を行います。

新規のプロセスは、まず最大優先度のキューにつながれ、ディスパッチされてタイムスライスを使い切ると一つ低い優先度へ移動されます。

これにより、長時間実行されるプロセスは優先度を低くすることができます。

しかし、長時間実行で優先度を落とすということは、飢餓状態または無限のブロックを起こす可能性があるので、注意が必要です。

15

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（特定のプロセスによるプロセッサの占有を抑える）
- UNIXの実装
  - 優先度値=基本優先度　＋　直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**

飢餓状態を避けるためには、先に説明したエージングの他に、プロセッサを多く消費するプロセスは優先度を下げて、特定のプロセスによるプロセッサの占有を抑えるという方法があります。

なお、UNIXでのスケジューリングの実装では、このような手法が取り入れられており、直近のプロセッサ消費量が大きいと、プロセスの優先度が低くなるようになっています。

また、ユーザの指定により、自分が生成した、特定のプロセスの優先度を下げることができるようになっています。

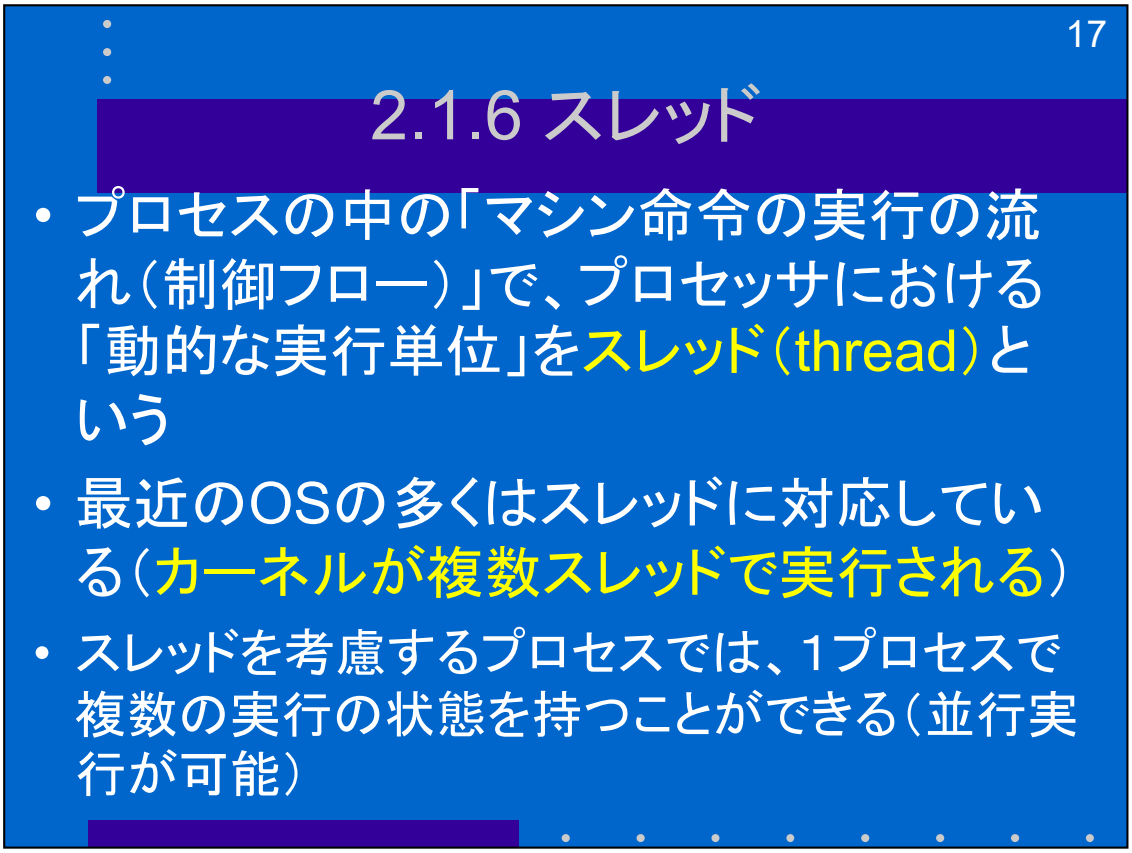
（プロセスの優先度を「上げる」ことができるのはスーパーユーザのみです）

16

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

以上説明した、プロセスのスケジューリングについてまとめるとこの表のようになります。

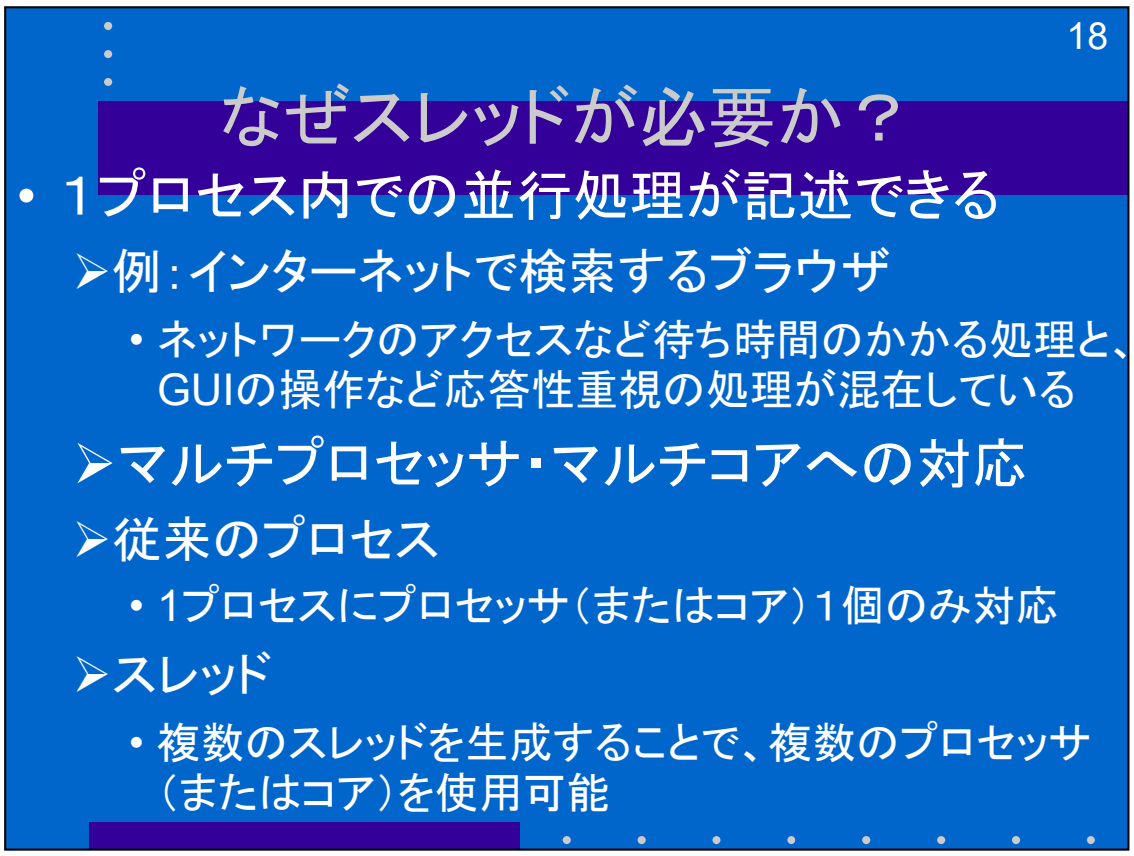


ここからは、スレッドについて説明します。

スレッドとは、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」を指します。
プロセスの他に、スレッドが必要な理由については、次のスライドで説明します。

最近のOSの多くはスレッドに対応しています。これは、OSのカーネルが複数スレッドで実行されるようになっていることを意味しています。

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、つまり並行実行が可能となります。

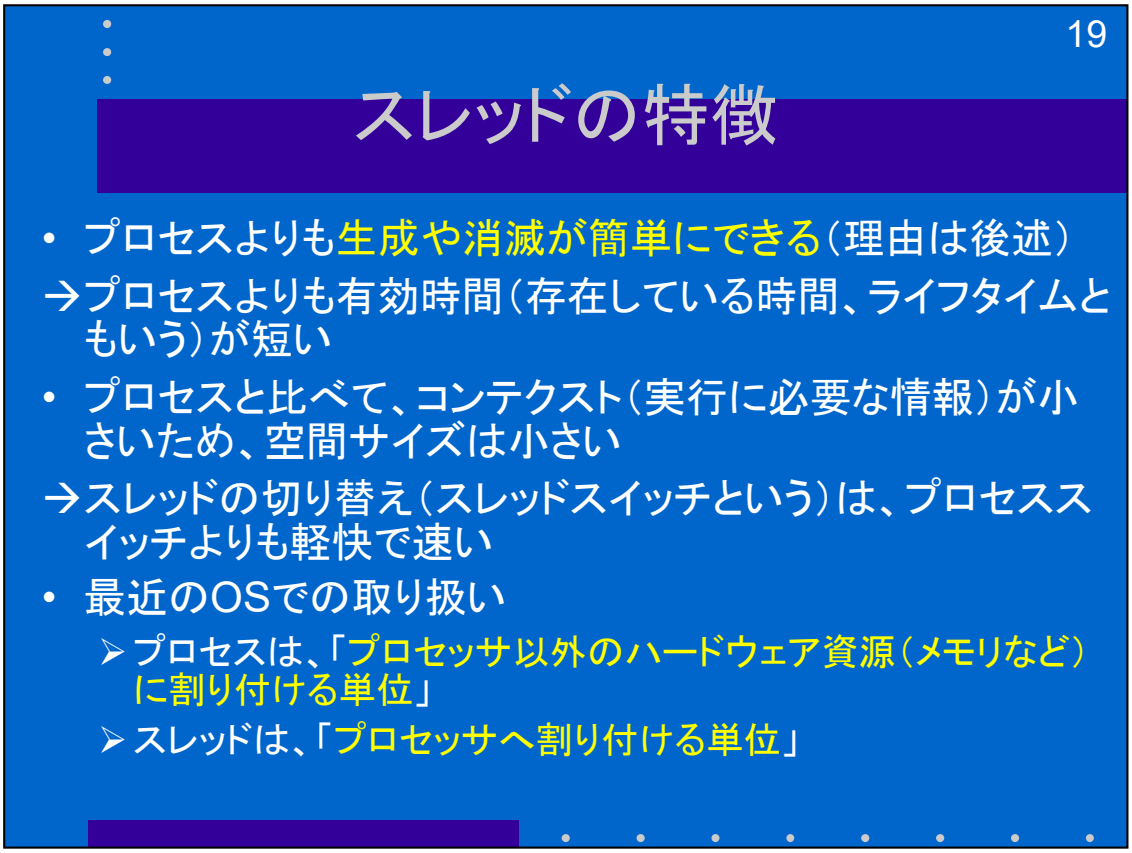


なぜ、スレッドが必要なのでしょうか？
プロセスだけでは不十分なのでしょうか？

この理由の一つに、最近のOSのユーザインタフェースの高度化があげられます。

インターネットで検索するブラウザだと、ネットワークのアクセスなど待ち時間のかかる処理と、GUIの操作など応答性重視の処理が混在していますが、これらはタイミングが異なるため、別々に処理できる方が管理が容易になります。

また、マルチプロセッサ・マルチコアへの対応があります。
従来のプロセスでは、1プロセスにプロセッサ（またはコア）1個のみ対応づけていました。
スレッドがあると、複数のスレッドを生成することで、1プロセスの中で複数のプロセッサ（またはコア）を使用可能となります。



スレッドの特徴をもう少し詳しく見て行きます。

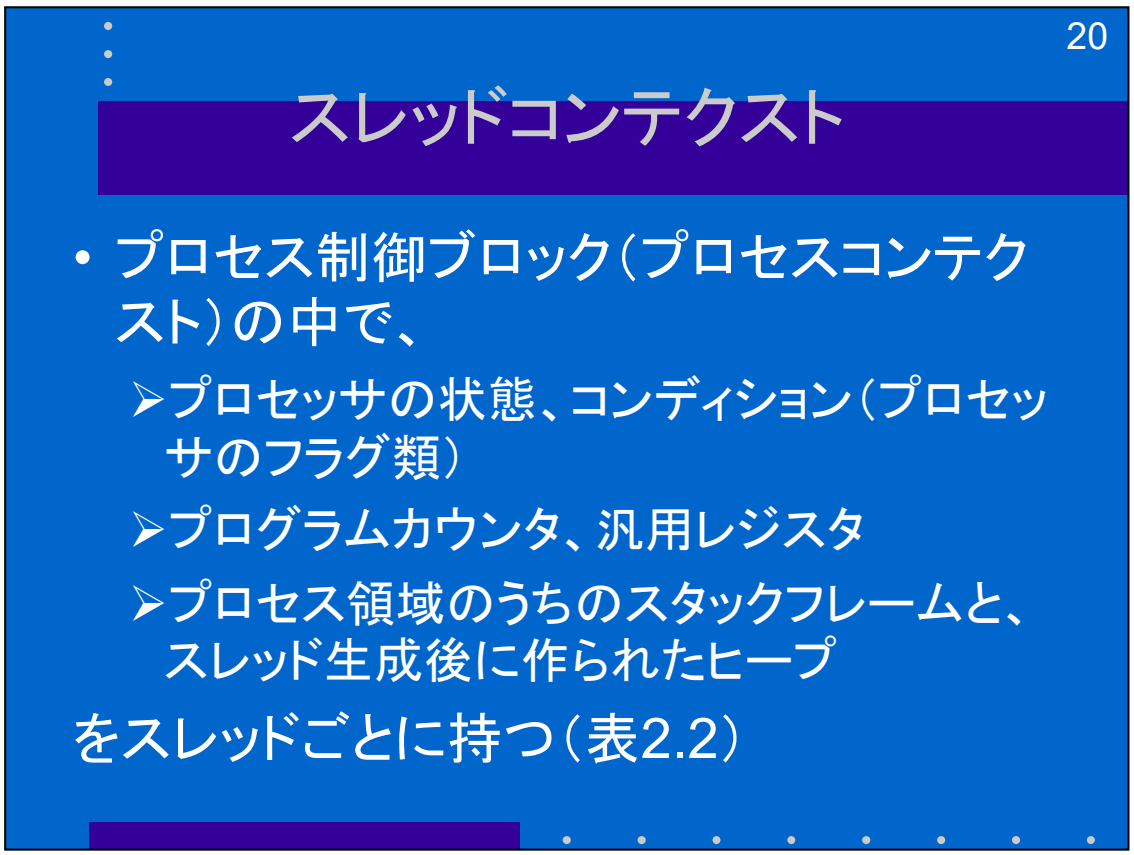
スレッドは、プロセスよりも生成や消滅が簡単にできるということがあります（理由は後述）

このことから、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）が短くなります。

また、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、空間サイズは小さくできます。

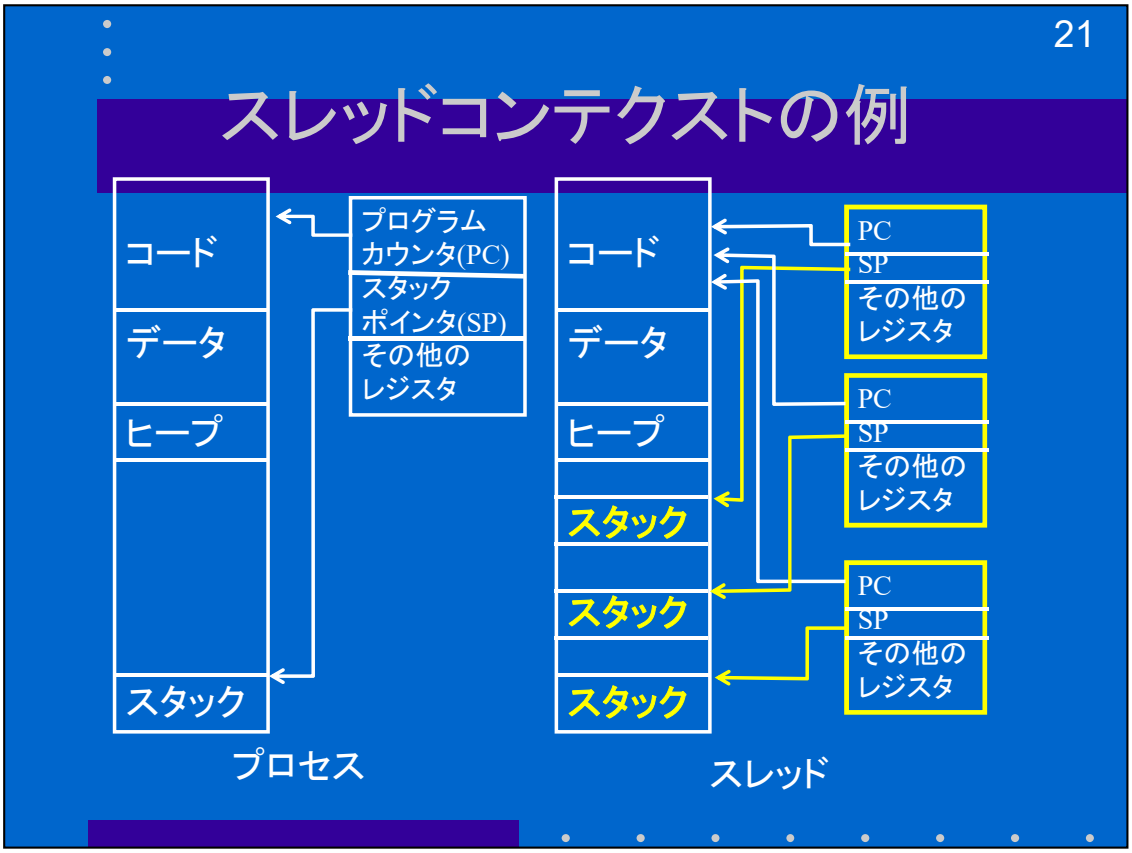
このため、スレッドの切り替え（スレッドスイッチという）は、プロセススイッチよりも軽快で速くなります。

最近のOSでは、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」ととらえ、スレッドは、「プロセッサへ割り付ける単位」ととらえるようになってきています。



スレッドの実行に必要な領域である、スレッドコンテクストについて見て行きます。

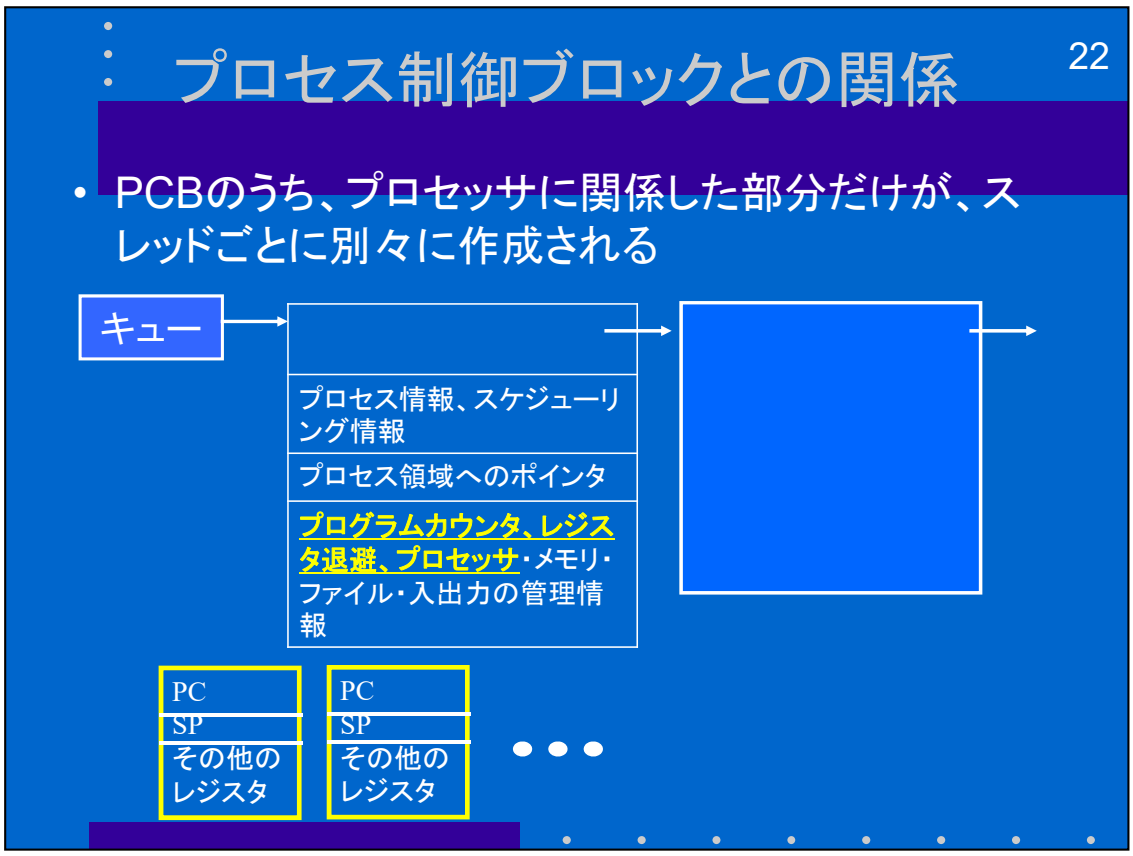
スレッドコンテクストは、プロセス制御ブロック中で、このスライドにあげられているような領域を、スレッドごとに持ちます。



スレッドコンテクストを図示すると、このスライドのようになります。

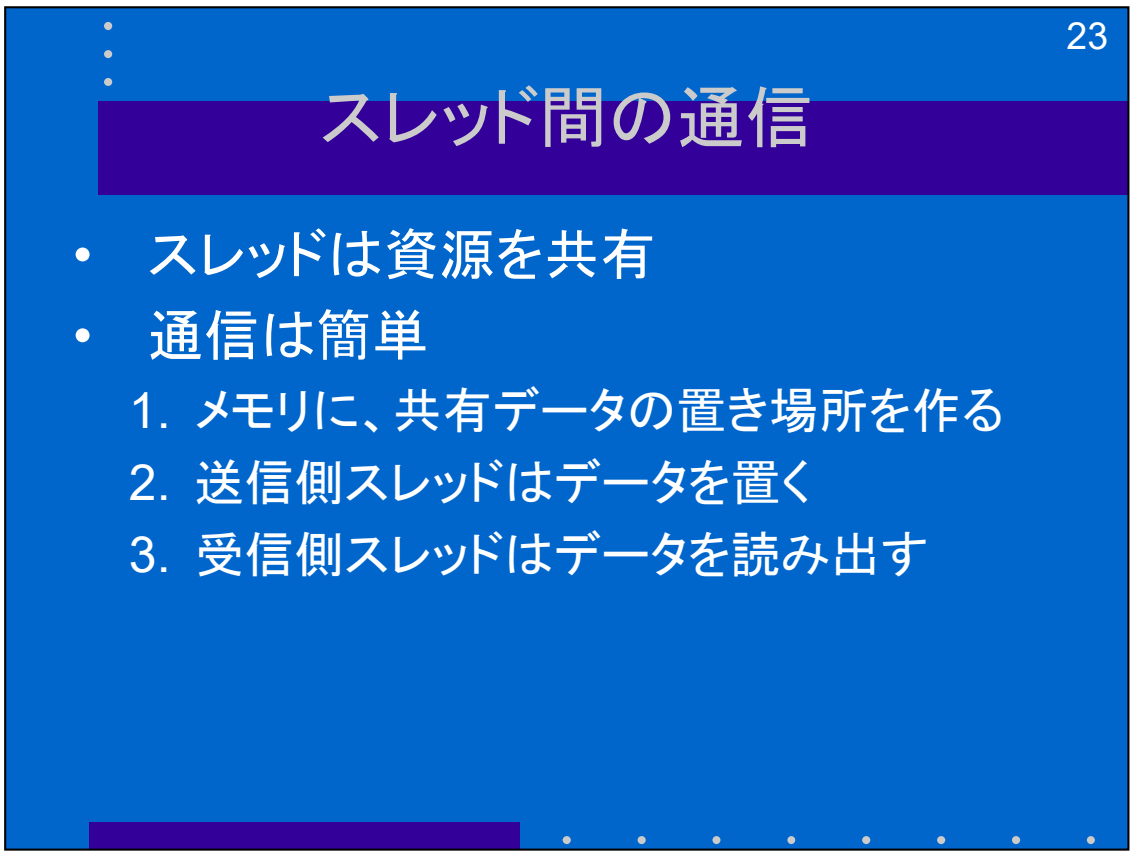
左側は、これまでのプロセスコンテクストを表し、右側の黄色い四角で囲まれた領域がスレッドコンテクストとなります。

以前の回の講義でも説明したように、プロセス領域のうち、スタックは共有できないので、スレッドごとに別々に領域が作られます。



スレッドコンテクストとプロセス制御ブロック(PCB)との関係を詳しく見たのがこの図です。

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ます。



スレッド間の通信は比較的簡単です。

スライド21に書いたように、スレッドはデータ領域を共有しているため、通信のための領域を決めて、送信スレッドはそこにデータを書き、受診スレッドはそこからデータを読み出すだけですみます。

24

カーネルでのスレッドの処理

1プロセス1スレッドのとき

- プロセスが入出力処理等の事象待ちシステムコールを実行
  - そのプロセスはブロックして、別のプロセスをディスパッチ

1プロセス複数スレッドのとき

- あるスレッドが事象待ちシステムコールを実行
  - 1スレッドのときのようにプロセスをブロックするとプロセス内の全スレッドがブロックしてしまう

→カーネルでもスレッド単位の実行管理が必要（カーネルネルスレッド）

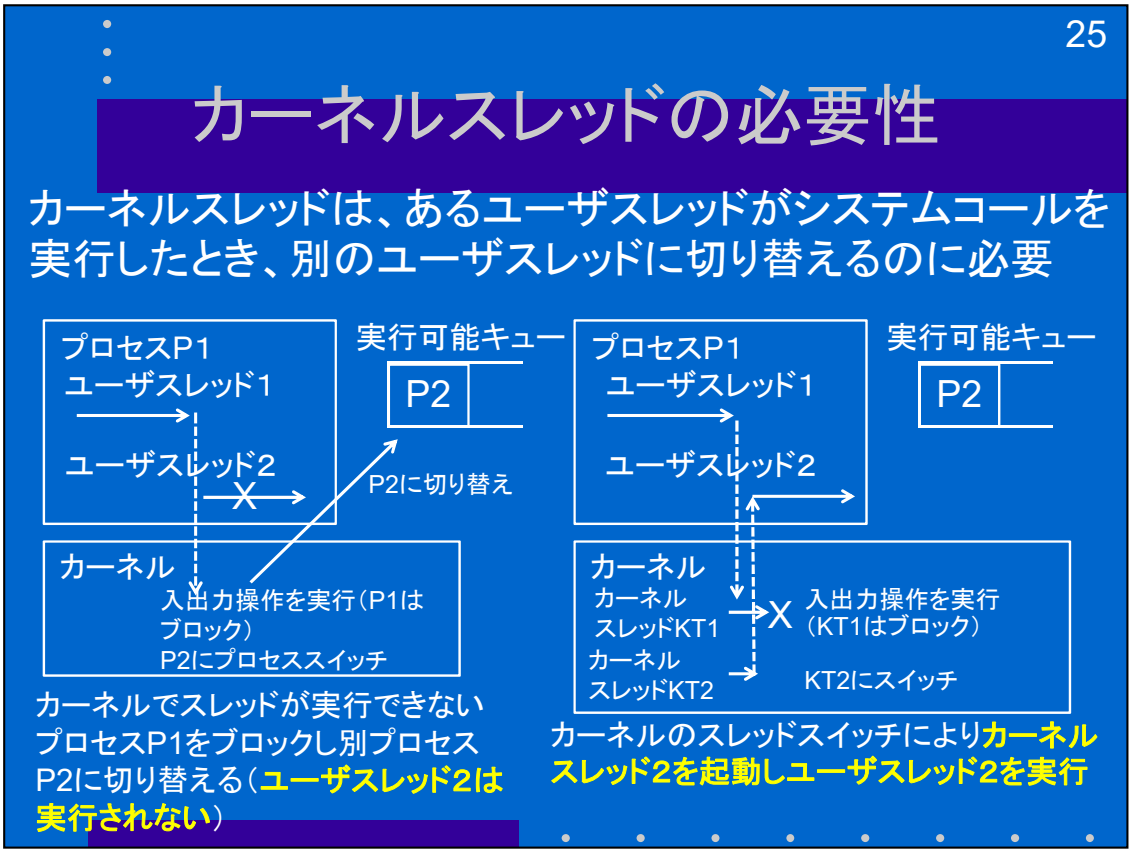
スレッドは、1つのプロセス内で簡単に並行実行ができるので便利ですが、スレッドがカーネルを呼び出すときは注意が必要です。

ここでプロセスが入出力処理等の事象待ちシステムコールを実行したときのことを思い返してみると、そのプロセスはブロックして、別のプロセスがディスパッチされることになっていました。

1つのプロセスに1つのスレッドしかないときはそれでかまわないのですが、同一プロセス内で複数のスレッドが実行されているとき、あるスレッドが事象待ちシステムコールを実行したからと言って、1スレッドのときのようにプロセスをブロックしてしまうと、そのプロセス内の全プロセスがブロックしてしまうことになります。

これでは、例えば、ブラウザの操作でネットワークにアクセスしているときは、他の操作が一切できなくなり、スレッドを作る意味がなくなります。

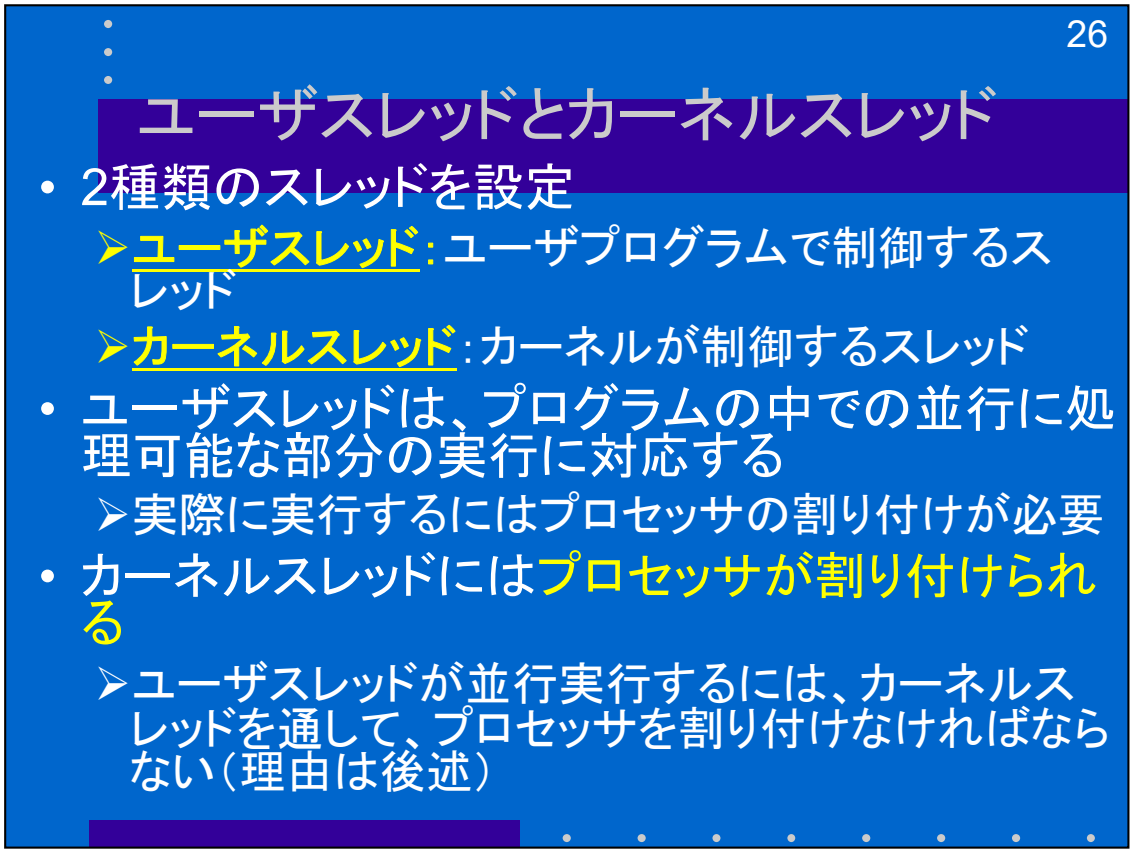
この問題を解決するには、ユーザプロセスだけでなくカーネルでもスレッド単位の実行管理が必要になります。これをカーネルスレッドと呼びます。



カーネルスレッドの動作を図で表すと、この図のようになります。

まず、左側はカーネルにスレッドがない状況で、今まで説明してきたプロセス単位での実行管理となるため、ユーザスレッド1が事象待ちを伴う処理（入出力処理など）を実行すると、プロセスがブロックしてしまい、ユーザスレッド2もブロックしてしまい、複数個のスレッドを作った意味がなくなります。

それに対して、右側ではカーネル内に複数のスレッド（カーネルスレッド）が存在するため、ユーザスレッド1の実行をカーネルスレッド1、ユーザスレッド2の実行をカーネルスレッド2に対応付けることで、ユーザスレッド1が入出力処理を実行してカーネル内でブロックとなっても、カーネルスレッド2に切り替えることで、そのまま処理を継続することができます。

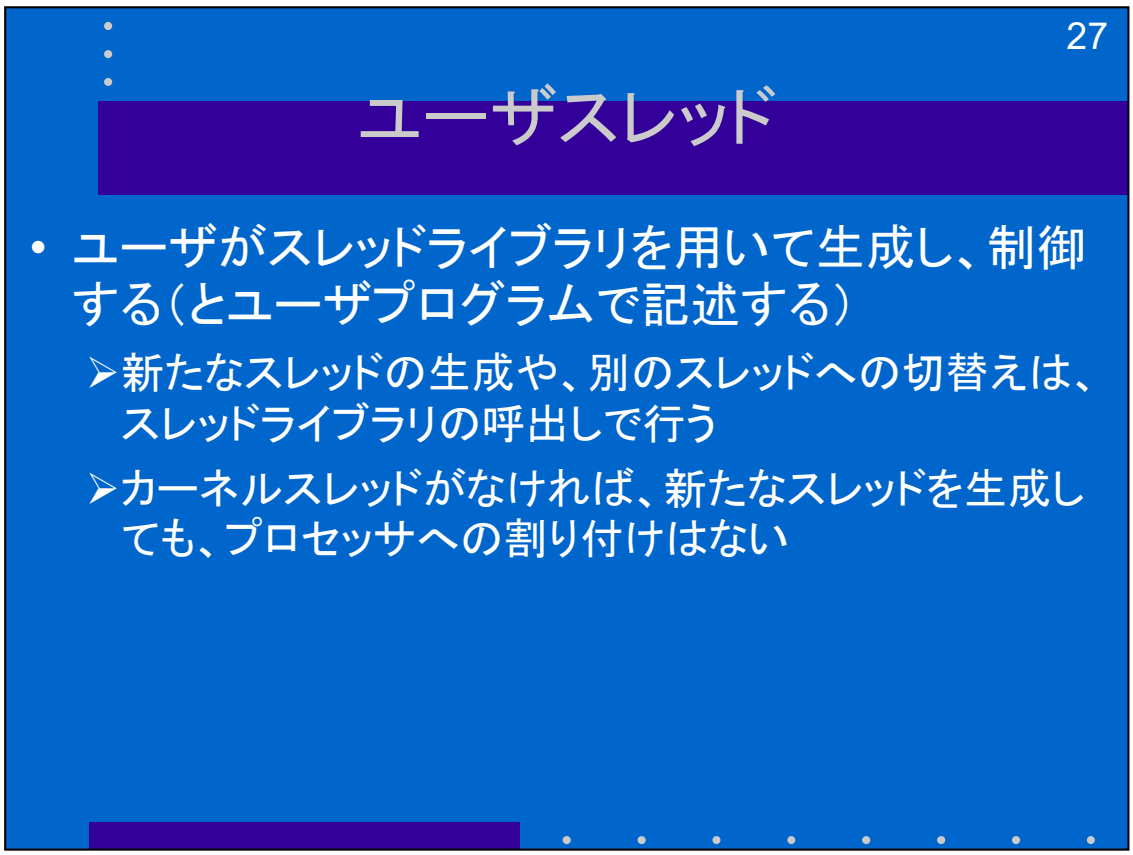


以上をまとめると、ユーザスレッドとカーネルスレッドの2種類が必要であることがわかります。

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しますが、実際に実行するにはプロセッサの割り付けが必要となります。

またカーネルスレッドでは、元々カーネルでプロセッサの管理をしていることから、プロセッサの割り付けはカーネルで行うことができます。

なお、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ことに注意します（理由は後述）。

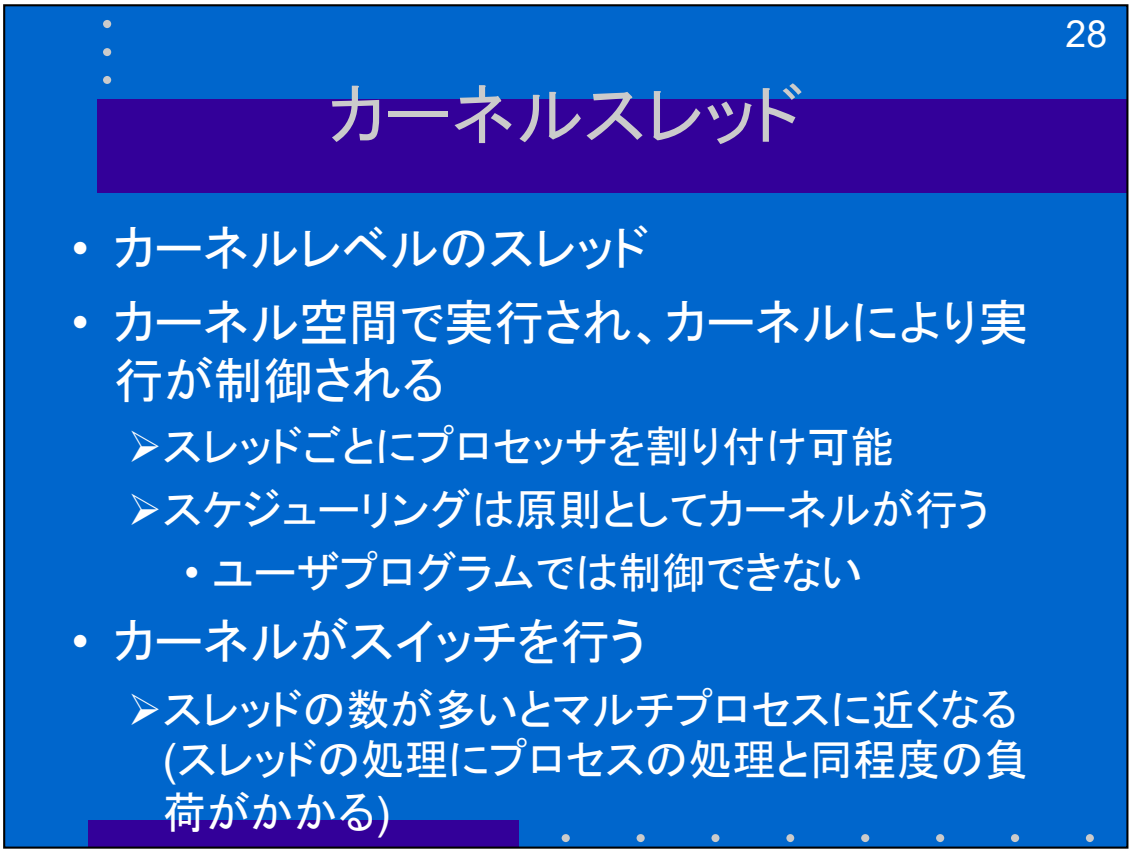


具体的な、スレッドの処理について見て行きます。

ユーザスレッドは、ユーザがスレッドライブラリを用いて生成し、制御します。

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われます。

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ありません。



これに対して、カーネルスレッドは、カーネルレベル、つまり特権モードで動作することになります。

カーネルスレッドは、カーネル空間で実行され、カーネルによりその実行が制御されます。

カーネルスレッドでは、スレッドごとにプロセッサを割り付けることが可能です。

また、スケジューリングは原則としてカーネルが行うため、ユーザプログラムでは制御できません。

カーネルがスレッドのスイッチを行います。

スレッドの数が多いとマルチプロセスに近くなります(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ります)

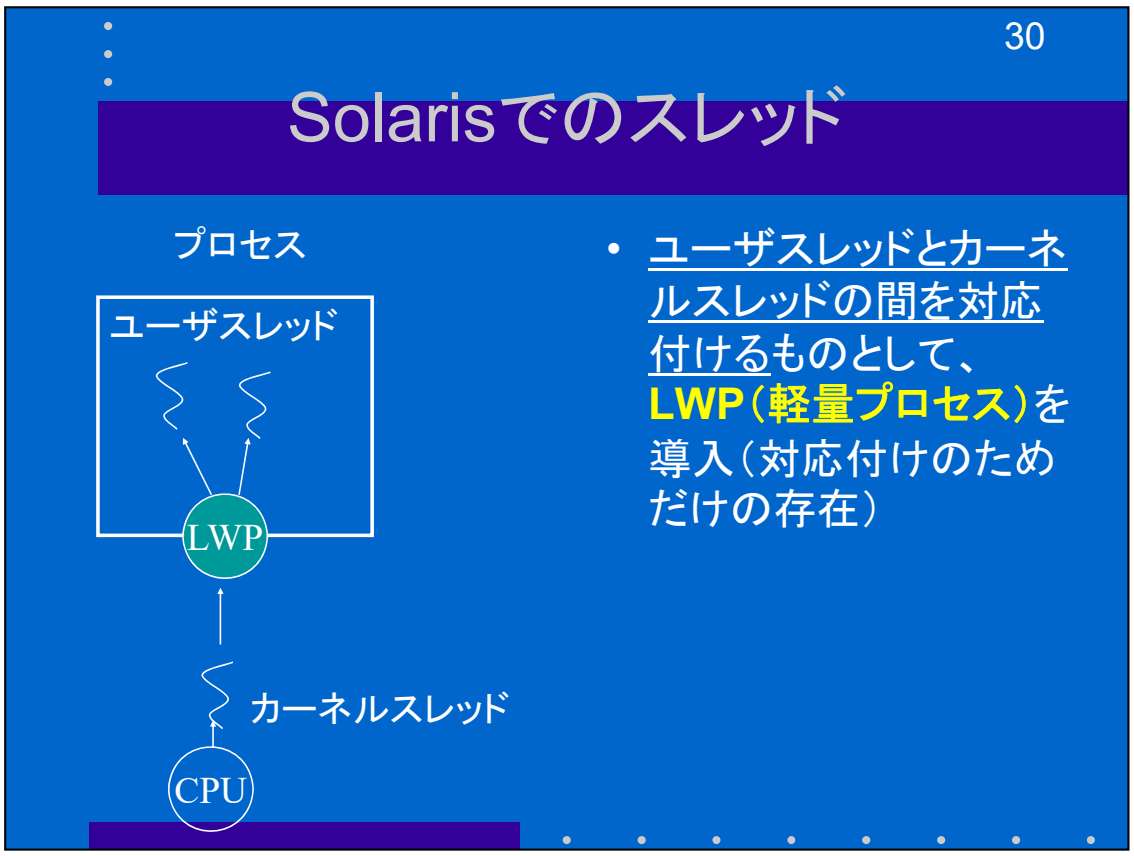
これらは、以降のスライドで図を使って説明します。

29

スレッドの利用例

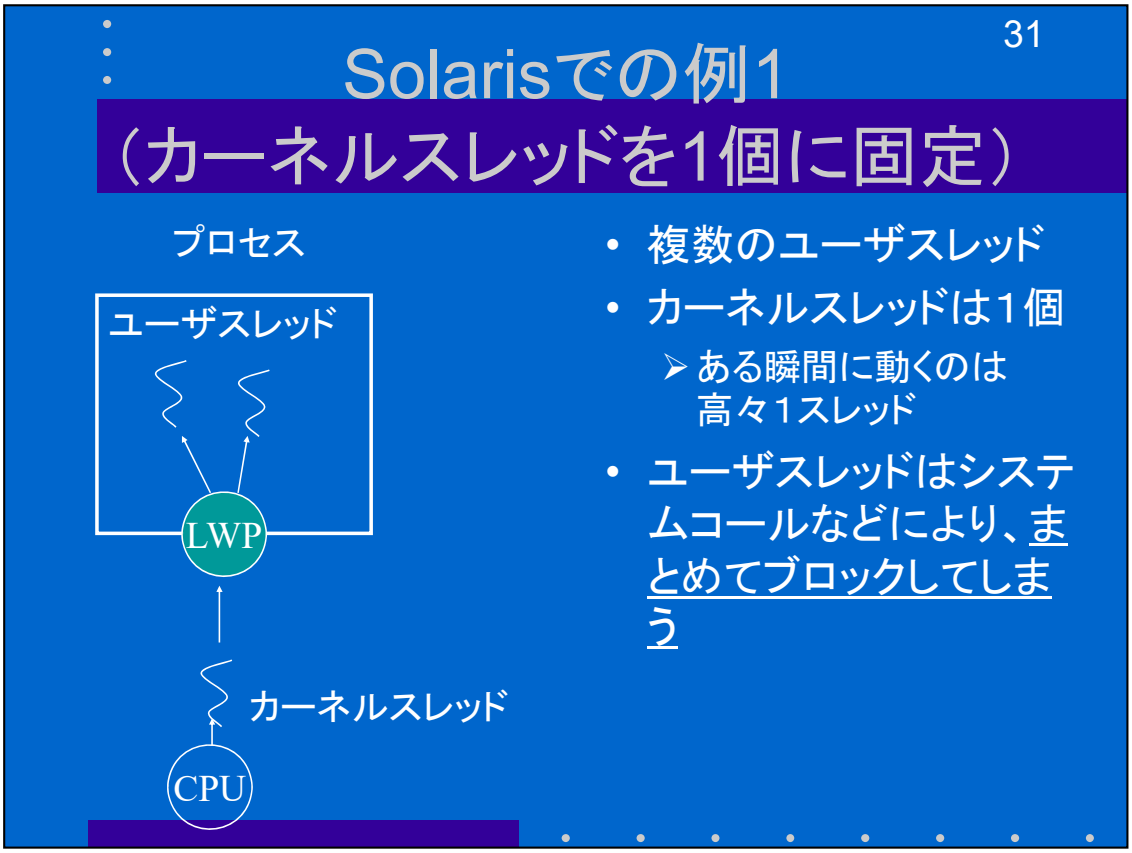
- IEEEで標準仕様が定められている
  - POSIX thread (pthread)と呼ばれる
  - 最も広く採用されている
- OSごとに少しずつ違った実装がある
- Solaris（UNIX系OS）のスレッドを例に説明

スレッドのインタフェースは、IEEEで標準仕様が定められています。
これは、POSIX thread (pthread)と呼ばれ、最も広く採用されています、
OSごとに少しずつ違った実装があるので、以降はSolaris（UNIX系OS）のスレッドを例に説明します。



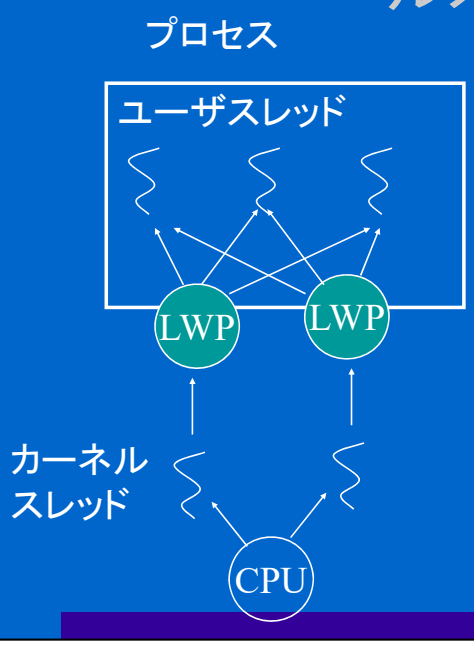
Solarisでは、ユーザスレッドとカーネルスレッドの間を対応付けるものとして、LWP（軽量プロセス）を導入しています。

LWPは、ユーザスレッドとカーネルスレッドの間の中間的なもので、両者の対応付けのためだけの存在です。



LWPを使って、すべてのユーザスレッドを一つのカーネルスレッドに対応付けると、先に述べたように、ある瞬間に動くのは高々1スレッドとなり、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ことになります。

Solarisでの例2（ユーザスレッド数より小さい数のカーネルスレッド）



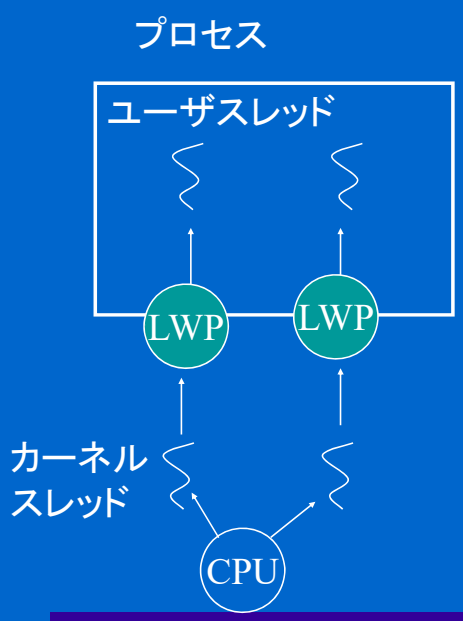
- ユーザスレッドの数＞カーネルスレッドの数≧2
  - カーネルスレッドの数だけ並行に動く
- ユーザスレッドが1個だけブロックしても、他のユーザスレッドは動く
- カーネルスレッドの数と同じ数（左図では2個）のユーザスレッドがブロックすると、残りのユーザスレッドもブロックする

ユーザスレッド数より小さい数のカーネルスレッドを設定するときは、ユーザスレッドはカーネルスレッドの数だけ並行に動くことになります。

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きます。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ことになります。

33

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

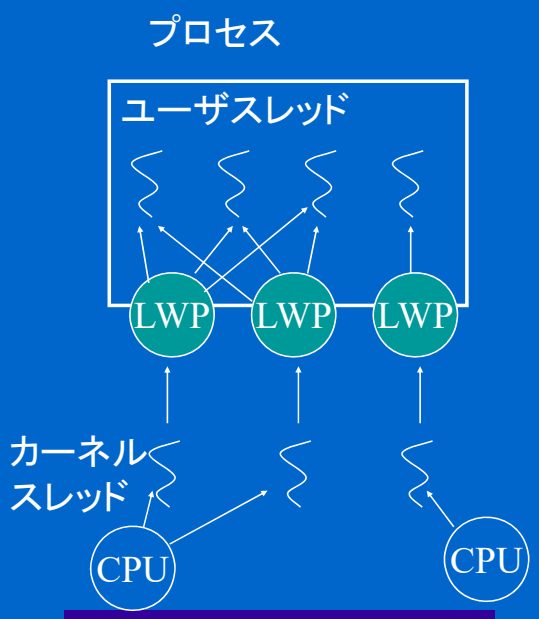
ユーザスレッドの数とカーネルスレッドの数が等しいときは、全部のユーザスレッドが並行に動くことになります。

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ます。

Windows　NT系 (2000,XP,Vista, 7, 8, 10)は、この方式となっています。

34

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

プロセッサまたはコアが複数個あるときは、カーネルスレッドとプロセッサ（コア）の対応を設定できます。

このため、プロセッサ（コア）に割り付けられたカーネルスレッドが並列に実行されることになります。