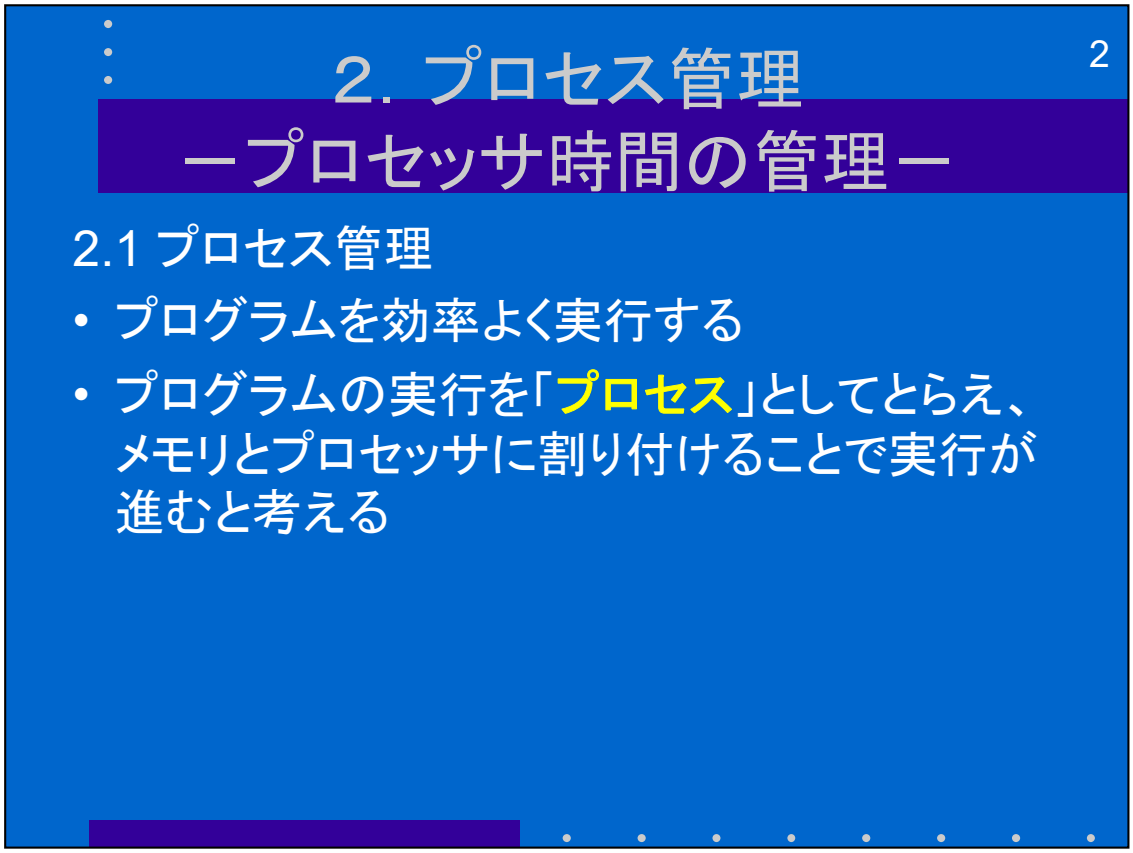
オペレーティングシステム

資料　第4分冊(2021)

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

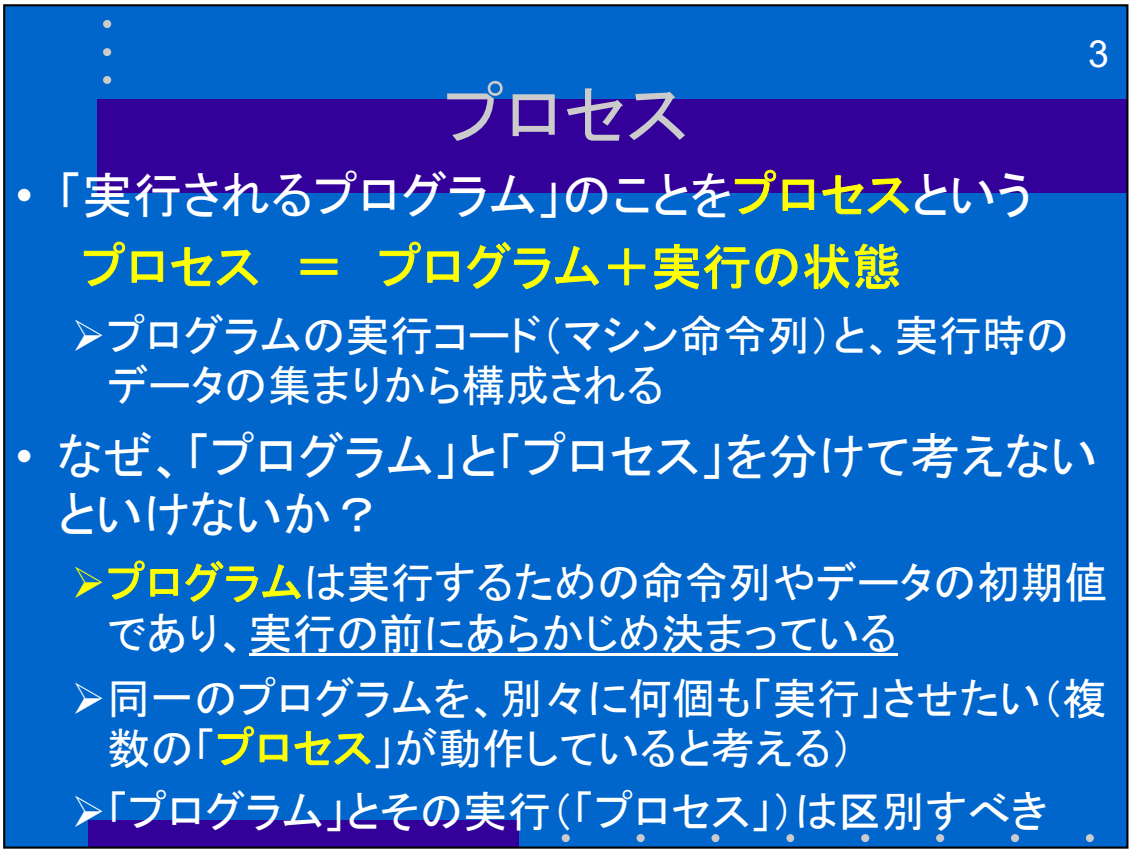


今回はプロセス管理について説明します。

プロセス管理の目的は、コンピュータシステムでプログラムを効率よく実行することです。

このために、プログラムの実行を「プロセス」としてとらえ、メモリとプロセッサに割り付けることで実行が進むと考えます。

特に、プロセッサの実行時間のうちのどこにプログラムの実行を割り付けるかが重要で、この意味ではプロセス管理はプロセッサの時間の管理と考えることができます。



より具体的には、「実行されるプログラム」のことをプロセスと呼びます。

つまり、プロセスとは、プログラムとそれの実行の状態が組み合わさったものを指し、プログラムの実行コードと、実行時のデータの集まりから構成されます。

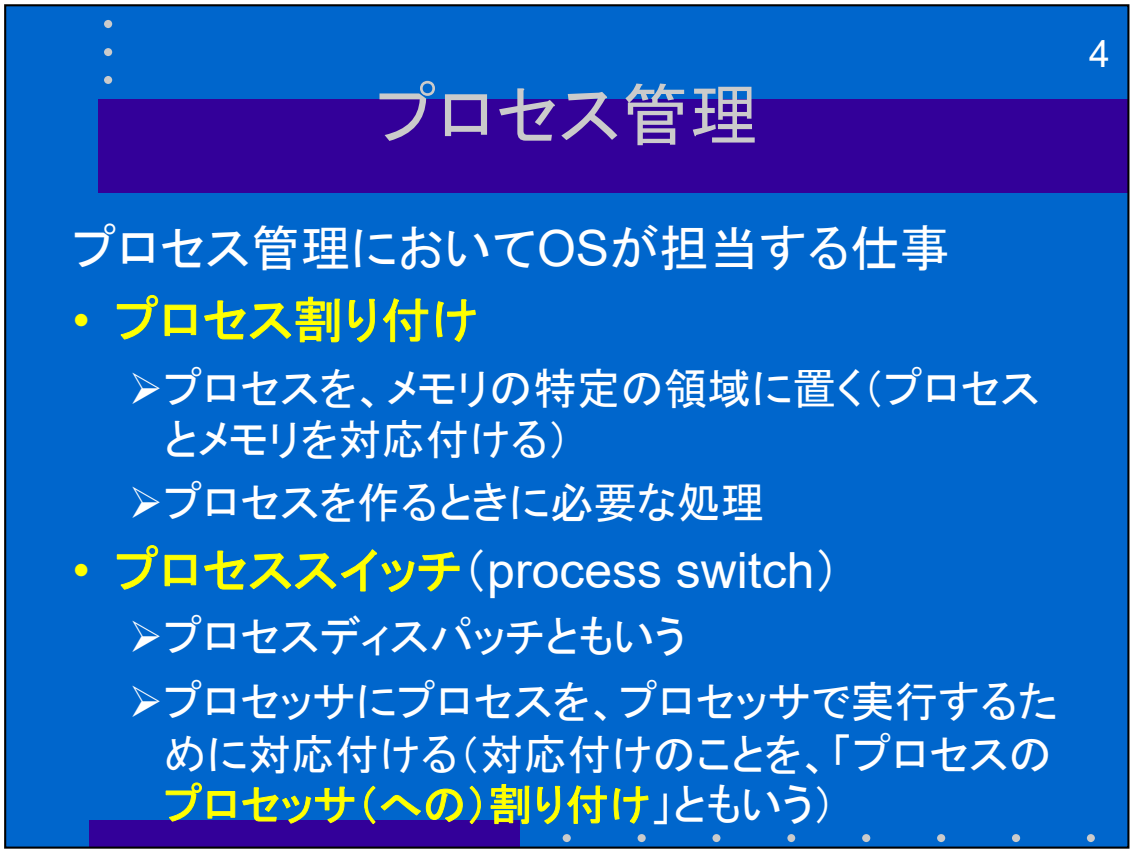
では、なぜ、「プログラム」と「プロセス」を分けて考えないといけないでしょうか？

「プログラム」は実行するための命令列やそこで使われるデータの初期値であり、コンパイル時、つまり実行の前にあらかじめ決まっています。

しかし、実際には、同じプログラムであっても、データを変えて複数個のプログラムを実行したいときがあります。

例えば、複数のファイルを、同じプログラム（例えば、Wordなど（で開いて編集することは普通に行われています。

この場合は、「プログラム」と「プロセス」を区別して。「プログラム」は同じでも、「プロセス」は異なっていると考えます。



プロセス管理においてOSが担当する仕事は次の2つです。

1つ目は、**プロセス割り付け**で、プロセスをメモリの特定の領域に置く（プロセスとメモリを対応付ける）ことです。これは、プロセスを作るときに必要な処理です。

2つ目は**プロセススイッチ**（process switch）です。これはプロセスディスパッチとも呼ばれ、プロセッサにプロセスを、プロセッサで実行するために対応付けます（対応付けのことを、「プロセスの**プロセッサ（への）割り付け**」ともいう）

5

プロセスの基礎

- **プロセス**とはプログラムの実行（プログラム＋プロセス領域）のこと （**タスク**とも呼ばれる）
  - 同一のプログラムでも複数実行されれば別のプロセス
  - **プロセス領域**とは、メモリに割り付けられるプロセスの実体を格納している領域（図2.7参照）
- **プロセス**と**ジョブ**の違い

  実質的には同じものを指すが、
  - ジョブという用語はユーザから見た処理の単位
  - プロセスという用語はOS側から見た処理の単位

  として使われることが一般的である

それでは、より具体的に、プロセスを見て行きます。

プロセスとはプログラムの実行（プログラム＋プロセス領域）のことで、OSによっては**タスク**とも呼ばれます。

同一のプログラムでも複数実行されれば別のプロセスとなり、

プロセス領域とは、メモリに割り付けられるプロセスの実体を格納している領域（教科書の図2.7参照）

プロセスとよく似た概念に、**ジョブ**があります。両者は、実質的には同じものを指すが、ジョブという用語はユーザから見た処理の単位を意味し、プロセスという用語はOS側から見た処理の単位を指すことが多いです。

つまり、OSから見る時は、「プロセス」と呼ぶのが一般的です。

プロセスとプロセッサ管理・制御方式

6

マルチプログラミング

- 複数個のプログラムの実行を切り替えることで、同時に実行されているように見せかける
- 1台のコンピュータ上に、「**複数個のプログラムが存在して動作する**」ことが主眼で、プロセッサ管理・制御方式は何でもよい

マルチタスキング（図2.1）

- プロセッサとプロセス（タスク）の対応は「**1プロセッサ対多プロセス**」
- プロセッサを多重化し時分割制御（TSS）によってプロセスを切り替える

マルチプロセッシング

- プロセッサとプロセスの対応付けである点は、マルチタスキングと同じ
- 違いは多数個のプロセッサによる「**多プロセッサ対多プロセス**」の対応

技術的な進展の度合い

マルチプログラミング → マルチタスキング → マルチプロセッシング

プロセスと、プロセッサ管理・制御方式との関係について説明します。

マルチプログラミングとは、複数個のプログラムの実行を切り替えることで、同時に実行されているように見せかける方式を指し、1台のコンピュータ上に、「複数個のプログラムが存在して動作する」ことが主眼で、プロセッサ管理・制御方式は特に定めずに何でもよいです。

マルチタスキング（教科書の図2.1）になると、プロセッサとプロセス（タスク）の対応は「1プロセッサ対多プロセス」となり、プロセッサを多重化し時分割制御（TSS）によってプロセスを切り替えるようになります。

マルチプロセッシングは、プロセッサとプロセスの対応付けである点は、マルチタスキングと同じですが、違いは、多数個のプロセッサによる「多プロセッサ対多プロセス」の対応付けを行うことです。

以上のことから、技術的な進展の度合いは、マルチプログラミング、マルチタスキング、マルチプロセッシングの順に複雑かつ高度になっていきます。

7

プロセスの状態

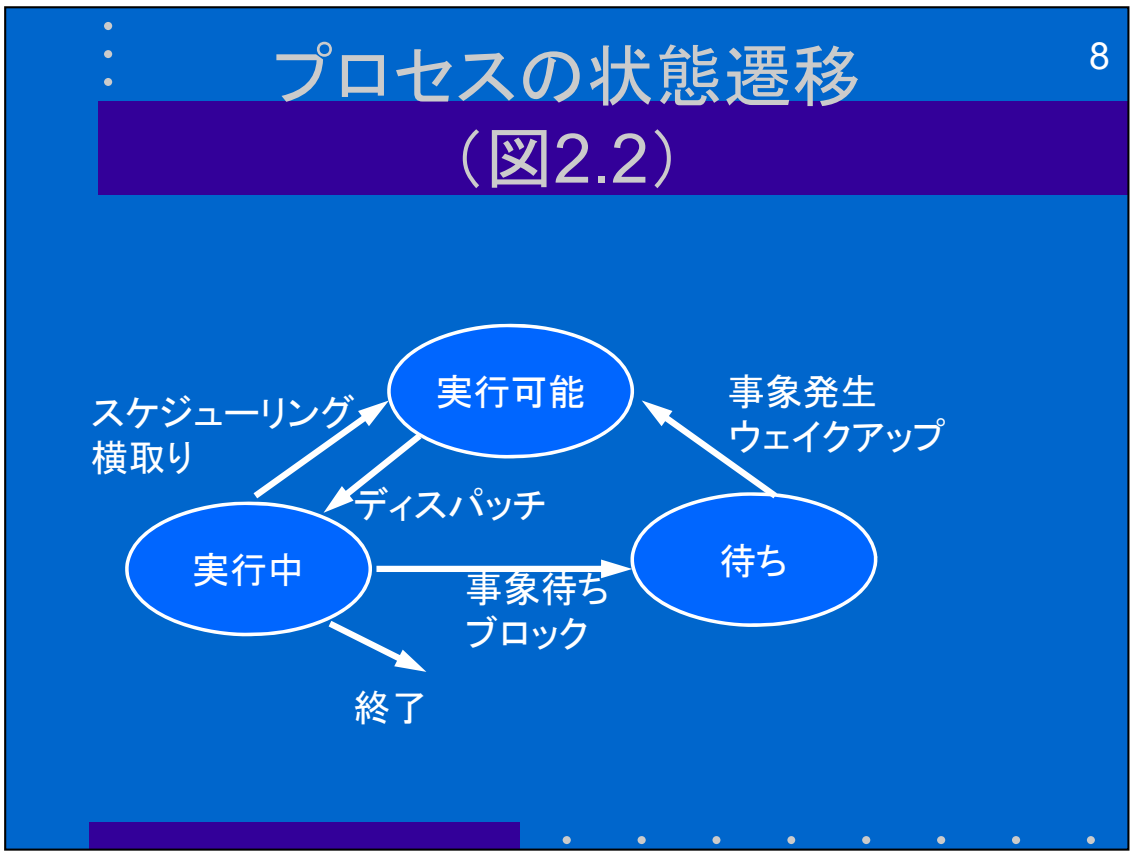
- **実行中**(running)
  - 実行中のプロセスの個数は、プロセッサの個数（最近ではコアの数）で決まる
- **実行可能**（ready)
  - プロセッサが割り付けられれば実行できる（実行の順番待ち）
- **待ち**（waiting）
  - 何らかの理由で実行できない
  - 何かを待っている

プロセスは、次のような状態を持ちます。

実行中(running)：実行中のプロセスの個数は、プロセッサの個数（最近ではコアの数）で決まります。

実行可能（ready)：プロセッサが割り付けられれば実行できる、つまり実行の順番待ちのプロセスの状態を指します。

待ち（waiting）：何らかの理由で実行できない状態で、何か（例えば、入出力処理の完了）を待っている状態を指します。



プロセスの状態は、この図（教科書の図2.2と同じ）のように遷移します。
以降のスライドで詳細に説明します。

9

プロセスの状態遷移の要因

- ディスパッチ
  - 実行可能→実行中の遷移
- 横取り（preemption）
  - 実行中→実行可能の遷移
- 事象（event）
  - 実行中→待ち、待ち→実行可能の遷移を起こす要因
- ブロック
  - 事象の発生を待つために、実行中→待ちの遷移
- ウェイクアップ
  - 事象の発生による、待ち→実行可能の遷移

プロセスの状態遷移の要因は次の通りです。
ディスパッチ：実行可能→実行中の遷移
横取り（preemption）：実行中→実行可能の遷移
事象（event）：実行中→待ち、待ち→実行可能の遷移を起こす要因
ブロック：事象の発生を待つために、実行中→待ちの遷移
ウェイクアップ：事象の発生による、待ち→実行可能の遷移

10

プロセス管理と割り込み処理

- 実行中のプロセスの状態遷移は、**割り込み処理**により行われる
- 2種類の状態遷移
  - **ブロック**:　実行中→待ち
  - **横取り**:　　実行中→実行可能
- 割り込み処理により、(ユーザ)プロセスの切り替え(**プロセススイッチ**)が発生する(図2.3, 図2.4)
- 入出力処理の例
  - 入出力処理の依頼(SVC:システムコール)で**ブロック**し、入出力装置からの入出力割り込みで**ウェイクアップ**する(図2.5, 図2.6)

プロセス管理と割り込み処理は、次のように関係します。

まず、実行中のプロセスの状態遷移は、割り込み処理により行われます。

2種類の状態遷移があり、

ブロック:　実行中→待ち

横取り:　　実行中→実行可能

となります。

割り込み処理により、(ユーザ)プロセスの切り替え(プロセススイッチ)が発生します(教科書の図2.3, 図2.4)

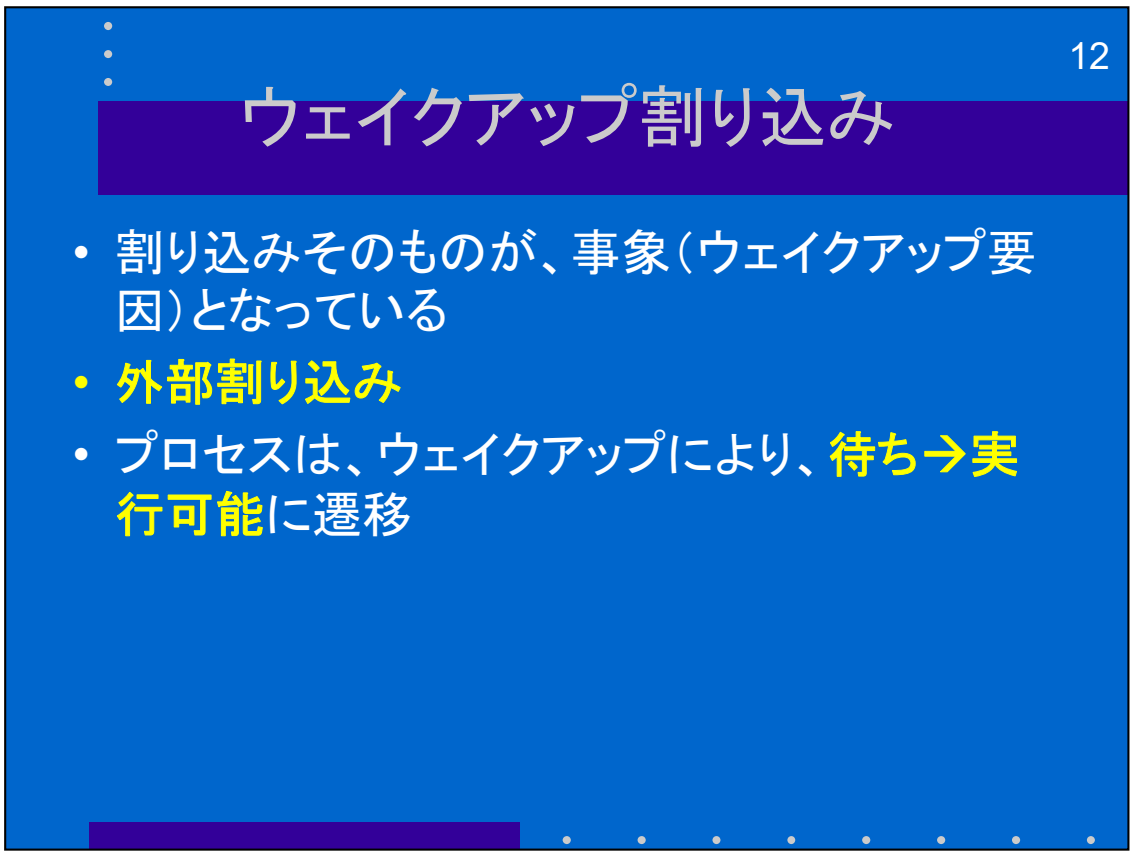
入出力処理の例:入出力処理の依頼(SVC:システムコール)でブロックし、入出力装置からの入出力割り込みでウェイクアップする(教科書の図2.5, 図2.6)

11

ブロック割り込み

- 実行中のプロセスがブロックする割り込みを指す
- プロセスの状態は、**実行中→待ち**に遷移
- 割り込み要因は、当該プロセス自身である（**内部割り込み**）
- 内部割り込みの発生頻度の大半は、ブロック割り込み
- ブロック割り込みの発生により、（他の）実行可能プロセスを**ディスパッチ**（**実行可能→実行中**に遷移）

ブロック割り込みとは、実行中のプロセスがブロックする割り込みを指します。
プロセスの状態は、実行中→待ちに遷移します。
割り込み要因は、当該プロセス自身であり、内部割り込みに分類されます。
内部割り込みの発生頻度の大半は、このブロック割り込みとなります。
ブロック割り込みの発生により、（他の）実行可能プロセスをディスパッチし、そのプロセスは実行可能→実行中に遷移します。



ウェイクアップ割り込みとは、ウェイクアップ（入出力処理など、プロセスが待っていた処理が完了し、待ち状態の原因がなくなること）により発生する割り込みのことです。

この割り込みそのものが、事象（ウェイクアップ要因）となっています。

これは、外部割り込みとなります。

プロセスは、ウェイクアップにより、状態が待ち→実行可能に遷移します。

システムコールと入出力割り込み（図2.5）

13

- プロセスは、入出力処理の実行を**SVC**（システムコール）によりOSに依頼
  - **内部割り込み**
  - 入出力処理の完了という事象待ちのための**ブロック割り込み**となる（実行中→待ちに遷移）
- 入出力処理が完了すると、**入出力割り込み**により通知（**外部割り込み**）
  - 事象の発生による**ウェイクアップ割り込み**となる（待ち→実行可能に遷移）

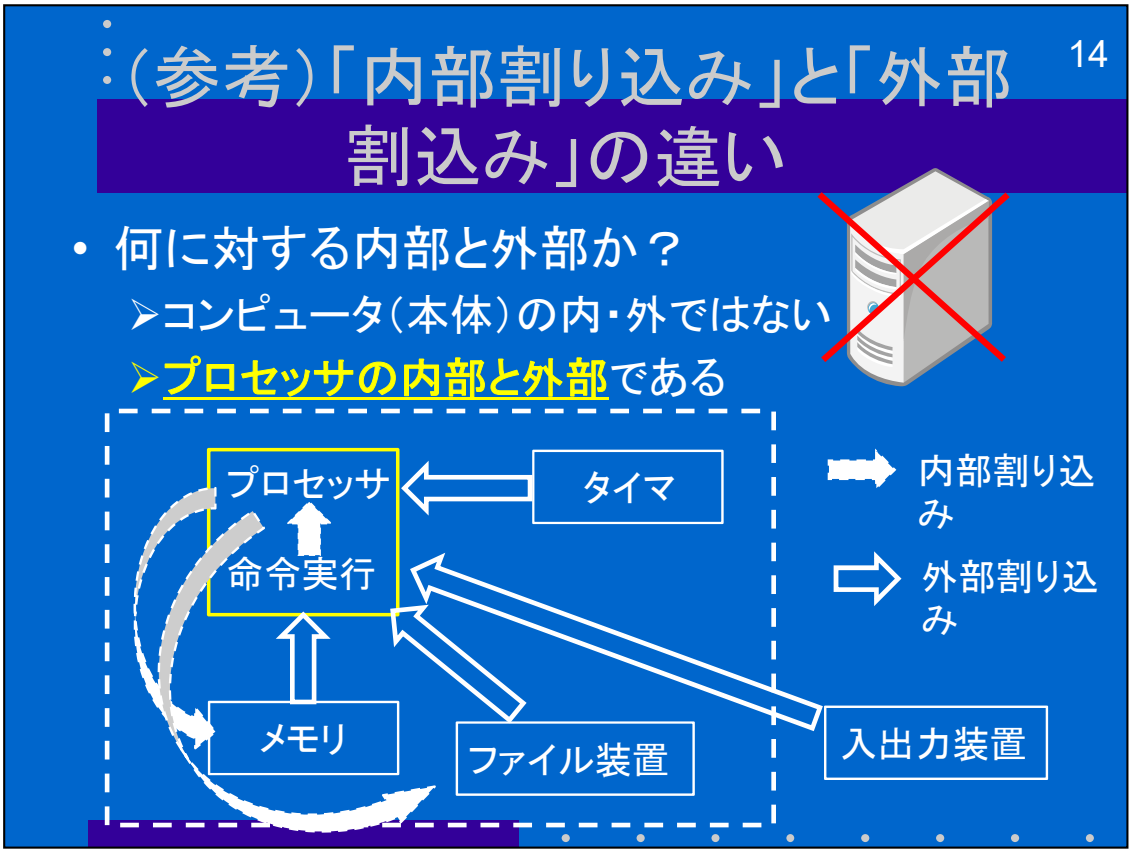
プロセスは、入出力処理の実行をSVC（システムコール）によりOSに依頼することで発生します。

これは、内部割り込みとなります。

入出力処理の完了という事象待ちのためのブロック割り込みとなります（実行中→待ちに遷移）

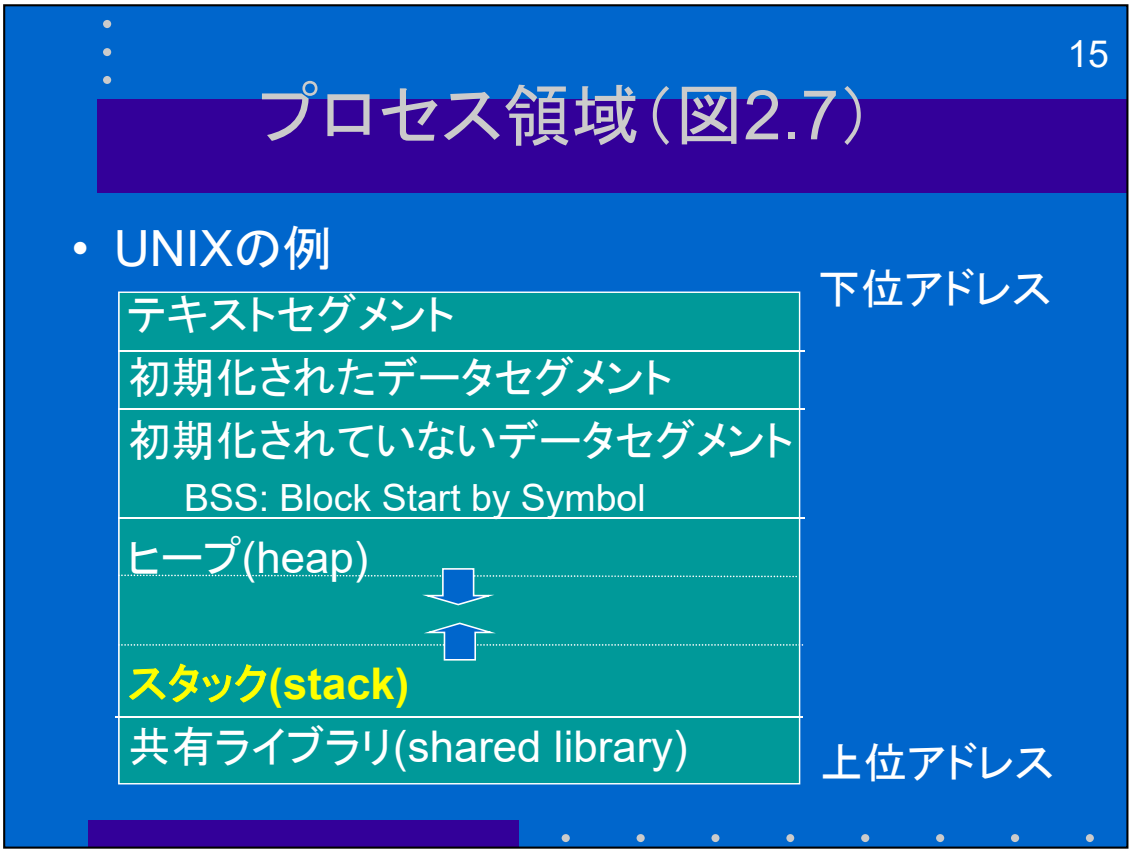
入出力処理が完了すると、入出力割り込みにより通知します（外部割り込み）

事象の発生によるウェイクアップ割り込みとなります（待ち→実行可能に遷移）



内部割り込みと外部割り込みの違いについて復習します。

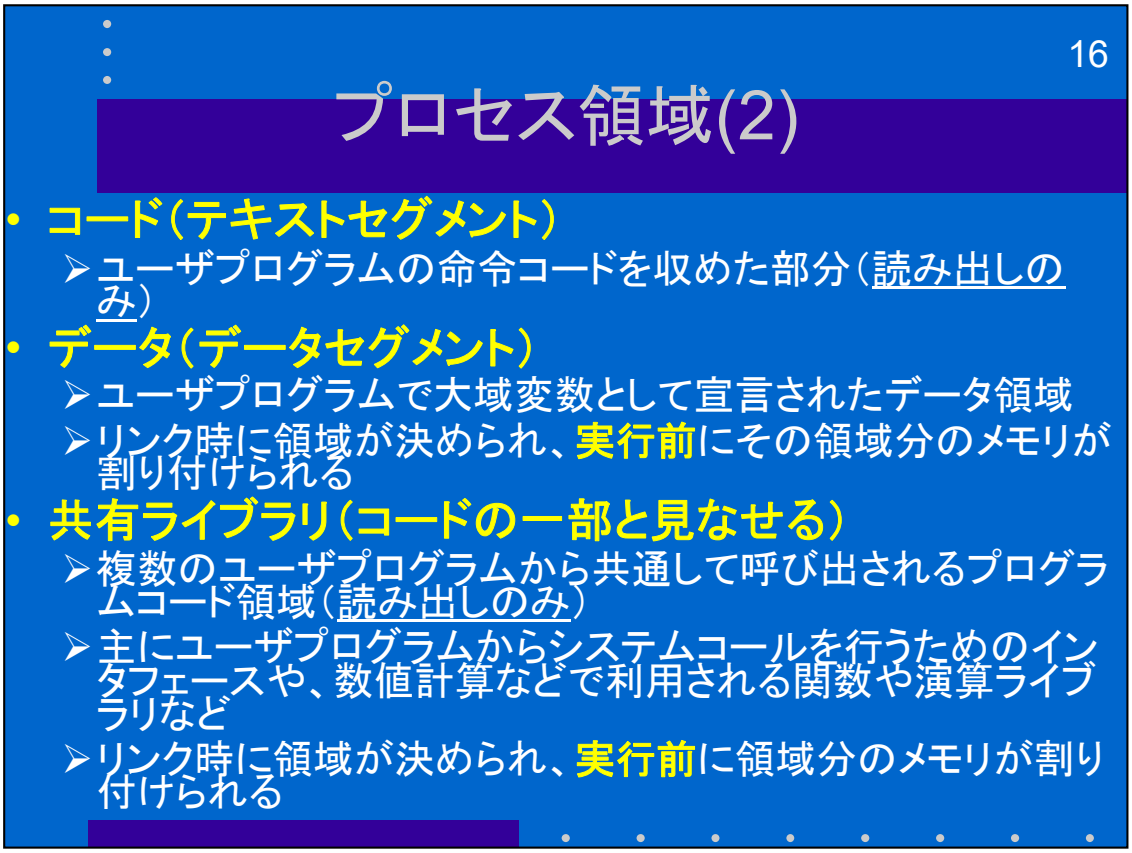
内部と外部とは、プロセッサの内部と外部であることに注意します。



プロセス領域について説明します（教科書の図2.7）

この図は、UNIXの例を示します。

この図のように、UNIXではプロセス領域はいくつかのセグメントと呼ばれる小領域に分かれます。



プロセス領域中の個々の領域について説明します。

コード(テキストセグメント)とは、ユーザプログラムの命令コードを収めた部分で、この領域の参照は読み出しのみです。

データ(データセグメント)は、ユーザプログラムで大域変数として宣言されたデータ領域であり、リンク時に領域が決められ、実行前にその領域分のメモリが割り付けられます。

共有ライブラリは、コードの一部であり、複数のユーザプログラムから共通して呼び出されるプログラムコード領域で、参照は読み出しのみです。

主にユーザプログラムからシステムコールを行うためのインタフェースや、数値計算などで利用される関数や演算ライブラリなどがここで格納されます。

リンク時に領域が決められ、実行前に領域分のメモリが割り付けられます。

17

プロセス領域(3)

スタック

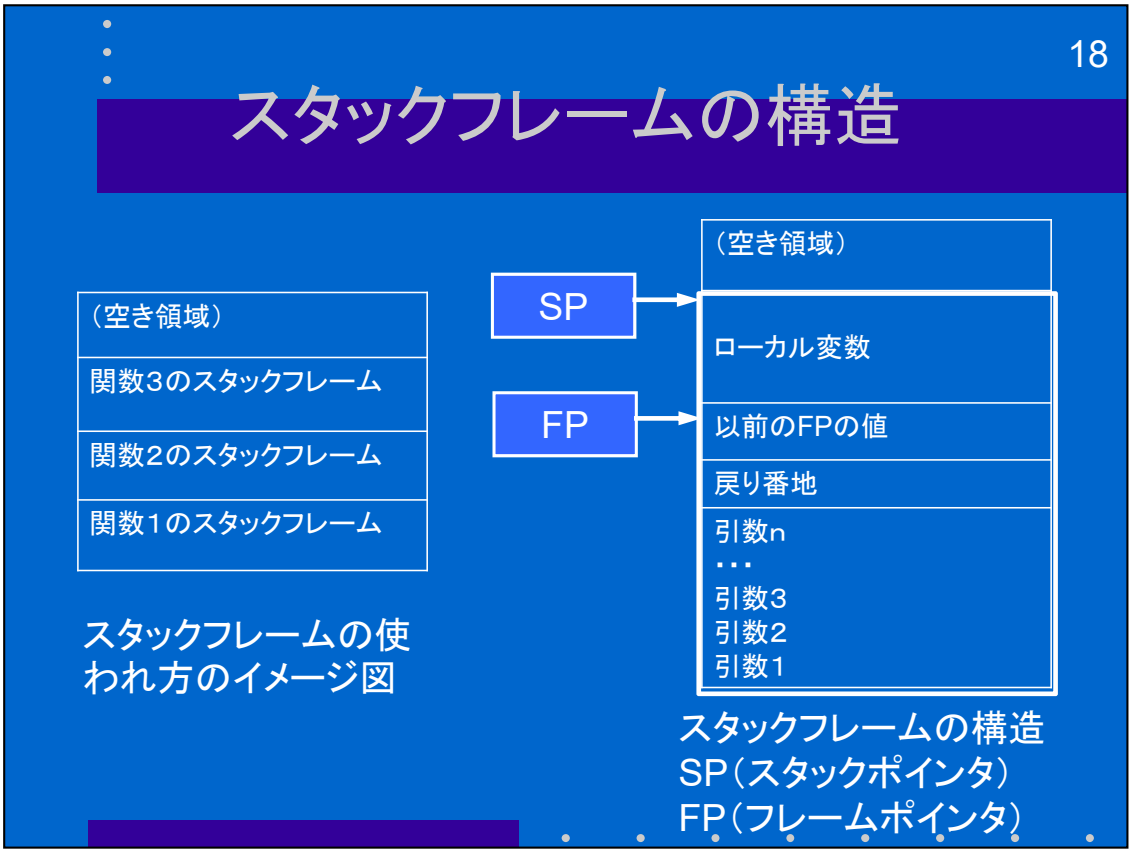
- スタックフレームを格納
- スタックフレームとは？
  - ➢活性レコード(activation record)と呼ぶこともある
  - ➢スタック上に作成される領域
    - → **関数呼び出しごと**に、連続した領域に順次作られる
  - ➢関数の引数、局所変数、戻り番地、前のフレームへのポインタなどから構成される
- **図2.7の「スタックフレーム」は「スタック」の意味**

スタックとは、スタックフレームを格納する領域のことです。

スタックフレームとは、活性レコード(activation record)と呼ぶこともあり、スタック上に作成される領域で、関数呼び出しごとに、連続した領域に順次作られます。

関数の引数、局所変数、戻り番地、前のフレームへのポインタなどから構成されます。

教科書の図2.7の「スタックフレーム」とは通常は関数呼び出しごとに作られる領域を指すので、これは「スタック」と表記すべきと思います。



スタックフレームの構造はこの図の通りです。

19

プロセス領域(4)

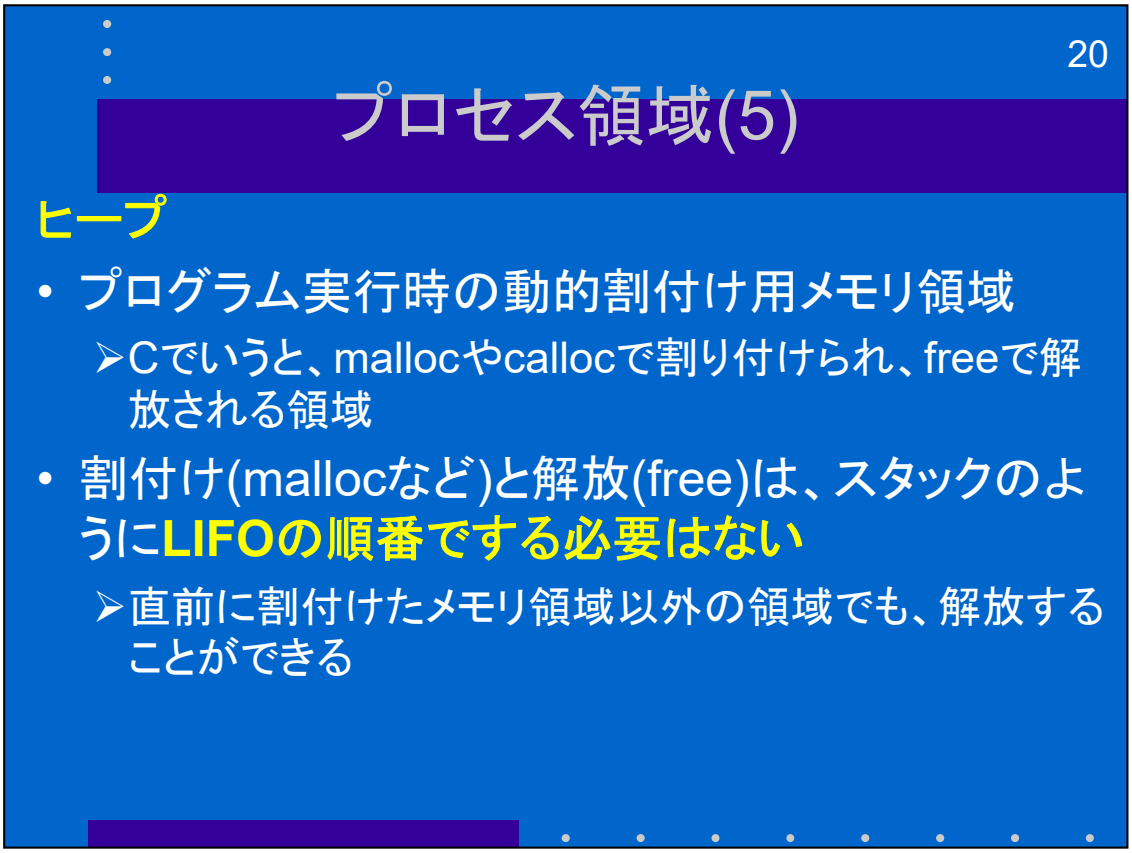
スタック（つづき）

- 昔の使われ方
  - サブルーチンコールの戻り番地
- 現在の使われ方
  - 高級言語での手続き呼出しの引数領域
  - 手続き内の局所変数の置き場所
  - その他の作業領域
- 昔も現在も、割り付けと解放の順番が**LIFO (Last-In First-Out)**であることは共通（図2.8）

スタックの使われ方は、昔と現在では少し違っています。
昔の使われ方は、サブルーチンコールの戻り番地だけでした。
現在の使われ方は、次のようにいろいろなことに使われます。

高級言語での手続き呼出しの引数領域
手続き内の局所変数の置き場所
その他の作業領域

ただし、昔も現在も、割り付けと解放の順番がLIFO (Last-In First-Out)であることは共通しています（教科書の図2.8）



ヒープとは、プログラム実行時の動的割付け用メモリ領域を指します。

Cでいうと、mallocやcallocで割り付けられ、freeで解放される領域です。

割付け(mallocなど)と解放(free)は、スタックのようにLIFOの順番でする必要はないことに注意する必要があります。

直前に割付けたメモリ領域以外の領域でも、解放することができます。

21

プロセス領域（まとめ）

領域名	メモリの割付け	領域の参照
コード（テキストセグメント、共有ライブラリ）	実行前に割り付け	読み出しのみ
データ（データセグメント）	実行前に割り付け	読み出しと書き込み
ヒープ	実行中に割り付け（割り付けと解放の順番は任意）	読み出しと書き込み
スタック	実行中に割り付け（割り付けと解放はLIFOの順）	読み出しと書き込み

以上のプロセス領域をまとめるとこの表になります。

22

プロセス領域と共有

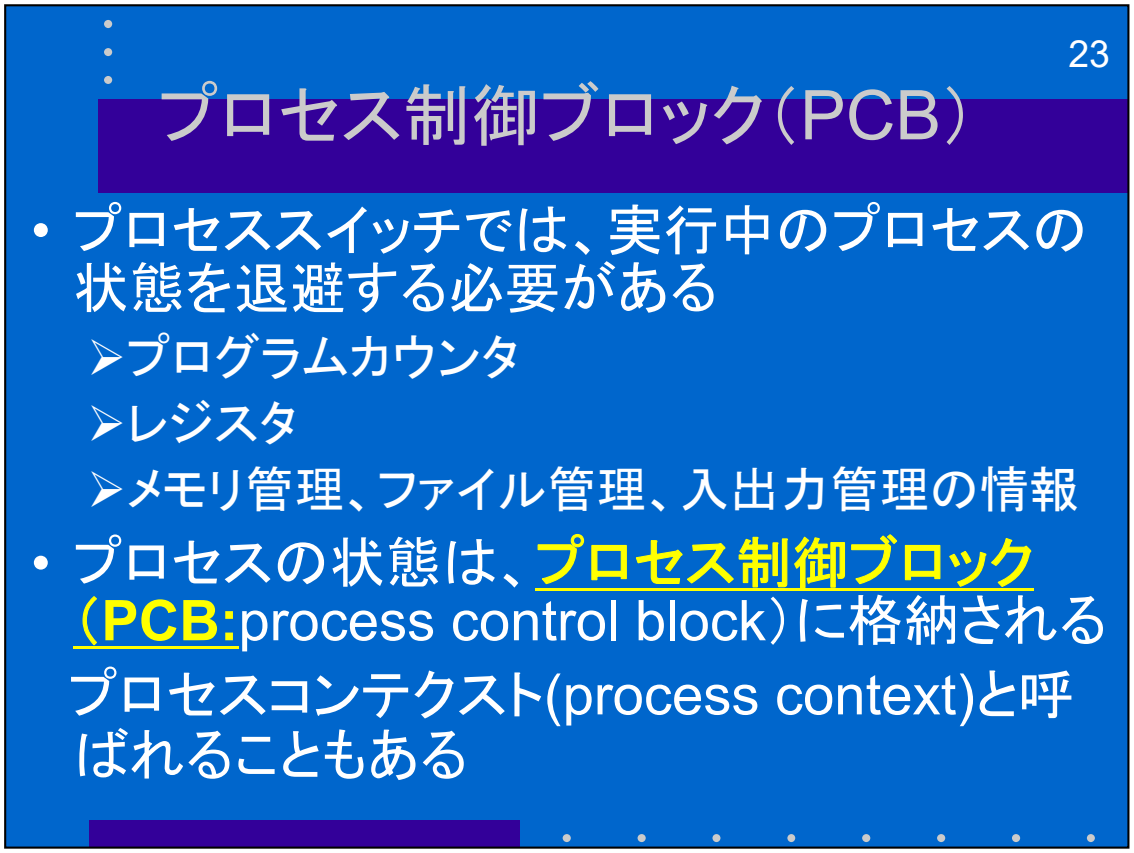
- コードは共有可、データとヒープは共有または非共有、スタックは共有不可となっている（図2.7）
- プロセス間で共有すると問題がある領域と、問題がない領域の違いは何か？（続きは課題）

一つのプロセス領域を、複数のプロセスで共有できるかどうかを考えます。

教科書の図2.7では、コードは共有可、データとヒープは共有または非共有、スタックは共有不可となっています、

プロセス間で共有すると問題がある領域と、問題がない領域の違いは何でしょうか？

続きは課題で考えてみましょう。



プロセス管理では、プロセス領域の他に、プロセス領域ブロックという領域も管理されます。

プロセススイッチでは、実行中のプロセスの状態として次のようなものを退避する必要があります。

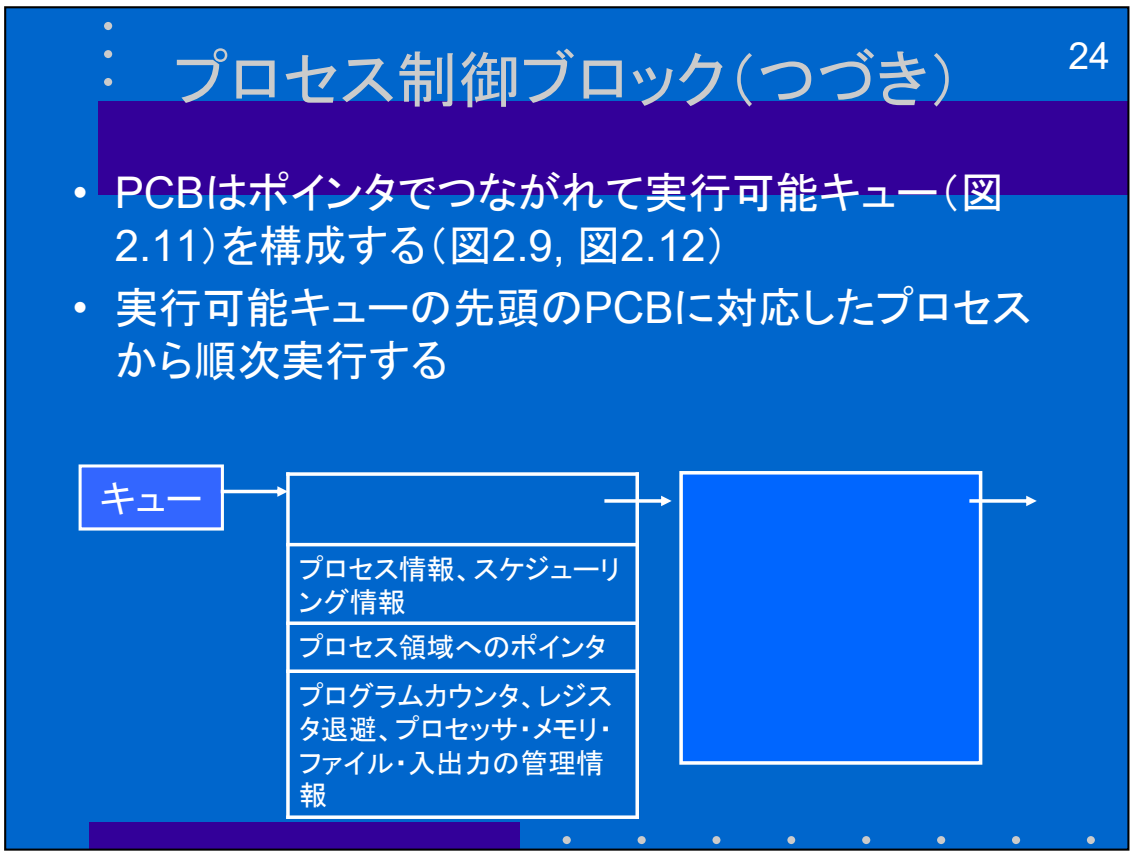
プログラムカウンタ

レジスタ

メモリ管理、ファイル管理、入出力管理の情報

プロセスの状態は、プロセス制御ブロック(PCB:process control block)にまとめられて格納されます

プロセスコンテクスト(process context)と呼ばれることもあります



プロセス制御ブロック（PCB）はポインタでつながれて実行可能キューを構成します。

キューの先頭のPCBのプロセスから順次実行されます。

25

プロセススイッチ

- プロセススケジューリング
  - スケジューリング方式に基づいて、次に実行すべきプロセスを選択すること
- プロセスディスパッチ
  - プロセススケジューリングにより選択されたプロセスにプロセッサを割り付けること
- オペレーティングシステムの方針と機構の分離の例にもなっている（スケジューラとディスパッチャ）
- 具体的な動作（図2.13、図2.14）

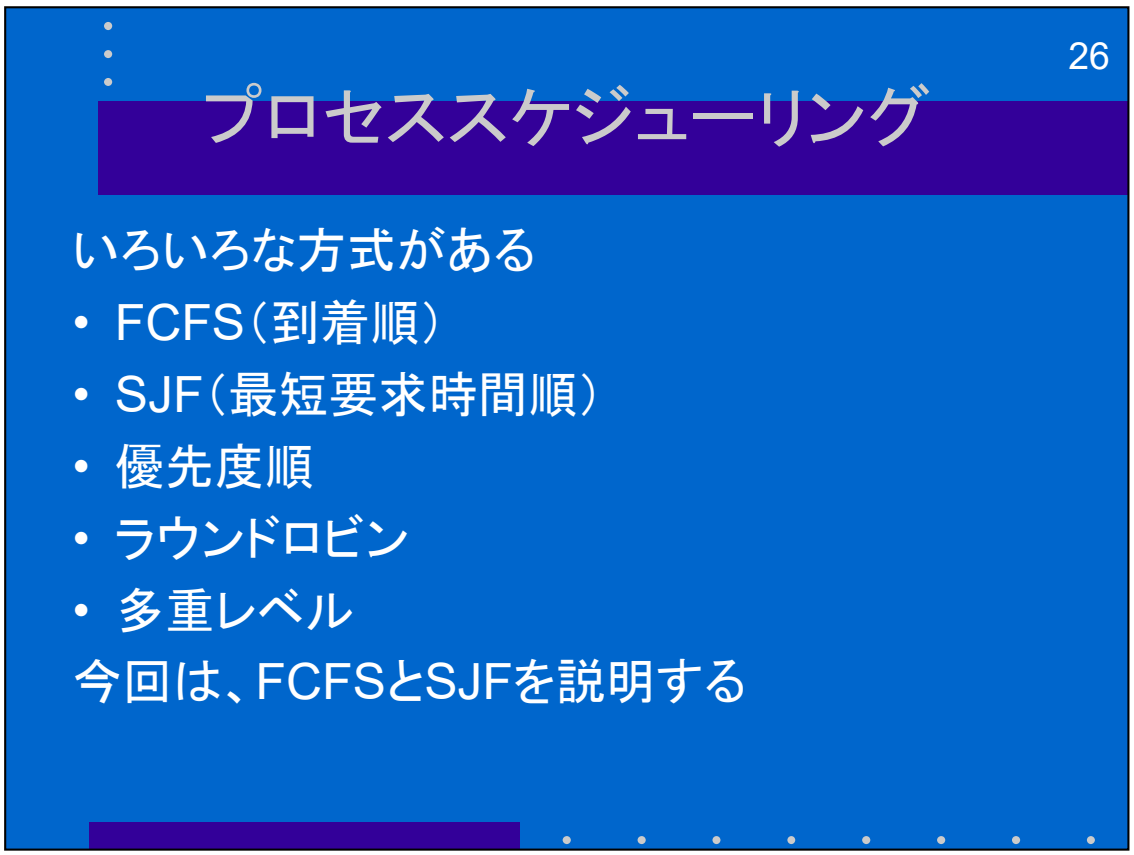
プロセススイッチでは、次の2つを行います。

1つ目はプロセススケジューリングで、スケジューリング方式に基づいて、次に実行すべきプロセスを選択することです。

2つ目は、プロセスディスパッチで、プロセススケジューリングにより選択されたプロセスにプロセッサを割り付けることを指します。

この2つは、オペレーティングシステムの方針と機構の分離の例にもなっています（スケジューラとディスパッチャ）。

具体的な動作は、教科書の図2.13、図2.14に書かれています。



プロセススケジューリングでは、ここにあげられているように、いろいろな方式があります。

以下、それぞれについて説明していきます。

スケジューリングアルゴリズムの選定指標

27

- **プロセッサ（CPU）利用率**
  - （プロセッサの有効動作時間）／（総稼働時間）
- **ターンアラウンド時間**（turnaround time）
  - プロセスの到着（生成）から、完了（消滅）までの時間
- 待ち時間
  - プロセスの到着から完了までに、実行可能キューで費やす時間
- 応答時間
  - プロセスの到着から応答を開始するまでの時間（実質的には実行中になるまでの時間）

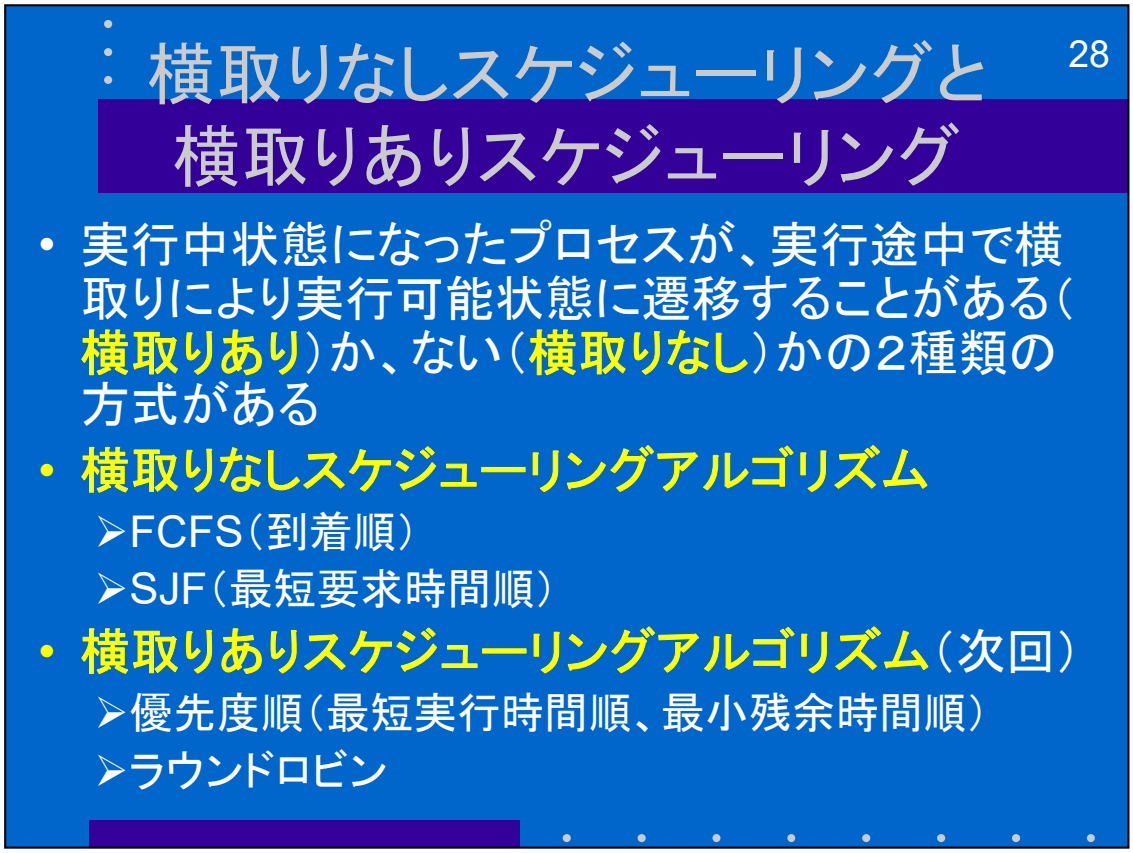
スケジューリングアルゴリズムの選定指標にはいろいろなものがあります。

プロセッサ（CPU）利用率：（プロセッサの有効動作時間）／（総稼働時間）

ターンアラウンド時間（turnaround time）：プロセスの到着（生成）から、完了（消滅）までの時間

待ち時間：プロセスの到着から完了までに、実行可能キューで費やす時間

応答時間：プロセスの到着から応答を開始するまでの時間（実質的には実行中になるまでの時間）



横取りなしと横取りありのプロセススケジューリングについて説明します。

実行中状態になったプロセスが、実行途中で横取りにより実行可能状態に遷移することがある（横取りあり）か、ない（横取りなし）かの2種類の方式があります。

横取りなしスケジューリングアルゴリズム：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アルゴリズム（次回の講義で説明します）：優先度順（最短実行時間順、最小残余時間順）、ラウンドロビン

29

FCFS（到着順）

- First Come First Servedの略
  - FIFO (First In First Out)ともいう
- **実行可能キューに到着した順**に、プロセスをプロセッサを割り付ける（図2.16）
- 欠点
  - 処理に時間がかかるプロセスが、処理の短いプロセスを妨害する
  - プロセスに優先度がつけられない（急いで実行して欲しいプロセスがあっても、後に到着すれば後回し）

FCFSとは、First Come First Servedの略で、到着準とも呼ばれます。

FCFSは、FIFO (First In First Out)とも言います。

実行可能キューに到着した順に、プロセスをプロセッサを割り付けます（教科書の図2.16）

FCFSの欠点は次の通りです。

処理に時間がかかるプロセスが、処理の短いプロセスを妨害する

プロセスに優先度がつけられない（急いで実行して欲しいプロセスがあっても、後に到着すれば後回し）

30

SJF（最短要求時間順）

- Shortest Job Firstの略
- （予想される）**処理時間の短い順**で処理する（図2.17）
- ターンアラウンド時間（プロセスの完了時刻と到着時刻の差）が短くなる
- 処理時間があらかじめ与えられていることが前提（実際は困難）

SJF（最短要求時間順）とは、Shortest Job Firstの略です。

（予想される）処理時間の短い順で処理します（教科書の図2.17）

ターンアラウンド時間（プロセスの完了時刻と到着時刻の差）が短くなるという特徴があります。

処理時間があらかじめ与えられていることが前提となっていますが、実際は困難です。

31

スケジューリングの例題

- 初期状態としてプロセスがないときに、新しいプロセスが3個到着したとする
- それぞれの到着時刻と所要実行時間
  - ➢プロセス1　到着時刻　0, 所要実行時間　6
  - ➢プロセス2　到着時刻　1, 所要実行時間　20
  - ➢プロセス3　到着時刻　2, 所要実行時間　1
- FCFSとSJFでプロセスの実行開始時刻と実行終了時刻はどうなるか？

スケジューリングを例題で説明します。

初期状態としてプロセスがないときに、新しいプロセスが3個到着したと仮定します。

それぞれの到着時刻と所要実行時間

プロセス1　到着時刻　0, 所要実行時間　6

プロセス2　到着時刻　1, 所要実行時間　20

プロセス3　到着時刻　2, 所要実行時間　1

FCFSとSJFでプロセスの実行開始時刻と実行終了時刻はどうなるでしょうか？

32

平均ターンアラウンド時間

- ターンアラウンド時間（TAT）
  - プロセスが到着してから実行終了までの経過時間
  - 平均ターンアラウンド時間は、各プロセスのターンアラウンド時間の平均値
- プロセスが3個、到着時刻(所要実行時間)
- 0(6), 1(20), 2(1)
- FCFSの実行開始時刻、実行終了時刻、平均TATは？
  0-6, 6-26, 26-27
  TAT 6, 25, 25, 平均18.7
- SJFでは？
  0-6, 6-7, 7-27
  TAT 6, 5, 26, 平均12.3

ターンアラウンド時間（TAT）
プロセスが到着してから実行終了までの経過時間
ターンアラウンド時間は、英語の頭文字を取って、TATとも呼ばれます。
平均ターンアラウンド時間は、各プロセスのターンアラウンド時間の平均値です。

プロセスが3個、到着時刻(所要実行時間)が次のように与えられたとします。
到着時刻(所要実行時間):0(6), 1(20), 2(1)

FCFSの実行開始時刻、実行終了時刻、平均TATはどうなるか？
　実行開始ー実行終了時刻:0-6, 6-26, 26-27
TAT 6, 25, 25, 平均TATは18.7

SJFの実行開始時刻、実行終了時刻、平均TATはどうなるか？
実行開始ー実行終了時刻:0-6, 6-7, 7-27
TAT 6, 5, 26, 平均TATは12.3

平均ターンアラウンド時間は、SJFの方がかなり短くなっています。