オペレーティングシステム

資料　第4分冊(H27)

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

2. プロセス管理
ープロセッサ時間の管理ー

2.1 プロセス管理

- プログラムを効率よく実行する
- 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る

プロセス

- 「実行されるプログラム」のことを**プロセス**という

  プロセス　＝　プログラム＋実行の状態

  – プログラムの実行コード（マシン命令列）と、実行時のデータの集まりから構成される
- 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か？
  – **プログラム**は実行するための命令列やデータの初期値であり、実行の前にあらかじめ決まっている
  – 同一のプログラムを、別々に何個も「実行」させたい（複数の「**プロセス**」が動作していると考える）
  – 「プログラム」とその実行（「プロセス」）は区別すべき

プロセス管理

プロセス管理においてOSが担当する仕事

- **プロセス割り付け**
  - 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）
  - プロセスを作るときに必要な処理
- **プロセッサスイッチ**（process switch）
  - プロセスディスパッチともいう
  - 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る（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）

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

プロセスとプロセッサ管理・制御方式

マルチプログラミング

- １台のコンピュータ上で、複数個のプログラムを時分割制御などにより切り替えることで、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

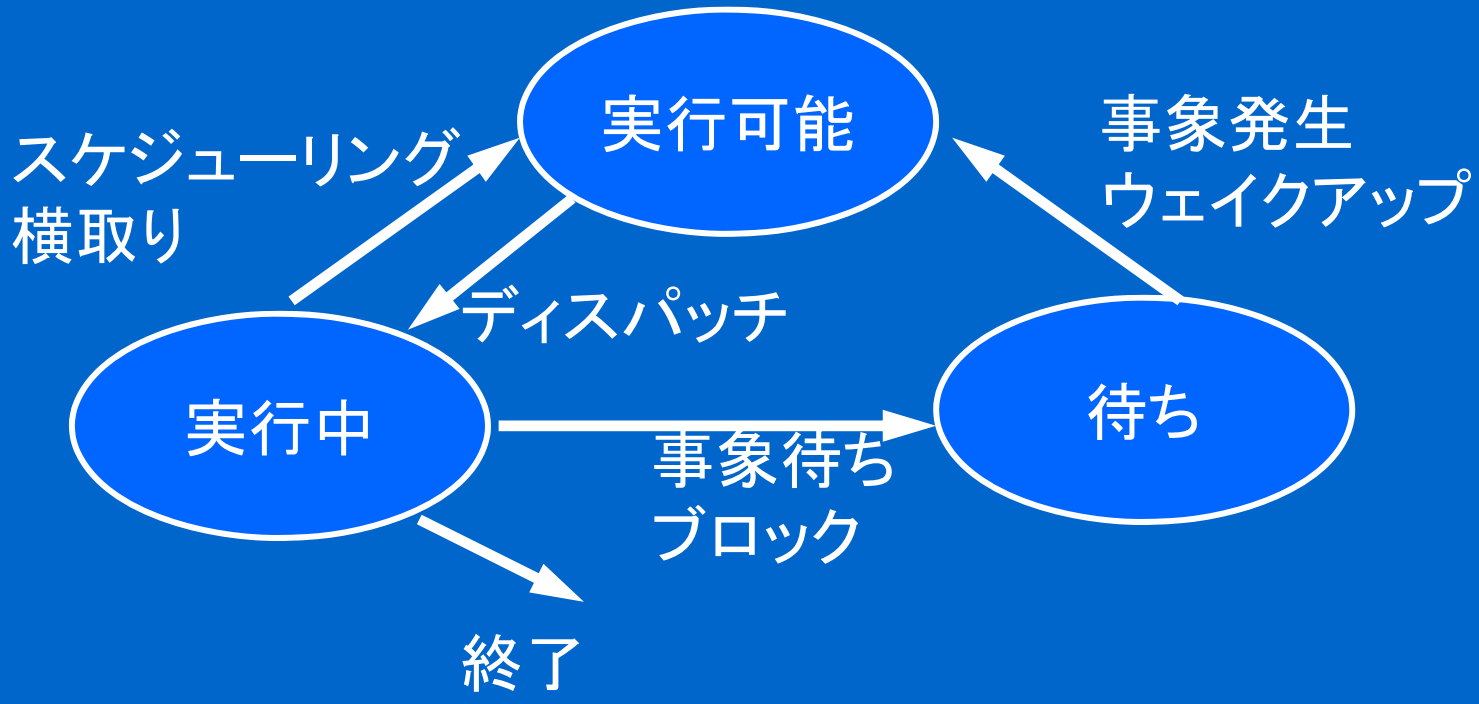
技術的な進展の度合い

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可能
- **待ち**（waiting）
  - 何らかの理由で実行できない
  - 何かを待っている

プロセスの状態遷移（図2．2）



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)

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→実行中の遷移）

ウェイクアップ割り込み

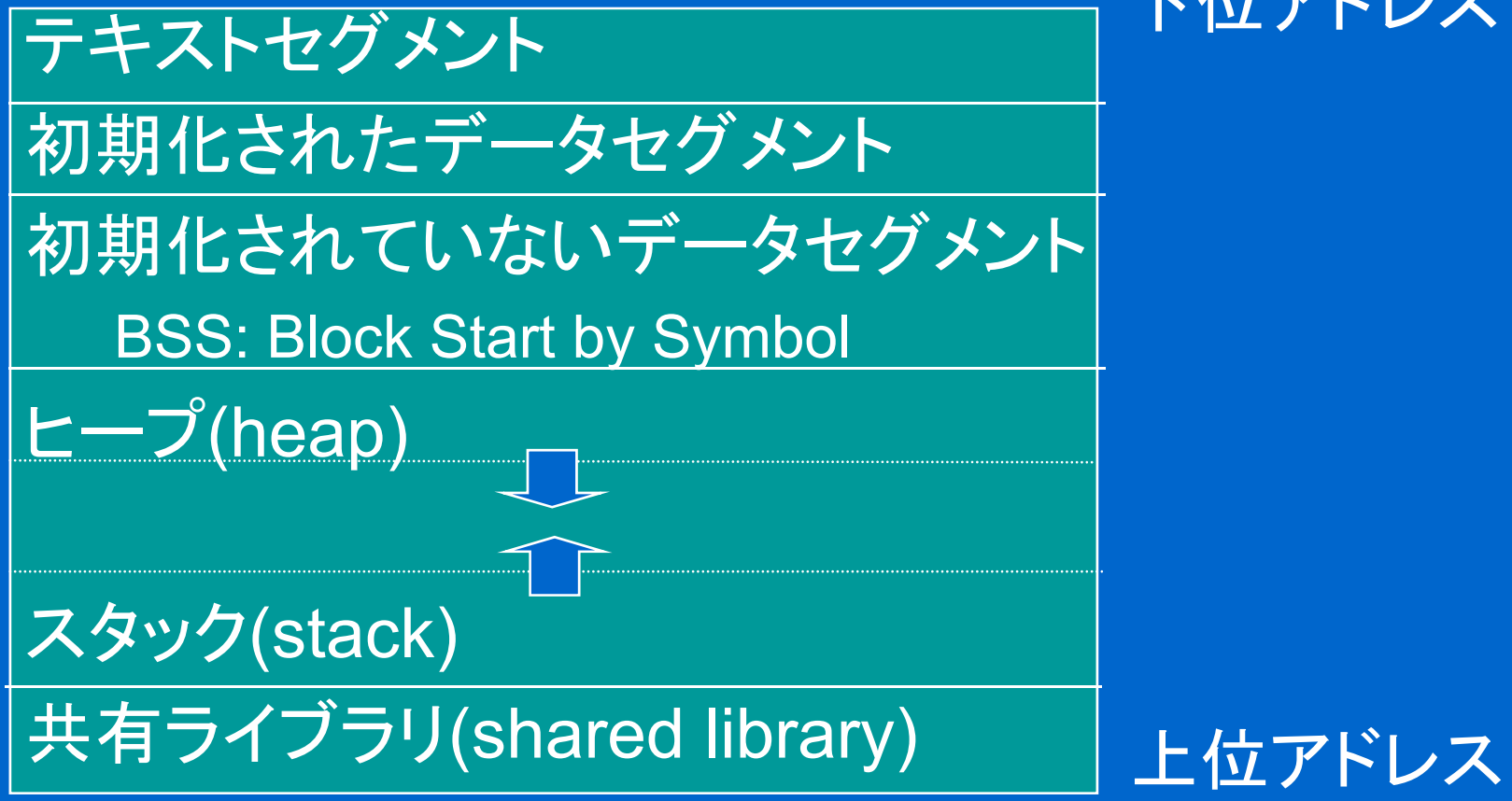
- 割り込みそのものが、事象（ウェイクアップ要因）となっている（外部割り込み）
- プロセスは、ウェイクアップにより、待ち→実行可能に遷移

入出力割り込み（図2.5）

- 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）

プロセス領域（図2．7）

- UNIXの例



プロセス領域(2)

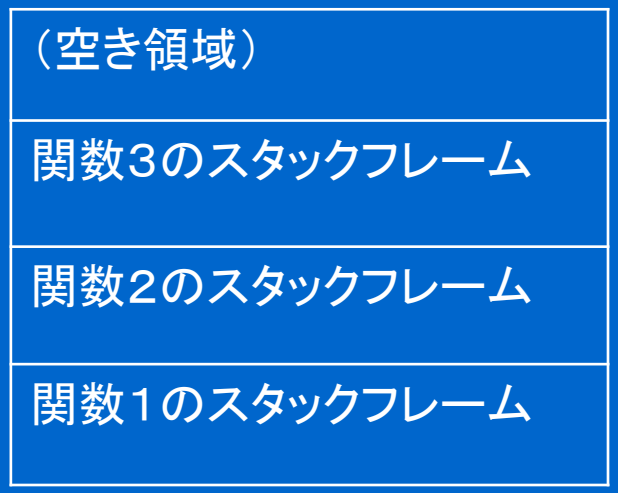
- **コード(テキストセグメント)**
  - ユーザプログラムの命令コードを収めた部分(読み出しのみ)
- **データ(データセグメント)**
  - ユーザプログラムで大域変数として宣言されたデータ領域
  - リンク時に領域が決められ、**実行時**にその領域分のメモリが割り付けられる
- **共有ライブラリ(コードの一部と見なせる)**
  - 複数のユーザプログラムから共通して呼び出されるプログラムコード領域(読み出しのみ)
  - 主にユーザプログラムからシステムコールを行うためのインタフェースや、数値計算などで利用される関数や演算ライブラリなど
  - リンク時に領域が決められ、**実行前**に領域分のメモリが割り付けられる

プロセス領域(3)

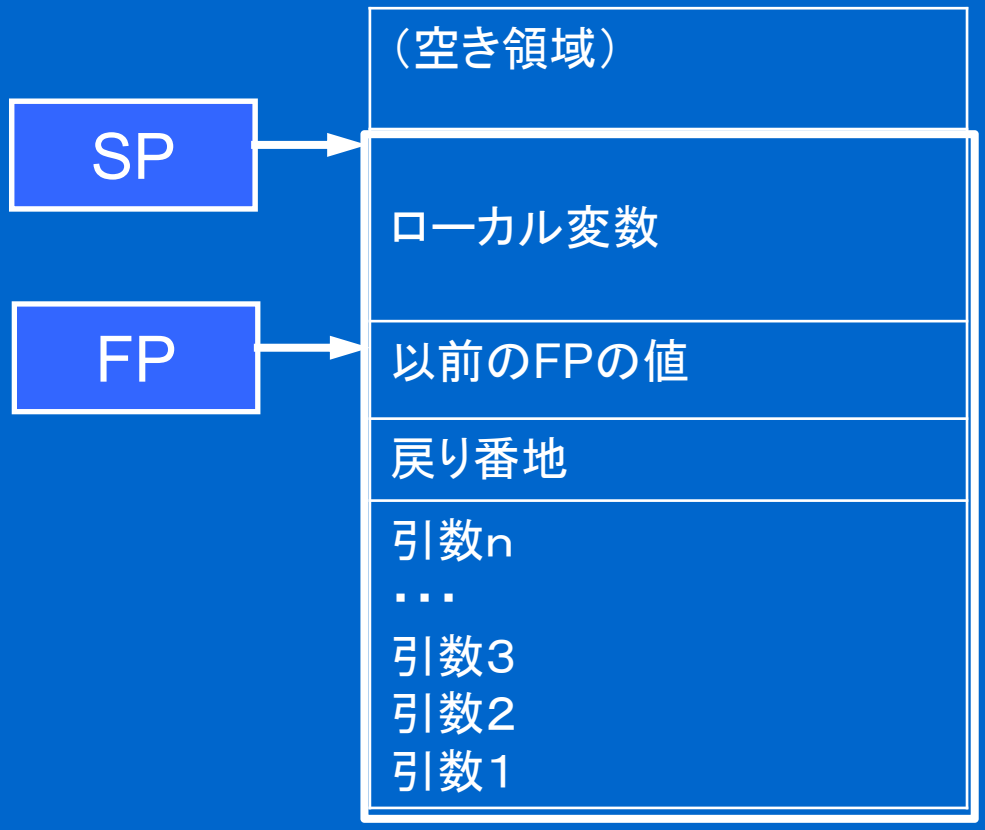
スタック

- 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**

スタックフレームの構造



スタックフレームの使われ方のイメージ図



スタックフレームの構造
SP（スタックポインタ）
FP（フレームポインタ）

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）

プロセス領域(5)

ヒープ

- プログラム実行時の動的割付け用メモリ領域
  - Cでいうと、mallocやcallocで割り付けられ、freeで解放される領域
- 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**
  - 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る

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

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）
- 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？（つづきは演習問題で）

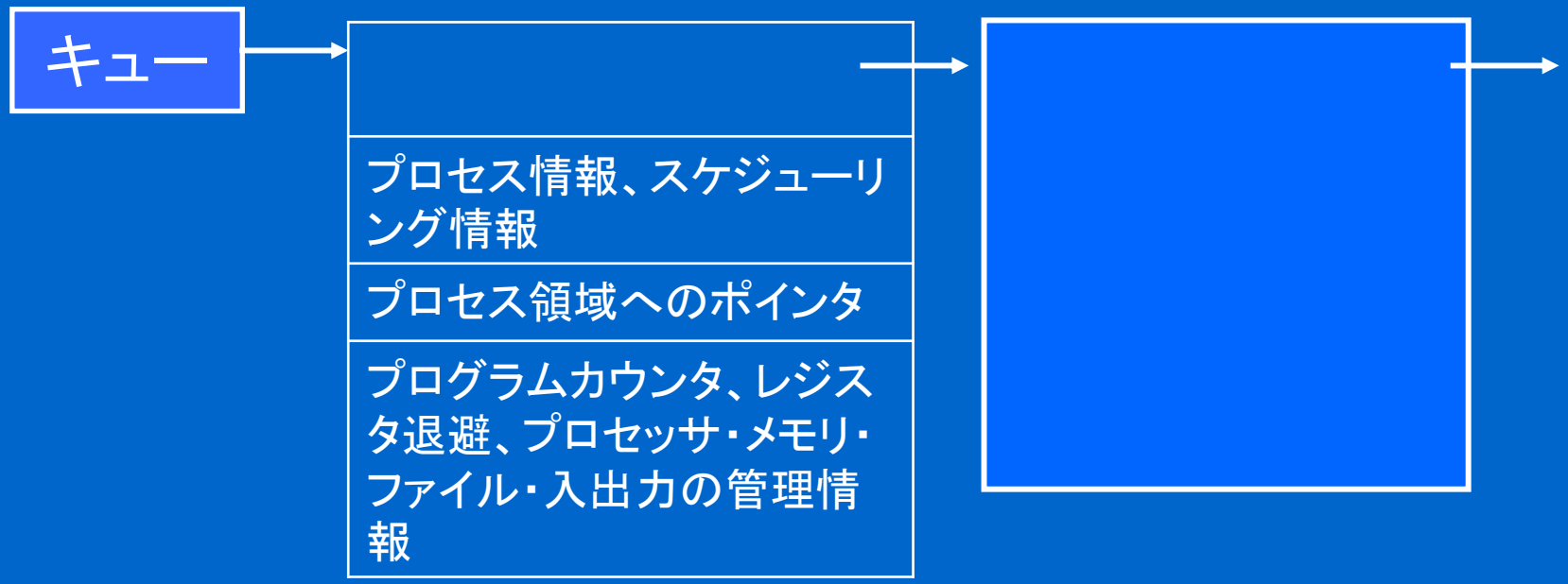
プロセス制御ブロック(PCB)

- プロセススイッチでは、実行中のプロセスの状態を退避する必要がある
  - 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)に格納される

プロセスコンテクスト(process context)と呼ばれることもある

プロセス制御ブロック(つづき)

- PCBはポインタでつながれてキュー(図2.11)を構成する(図2.9, 図2.12)
- キューの先頭のPCBに対応したプロセスから順次実行する



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）

プロセススケジューリング

いろいろな方式がある

- FCFS（到着順）
- SJF（最短要求時間順）
- 優先度順
- ラウンドロビン
- 多重レベル

スケジューリングアルゴリズムの選定指標

- 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）

横取りなしスケジューリングと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、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種類の方式がある
- **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**
  - 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

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）到着したプロセスの順に、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）

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）

優先度順

- 優先順位をつける
- 優先度を高くするプロセス
  - 最短実行時間（実行時間が最短のプロセスを優先）
  - 最小残余時間（「残っている」実行時間が最短のプロセスを優先）

優先度順の欠点

- 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

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
- 新しいプロセスが到着したとき
  - 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
  - **横取りあり**

ラウンドロビン

- 横取りがあることを除けば、FCFSと同じ
- プロセスのディスパッチ後、一定時間で横取り（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

タイムスライスの決め方

- タイムスライスの時間を短くすると、
  - プロセスの切替えが頻繁に起こり、他のプロセスが実行し続けることによる待ち時間が減る（応答時間が小さくなる）
  - 短くしすぎると、プロセススイッチばかりに時間が取られて、プロセスの実行効率が悪くなる
- 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、プロセスの実行効率が上がる
  - 一度プロセスが中断されると、次に実行が再開されるまでに長時間待たされることになる（待ち時間、応答時間が大きくなる）
- 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大きくして実行効率を上げる
  - 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

ラウンドロビンの改良

- 欠点　入出力事象待ちになったプロセス
  ×タイムスライスの間ただ待って過ごす
- 仮想ラウンドロビン(Virtual Round Robin)方式
  - 現在は単に「ラウンドロビン」といえばこの方式
  - 待ち状態のプロセス
    - 実行可能キュー以外の別のキューにつなぐ(wait queueなど)
    - 事象発生（ウェイクアップ）になったらディスパッチ

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

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？

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