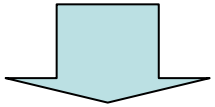
第4章 オペレーティングシステム

ソフトウェアと
オペレーティングシステム

4章の概要

- 情報を処理するため、ハードウエアを全て理解して、機械語でプログラムを実行する？？？
  - “メモリxxxにyyyを書き込んでディスクを呼び出し、ディスクのaaaのアドレスからbbbまで読み出す。転送は1回で行えないので分割して…”

    ->いちいちこんな指令を書いてられるかい！

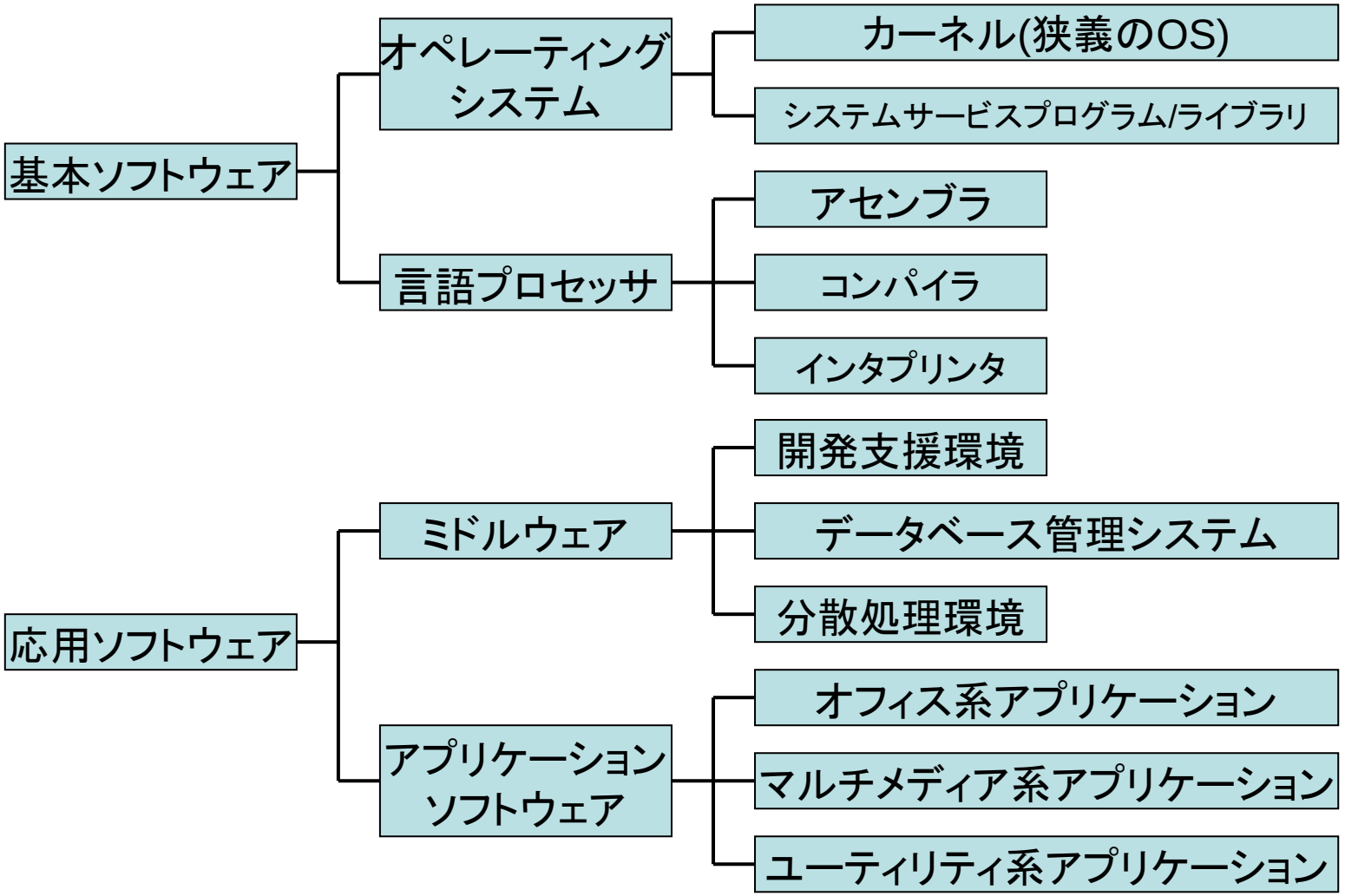


- オペレーティングシステム(OS)を用いて仮想化してしまいましょう
  - “ディスク上のファイルXXXの最初から10KBデータを読む”で勝手に分割されて転送されてくる
- ソフトウェアがOSの階層をどう利用して情報を処理しているか理解しましょう

節題目

- ソフトウェアの分類
- オペレーティングシステム(OS)
- OSの管理機能
  - プロセス管理
  - ファイル管理
  - メモリ管理
  - 入出力管理

4.1 ソフトウエアの分類

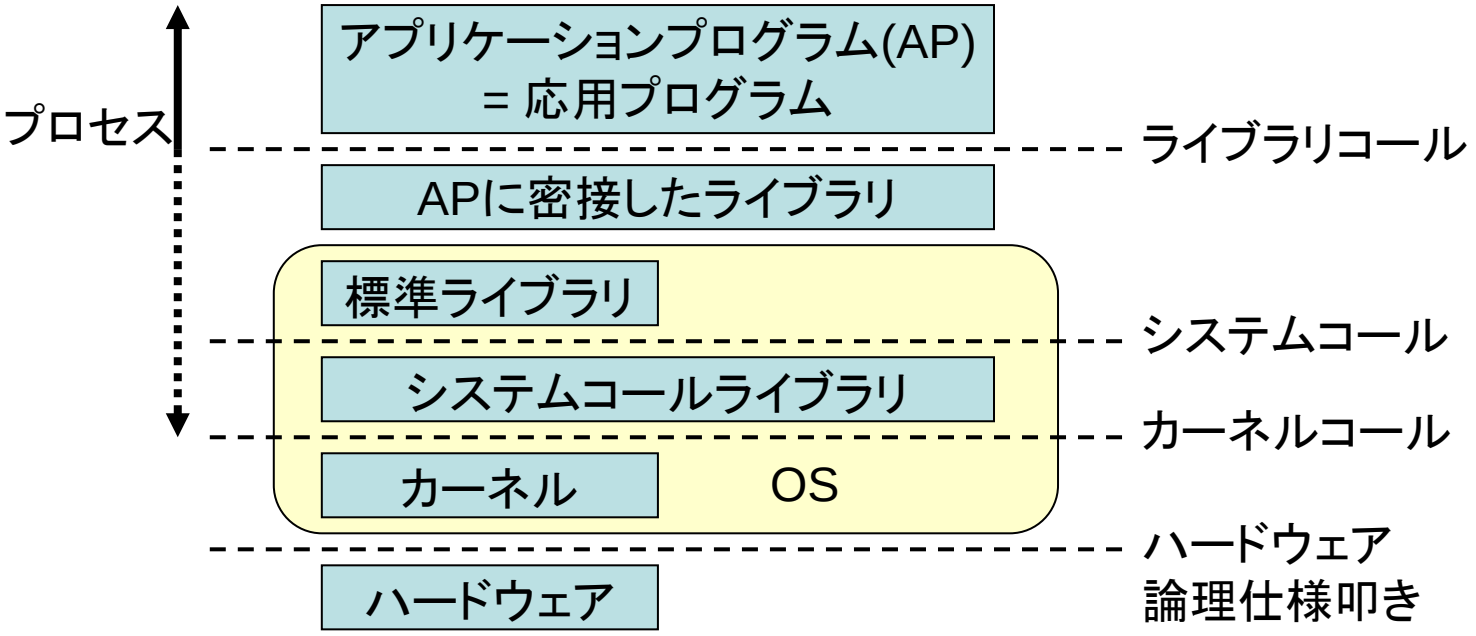


ソフトウエアの分類

- 基本ソフトウェア
  - オペレーティングシステム(OS)
    - 狭義にはカーネルのみをOSとする (例: Linuxカーネル)
  - 言語プロセッサも基本ソフトウェアに含めることもある
- 応用ソフトウェア
  - アプリケーションソフトウェア
  - ミドルウェア： 開発環境
    - 後で説明する、アプリケーション寄りのライブラリもこの分類

ハードウエア～アプリケーションまでの階層

- 下位の機能は基本的にライブラリを通して利用
  - やる気＆特権があるならば、直接ハードウェアも叩ける
- この先で出てくる処理単位プロセスはAP+ライブラリの階層に存在



(a) カーネル

- Kernel: 中心部、(豆の)さやの中の種子
- OSの中核となる機能を提供
  - システム制御
  - 実行管理
  - 入出力制御
  - ファイル管理
- カーネルの機能はカーネルコールで利用
  - 一般のプログラムはカーネルの許し無しにカーネルの機能を実行ことはできない
  - 一般ユーザはシェルを介してカーネルに指示

(b) ライブラリコール

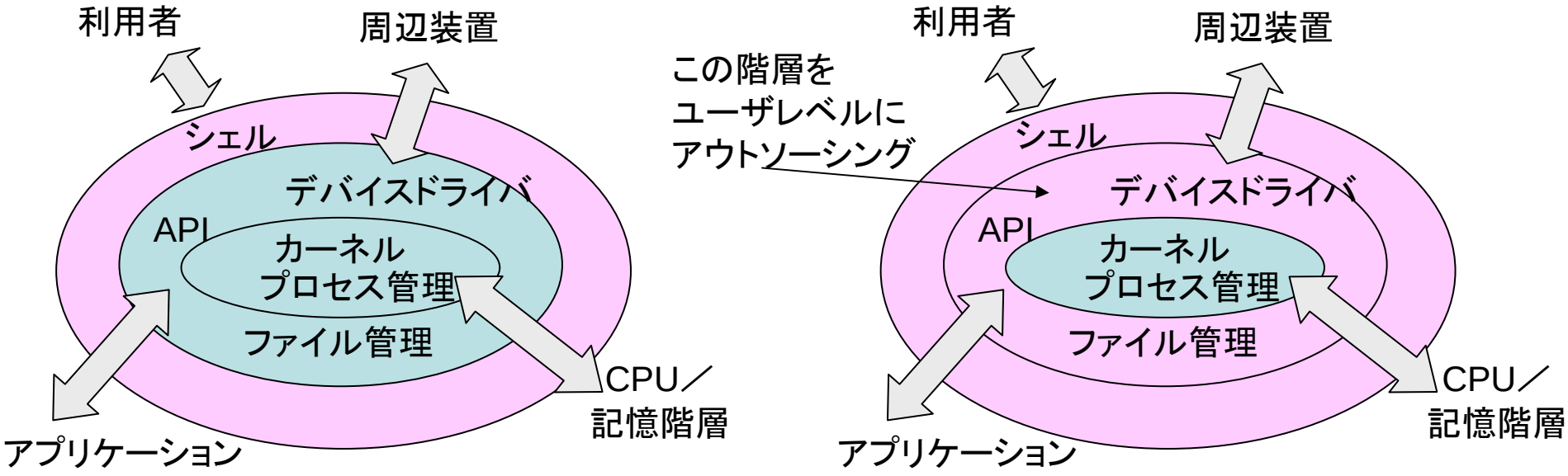
- ライブラリ
  - 特定の機能を、機能の仮想化の手段の1つ
  - カーネルコールをC言語などの高級言語によるインタフェースにするものは特にシステムコールライブラリと呼ぶ
  - 特にOSに標準でついているものを標準ライブラリと呼ぶ
- ライブラリコールの違いのイメージ
  - (標準)ライブラリコール
    - 「この文字を端末に出力する」という要求
  - システムコール
    - 「メモリの指定した領域のデータを指定装置に出力」と要求
  - カーネルコール
    - 「カーネルコールの種類として出力をレジスタ1、パラメータとして装置をレジスタ2、に格納し、指定領域の先頭アドレス…」と要求

4.2 オペレーティングシステム

- 狭義にはカーネルのみ
  - とういうか、Kernelしか存在しない物(Linux)も...
- 広義には、カーネル+API+利用者インタフェース
  - 例
    - Windows XP: NTカーネル＋Windows API＋Windows エクスプローラ
    - Linux: Linuxカーネル＋(GNOME API)＋(bash)
  - WindowsはブラウザIE独占のためにInternet Explorerも OSの一部に組み込んだり...

マイクロカーネルと
モノリシックカーネル

- モノリシック(一枚岩)カーネル
  - デバイスドライバなども含めて一枚岩なカーネル
  - 古典的だが、性能面で有利なので、いまだに使われている
- マイクロカーネル
  - デバイスドライバなどをユーザレベルにアウトソーシング
  - カーネルの安定性、セキュリティが向上する



(1) 広義のOSの役割

- ハードウエア資源の有効利用(リソース管理)
  - スループット（単位時間あたりの処理量）の向上
  - ターンアラウンドタイム（処理の依頼をしてから結果を入手するまでの時間）の短縮
  - プロセス管理、ファイル管理、記憶管理、入出力管理で実現
- 利用者へのインタフェースの提供(抽象化)
- API(Application Program Interface)の提供(抽象化)

(1) 広義のOSの役割

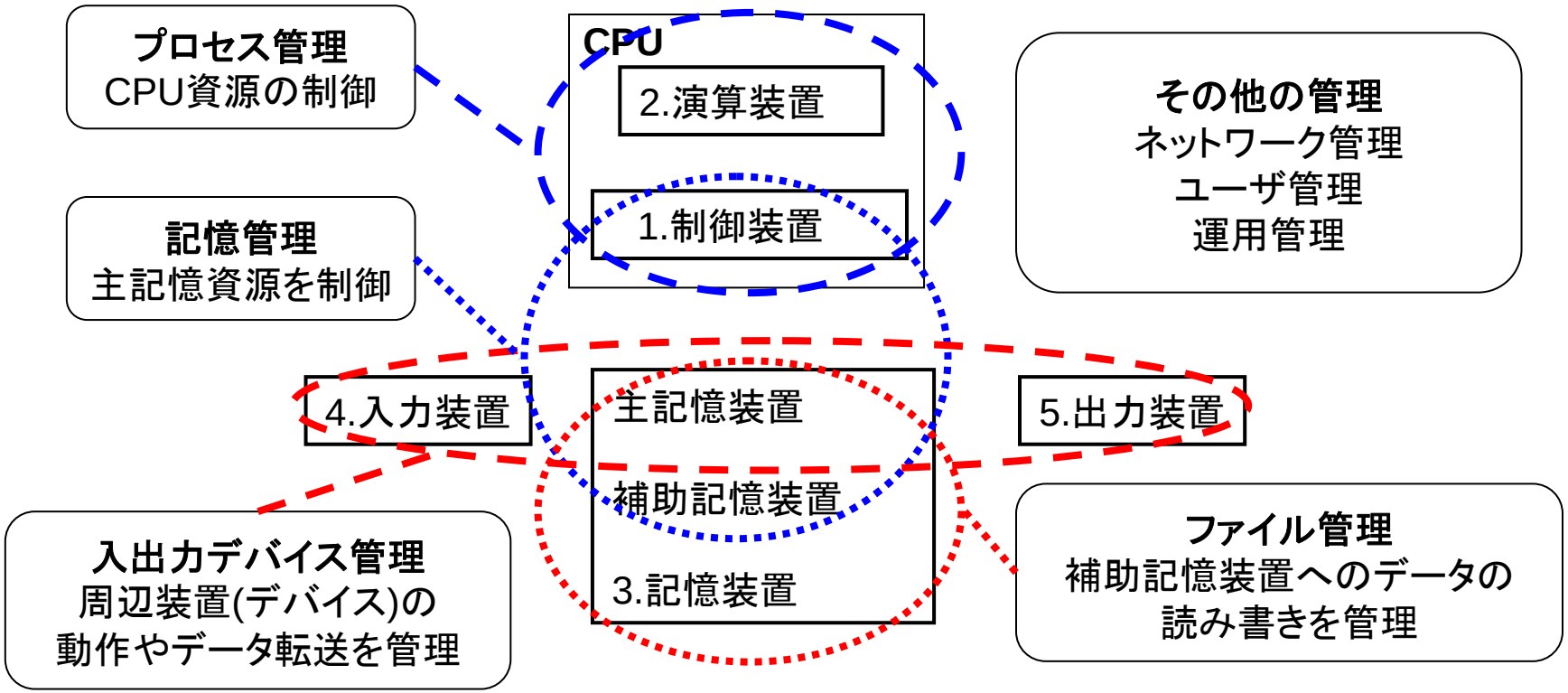
- ハードウエア資源の有効利用(リソース管理)
- 利用者へのインタフェースの提供(抽象化)
  - ハードウエア構造を意識することなくコンピュータシステムを利用できるための機能を提供
  - 例
    - メインフレームのジョブ管理のマスタスケジューラ@メインフレーム
    - UNIXやWindowsのGUI(グラフィカルユーザインタフェース)
    - UNIXのシェル(sh, bash, csh)
- API(Application Program Interface)の提供(抽象化)
  - アプリケーションプログラムから呼び出せる関数などを提供
  - アプリケーションプログラムの開発者がOSの機能を利用できるようにする機能をAPIという

(2) OSの分類

- メインフレーム用OS
  - 企業の基幹系業務に利用
  - ジョブと言う単位でOSに仕事を依頼する点が特徴
- Windows、UNIX系OS
  - 個人ユーザの利用を意識したOS
  - GUIの提供など、コンピュータに詳しくない人に向けたインタフェースも搭載される
- リアルタイム系OS
  - 携帯電話やディジタル用家電など、組み込み用途に利用されるOS
  - データ処理の期限が厳密に決まっているリアルタイム処理のスケジューリングに強い

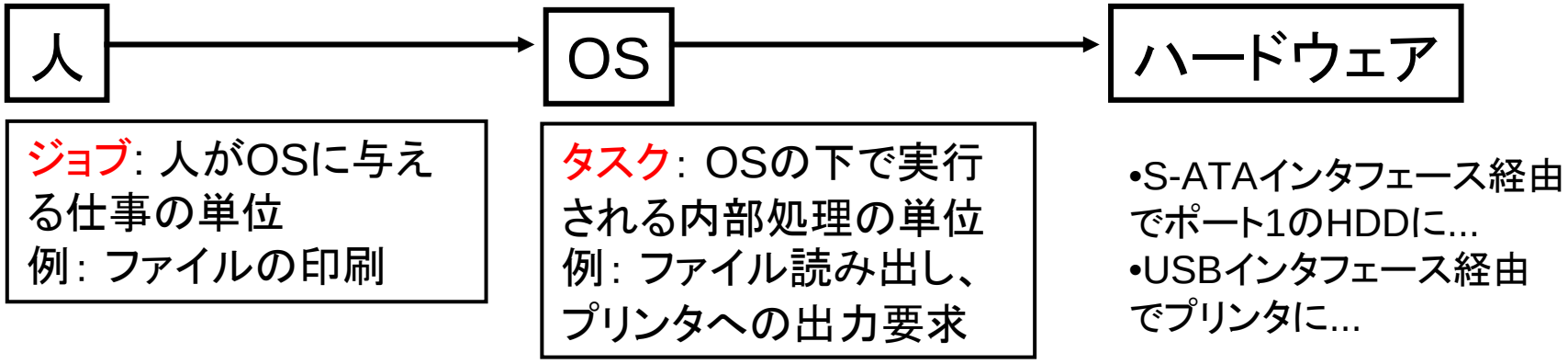
4.3 OSの管理機能

- 計算機の5大機能とOSの管理の範囲
  - 以下、各管理機能を説明する



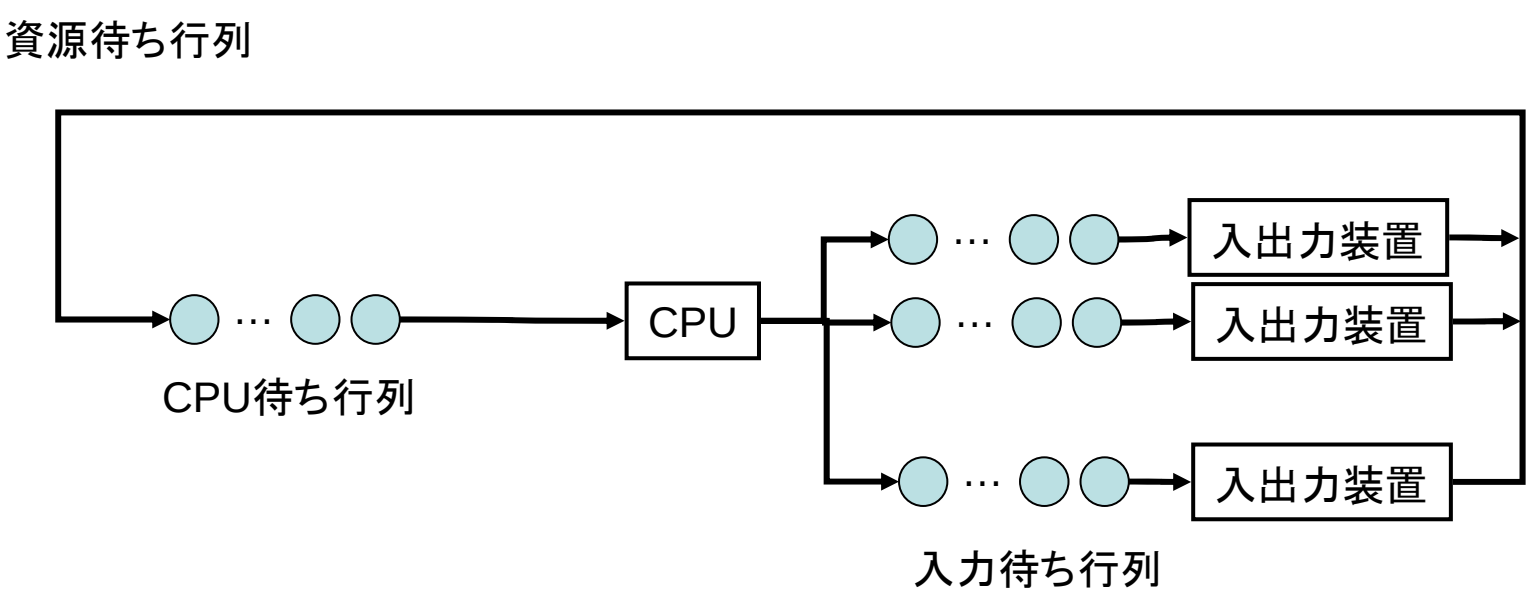
4.3.1 プロセス管理
ジョブとタスク(プロセス)

- ジョブ：人間がOSに与える仕事単位
- タスク：OSの下での内部処理の単位
  - プロセス：OSから許可を受けて実行中のプログラムとおほ同義
  - 1つのジョブが複数のタスクに分割されることもある



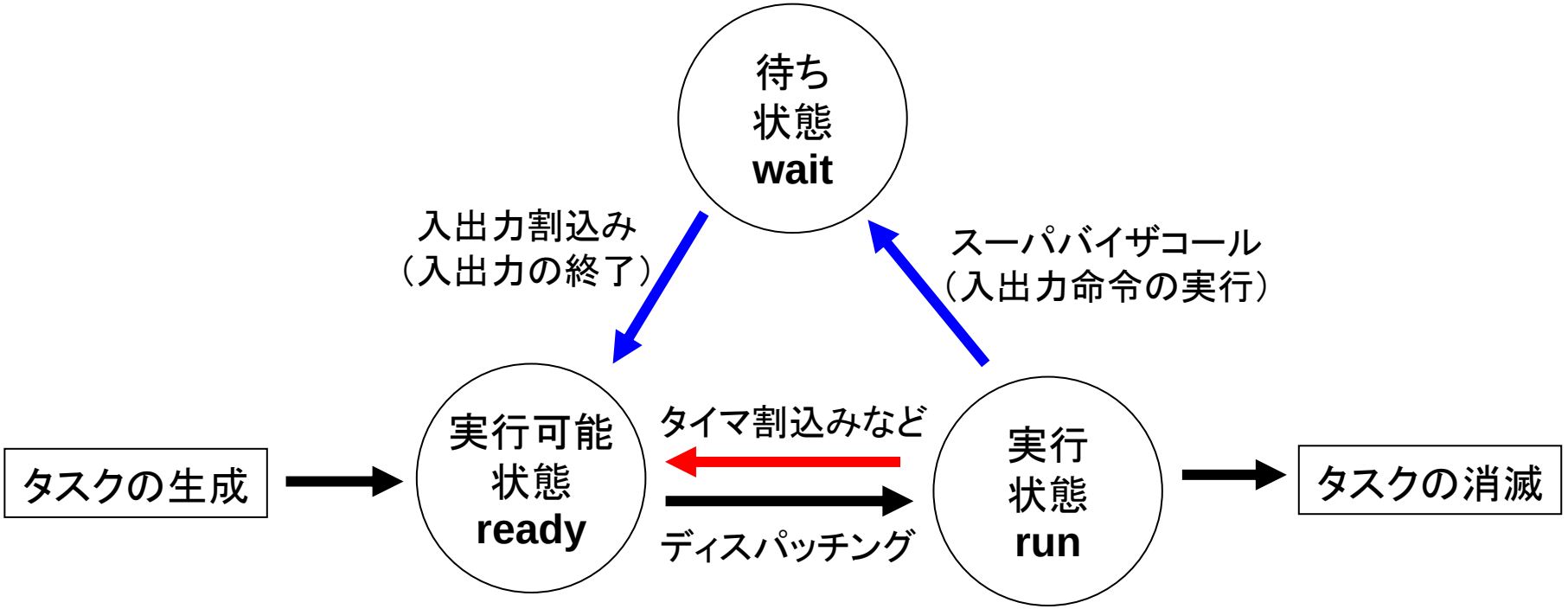
タスク管理の目標

- タスクスケジューリングで以下を実現する
  - トータルのスループットの向上
  - 平均応答時間の短縮
  - プロセス間の公平な資源(CPUなど)の割り当て
  - 応答時間が予測可能性を上げる



(b) タスクの状態遷移

- 実行状態: タスクの割り当てを受けて実行中
- 実行可能状態: 実行準備が整って実行待ち
- 待ち状態: 資源割り当て待ち(主に入出力)



タスクの状態遷移の条件

- CPU利用時間の超過
  - タイムスライス(一つのタスクが連続して利用できるCPU時間)を越えたとき、実行可能状態になる
- 優先順位の高いタスクの実行要求
  - より優先順位が高いタスクが実行可能となった場合、実行状態のタスクは実行可能状態に遷移する
- 入出力命令などの資源確保の要求
  - 入出力の実行やネットワークの利用、別プログラムの呼び出しなど、時間のかかる処理を要求したタスクは待ち状態になり、要求の終了を待つ

(c) 割り込み

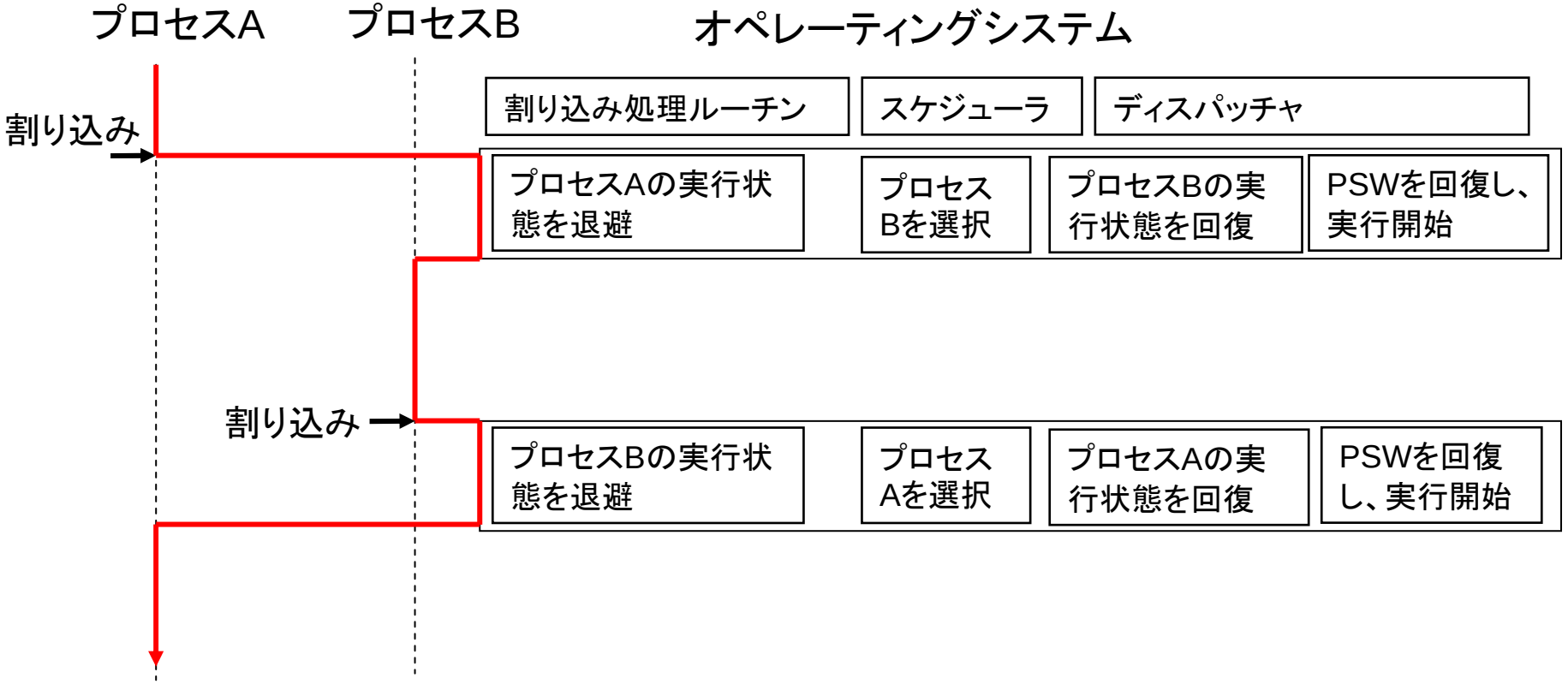
- 実行中のプログラムを一端停止させ、別のプログラムを実行すること
- 割り込みの種類
  - 外部割込み
    - 入出力割込み：入出力装置が指示された処理を終了したことを伝える
    - 異常割込み：電源異常などを通知する
    - タイマ割込み：所定の時間(タイムスライス)の経過を知らせる。

(c) 割り込み

- 割り込みの種類(続き)
  - ソフトウェア割り込み
    - プログラムがOSの機能を呼び出す場合に発生する割込みで、SVC（スーパーバイザーコール）ともいう。
  - 例外割り込み
    - プログラムが実行中に、桁あふれ（オーバーフローやアンダーフロー）やゼロによる除算などが発生した場合に起こる割込み
- 割り込み時の動作
  - 割り込みに対応したプログラムカウンタを設定
  - 実行中のプログラムのCPU状態を保存(レジスタ値など)
  - 割り込み処理のためのCPU状態準備->割り込み処理実行

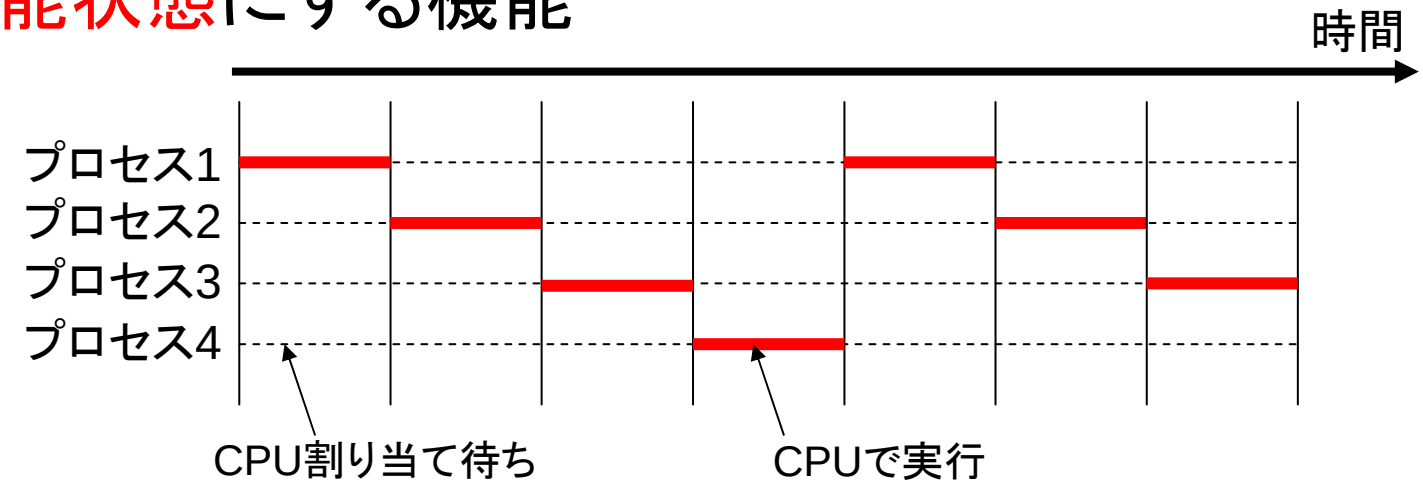
割り込みによるプロセス切り替え

- 割り込み処理の内容を「別プロセスの再開」とすることで、プロセス切り替えを実現可能
  - 他のプロセスの実行中のCPU状態は、以前の割り込みで保存されているものとうる



タイムスライス

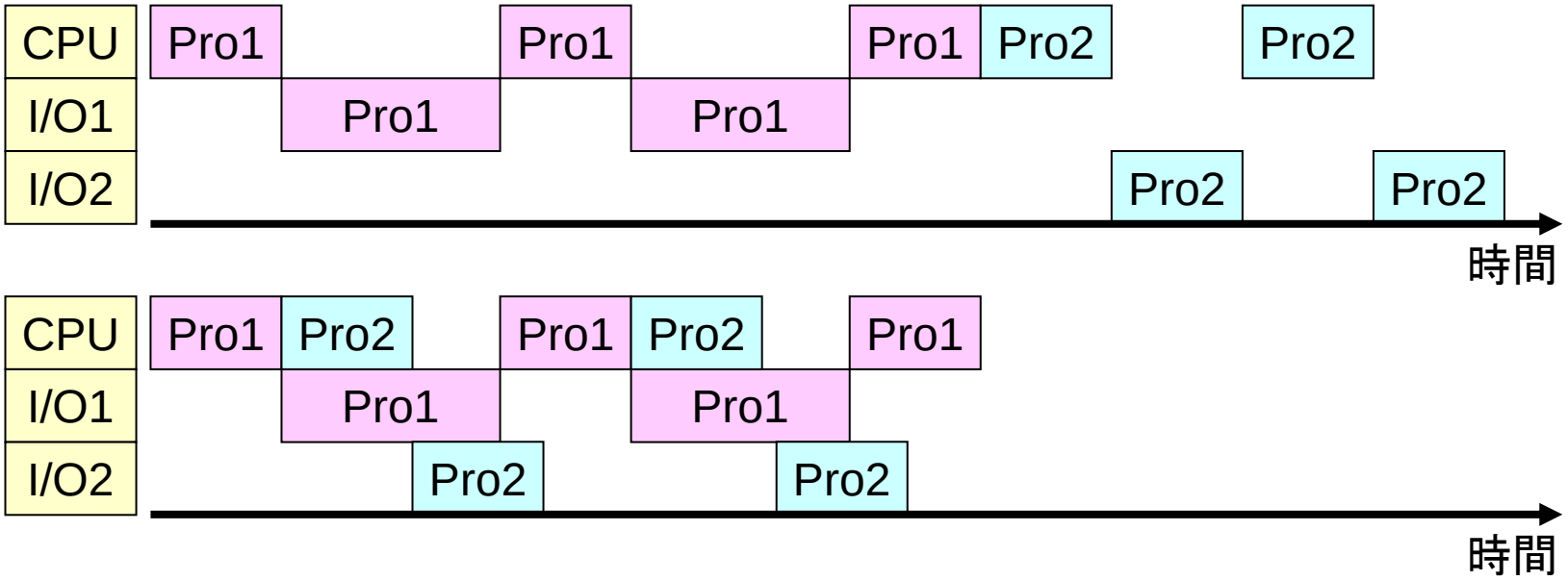
- プロセスが連続して実行できる時間をあらかじめ規定
- その時間を経過すると強制的にそのプロセスを実行可能状態にする機能



- 多くのプロセスに、均等にプロセッサを割り付けることができる

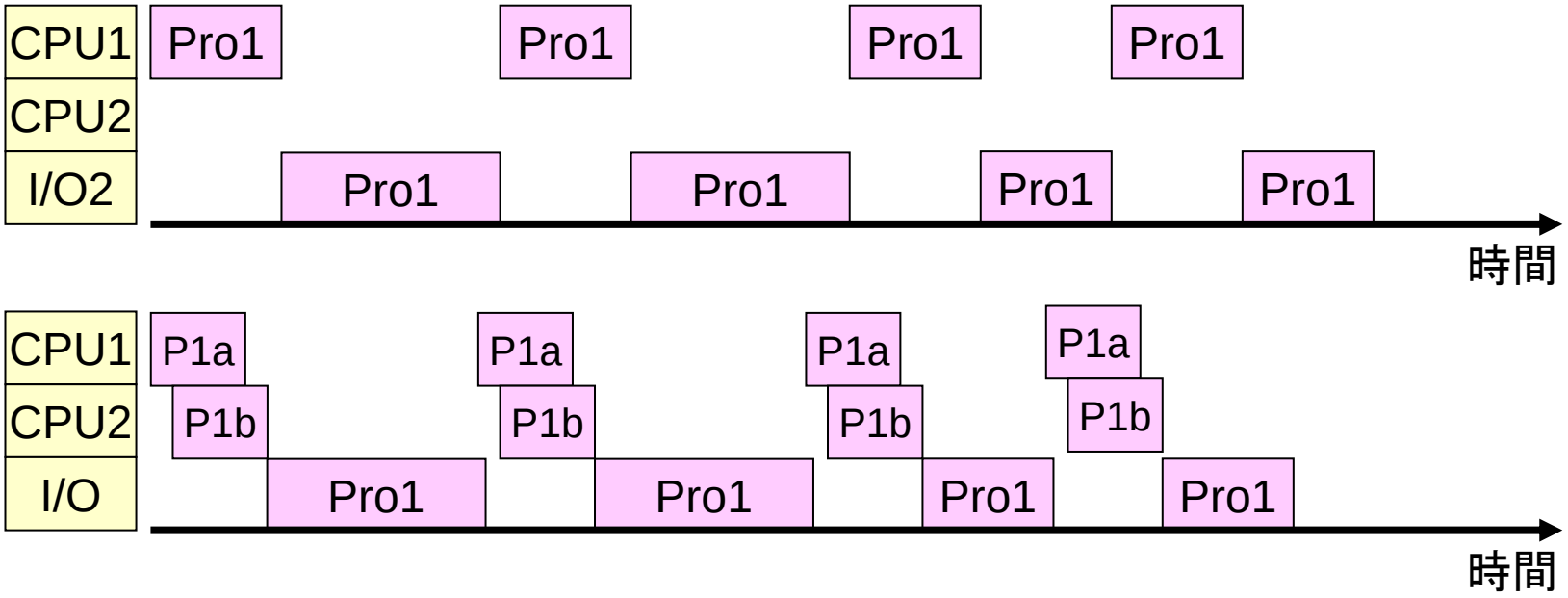
マルチタスク

- 複数のプログラムを効果的に並列実行することにより、スループットを向上させる方法
- 入出力処理などでCPU時間が空いている時に別のプログラムを実行



マルチスレッド

- より細かな単位でプログラムをマルチタスク実行して性能向上をする手法
  - スレッド：1つのプログラム中のCPUの資源割り当て単位をさらに細分化した実行単位
- マルチプロセッサ構成などで利用



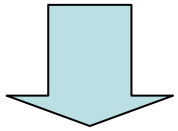
演習

- 以下の場合における、CPUとI/Oの占有状態を示せ。なお、プロセス1の方が先に開始されるものとする。
  - CPU1個でマルチタスクを行う場合
  - CPU2個(CPU1, CPU2)でマルチタスクを行う場合

プロセス1	プロセス2
•CPU 2ms	•CPU 2ms
•I/O1 4ms	•I/O2 6ms
•CPU 6ms	•CPU 4ms
•I/O2 4ms	•I/O1 6ms
•CPU 6ms	

プロセス間通信制御

- 多くのプロセスが勝手にデータをプリンタに出力すると考える
  ->各プロセスの出力結果が混在し、意味のないものになっていまう



- セマフォを用いて交通整理をする
  - 資源が利用可能であるかを示すフラグ(資源数)を準備
  - 資源を使いたい場合は利用要求を出す
    - 資源数があれば、資源を確保したとして確保した分を減らす
  - 資源の利用が終わったら、確保していた資源数を増やす
  - 資源数が不足していたら、開放待ちになる

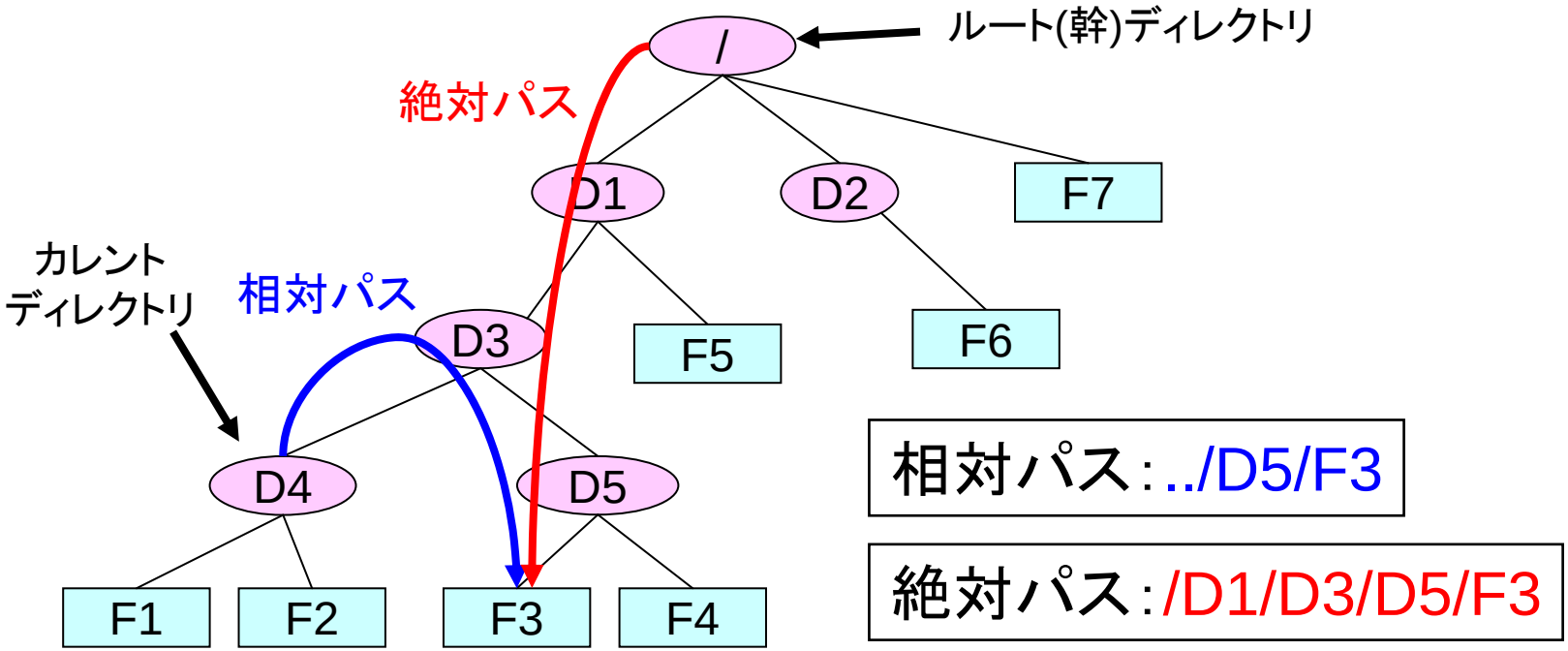
4.3.2 ファイル管理
(1) ファイル編成

- ファイルの配置や内容を物理的にどうするか？
  - 索引容易性と記憶密度をどう両立するか？
- ワークステーション以下ではファイルの内容は単なるビット列
  - アプリケーション側で解釈
- メインフレームでは、ファイルの内容も規定していることが多い
  - 順編成ファイル
  - 索引順編成ファイル
  - 直接編成ファイル
  - 区分編成ファイル
  - 仮想記憶編成ファイル

(2) ファイルシステム

- ファイルの論理的な配置をどうするか？
  - 人間がファイルの場所を示しやすくするためには？

  ->階層的なディレクトリ(フォルダ)構成
- 絶対パス指定と相対パス指定による指定



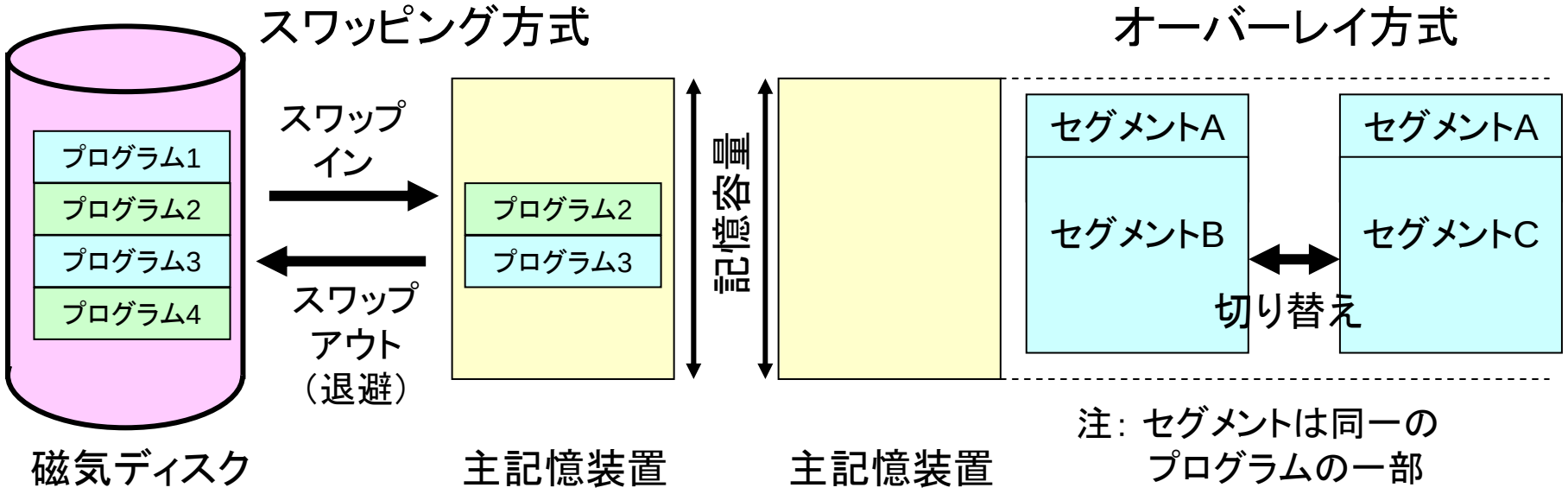
4.3.3 記憶管理

- 実行するプログラム数が多すぎたり、プログラムが大きすぎたりして主記憶に入りきらない場合は？
  ->実行中のプログラムの一部を補助記憶に追い出したり…
- このあたりをマネージメントするのが記憶管理

- 実記憶管理
  - 明示的にプログラムを補助記憶装置への追い出しを行う管理方式
- 仮想記憶管理
  - OSがプログラムのうちの使用頻度が低い部分を自動的に補助記憶装置に追い出す
  - OSがアクセスされたアドレスを主記憶に配置、残りは補助記憶装置に配置

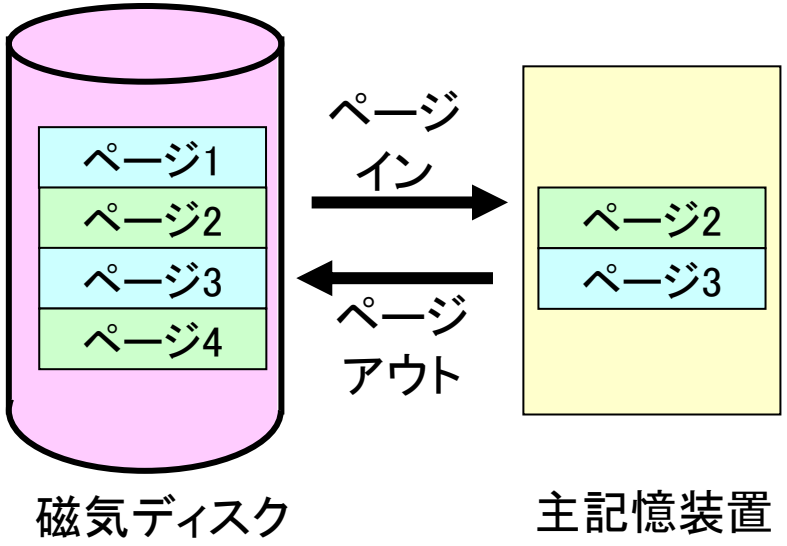
(1) 実記憶管理

- スワッピング方式
  - 複数のプログラムが実行中を想定
  - 実行待ちなどのプログラムを補助記憶装置に追い出す
- オーバーレイ方式
  - 1つのプログラムを分割して補助記憶装置に追い出すことを想定
  - プログラムの一部は管理部として必ず残る



(2) 仮想記憶管理

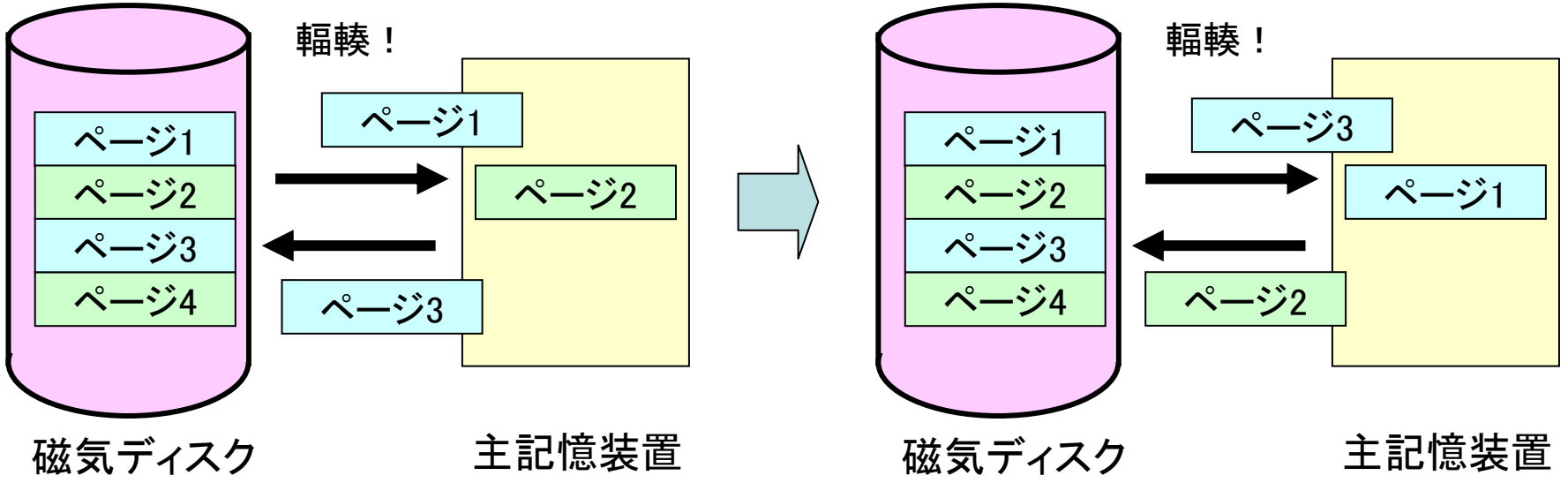
- OSがプログラムをページ単位に分割
  - ページの大きさは数KB～数百KB(OS側で固定値を設定)
- 実行状態となったページを主記憶に置き、残りは補助記憶装置に追い出す
  - ページのやりとりをページングと呼ぶ



スラッシング

- 主記憶がページ分しかない時に、ページ1,2,3に頻繁にアクセスするプログラムが存在したら？

  ->ページの入れ替えが輻輳する
  - これをスラッシングと呼ぶ
- 対策：主記憶を増やす、(プログラムのホットコードの縮小)

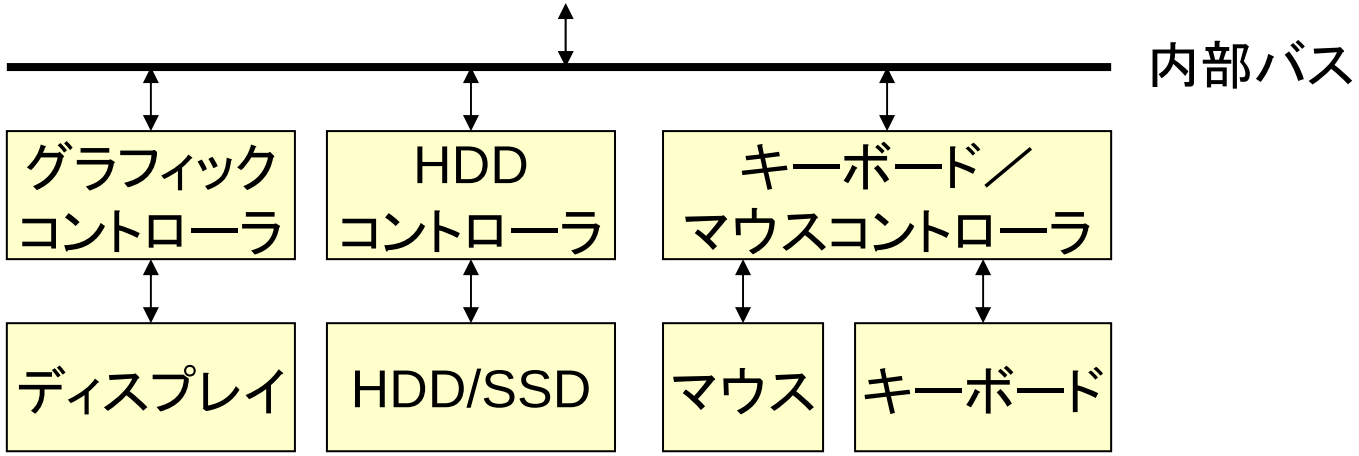


ページ置き換え方式

- FIFO(First In First Out)
  - 先入れ先出し方式
  - 滞在時間の長いページからページアウト
  - 最近読み込んだページほど利用頻度が高く、先に読み込んだページの再利用は少ないという考え方
- LRU(Least Recently Used)
  - 最後に参照してから、最も経過時間が長いページをページアウト
  - 最近参照したページほど再度参照する可能性が高いという考え方

4.3.4 入出力管理

- 入出力の要求を出すのは特に難しくない
- 入出力の終了をどのように検知するか？
  - CPU時間を利用して常に入出力機器を監視している
    - 壮大な無駄があるが、お手軽なので、あながち間違いではない
  - ポーリング：定期的に入出力に終了をチェックする
  - 割り込み：終了時に入出力側から割り込みを入れさせる



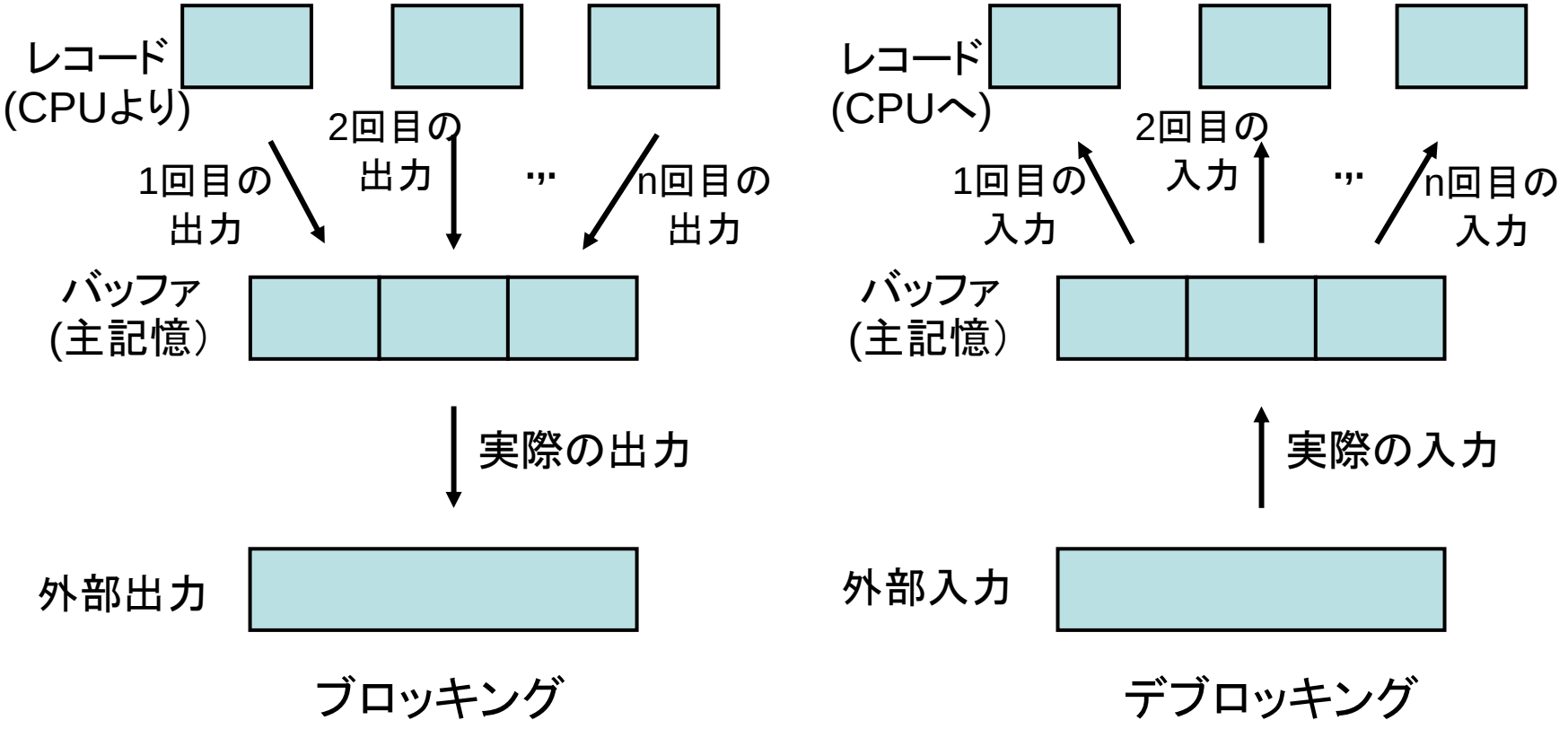
(a) バッファリング

- バッファ(buffer): 入出力のために主記憶上に確保される領域
- バッファリング(buffering): バッファを使って入出力の効率を上げる方法

- データはブロック単位で読み書きするものとし、1ブロックは複数のデータ(レコード)となるものとする。
  - ブロッキング係数n = 一つのブロックに含まれるレコード数
  - 順アクセスではn回のレコードアクセスに1回だけ読み書きすればOK
- 必要に応じて、バッファをフラッシュするのを忘れずに
  - c.f. C言語の標準出力はバッファリングされているので、エラー出力のタイミングを厳密に見たければ、標準エラー出力に出すなりする

ブロッキングとデブロッキング

- ブロッキング：出力時にバッファでデータを結合すること
- デブロッキング：入力時にバッファでデータを分解すること



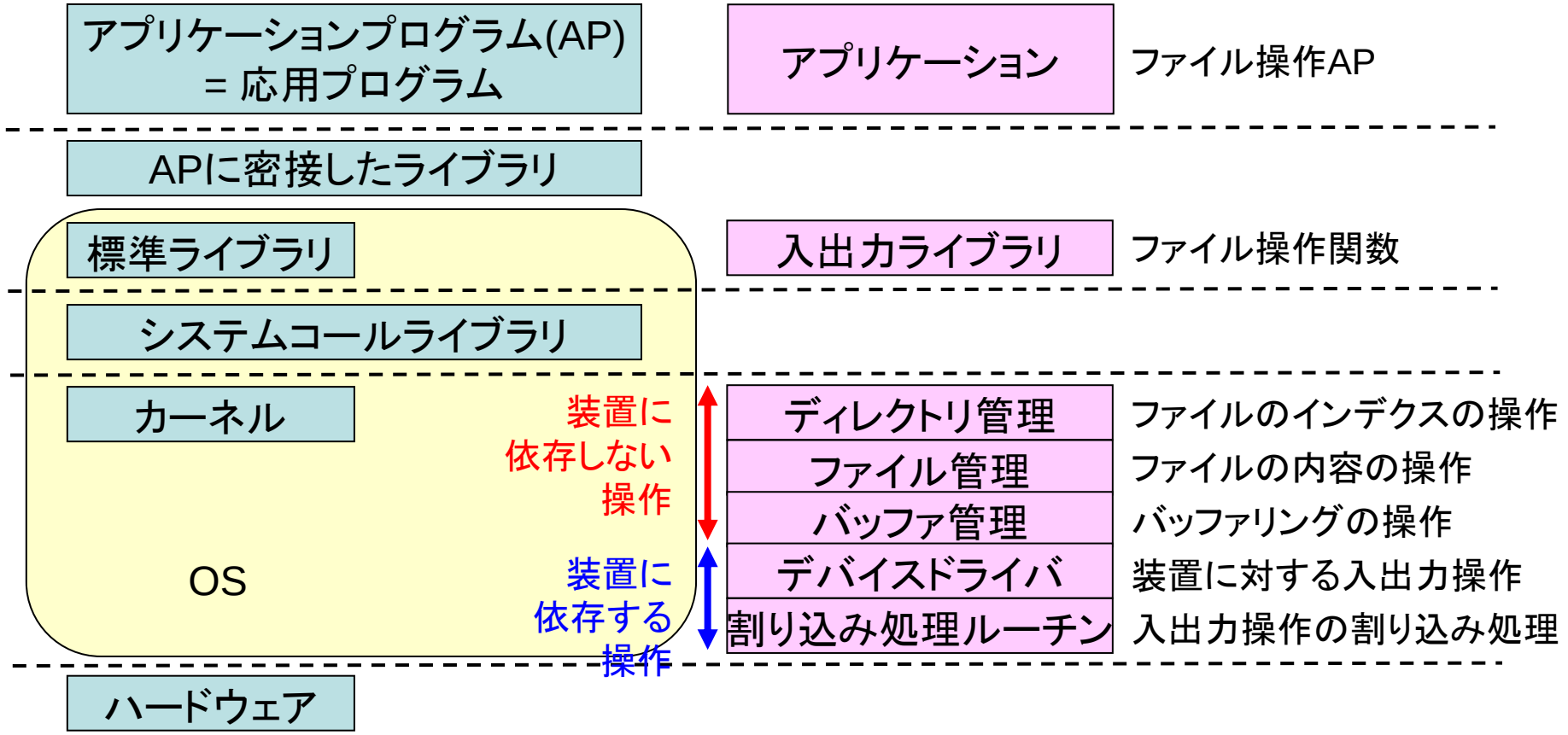
(b) スプーリング

- プリンタなどの低速な入出力装置と入出力データ送受信することを考える
  - バッファがすぐに一杯になってしまい、次の出力データを作れない
  - どうせなら、出力データは一気に作ったほうが効率が良い
- 一時的に入出力データを磁気ディスク装置に格納
  - CPUは磁気ディスク装置をあたかも高速な入出力装置として、データをやり取りする
- スプーリングと呼ぶ

(c) ファイルと入出力を行う実装の例

- 装置に依存する部分と依存しない部分を分けて実装
  - 別に装置に対する実装は、装置に依存する部分のみを実装

    →実装コスト削減、移植性が向上



4章のまとめ

- OSを用いてハードウェアを仮想化できる
  - ライブラリを用いるとさらに仮想化が進む
  - ソフトウェアはライブラリやOSを通してハードウェアを使うことが多い
- OSの役割
  - CPU資源を適切に割り当てるためにプロセス管理をする
  - ディスクのファイルシステムを管理する
  - ICメモリ以上の主記憶を使えるように記憶階層を管理する(仮想記憶)
  - 入出力の競合回避やバッファリングで入出力を管理する