今回は、OS内部の構成法と仮想化についてです。
教科書では、第11章の仮想化が対応します。

オペレーティングシステム
（2024年 第13回）

OSの構成法と仮想計算機について

前回の課題について

詳細は「第12回の課題について.pdf」をみてください

（1）利点としては、すべての文字が2バイトという固定の長さで表現されると処理が単純になることがあげられる。
比較の対象をマルチバイト表現の文字コードとすると、文字によって長さが異なる場合、コードの中を見て判断する処理が必要になるが、2バイト固定では必要ない。

問題点は、ラテン文字が主体であるような文書を格納するような場合に、要する記憶域が増加すること

（2）末尾が 10～59 のいずれかである文字列ということで、まず

*「末尾の2桁」

となり、末尾の2桁を考えると、1桁目が1～5なので [12345]、2桁目が [0123456789] 、
この2つを結合したもので

*[12345][0123456789]

となる。

オペレーティングシステムの構成

- OSは大規模で複雑なプログラム
  - 開発、移植、修正、変更が困難
  - OSでは、機能の変更や拡張が容易に行えることが必要
- 大規模プログラムへの対処
  - モジュール化
    - 機能分割
      機能単位として小さな部分に分割し、それらを合成して開発
      - プロセス管理、メモリ管理、ファイル管理、デバイス管理
        - プロセス管理ではスケジューラとディスパッチャに分割、・・・
    - データ分割
      - 各種の管理テーブル
  - 階層化
    システムをいくつかの層に分割
    - 上位層は下位層が提供する機能で実現

Linuxのカーネルの規模は2500万行、5万ファイルを超えています。このようにOSは大規模で複雑なプログラムであり、それをどのようにして開発するかという問題があります。

プログラム開発においては、規模が大きいものを分割し、複雑なものをその構造にしたがって下位レベルから上位へと積み上げ、2つのレベル間でのインターフェースに集中できるように構成して行われます。(機能が複雑にからみあっているため、容易なことではありませんが)

機能を分割してそれらを合成するわけですが、OSではプロセス管理やメモリ管理といった概念が、かなり大きなレベルでの分割になり、その中でプロセス管理ではスケジューラとディスパッチャに分割する・・・といったようになります。

カーネルの構成法

- カーネル
  基本的サービスを行うOSの根幹部分
  - プログラムの起動の準備や終了後の後始末、必要とするメモリ領域の割当て、CPUで実行するプログラムの切替え、などを行う
- モノリシックカーネル（単層カーネル）方式
  OSの機能、構成要素を単一の（カーネルと同一の）メモリ空間に実装・実行する手法
- マイクロカーネル方式
  最小限の機能をカーネルに置き、それ以外はユーザランドの別個のプログラムで実行
- ハイブリッドカーネル方式
  最小限のもの以外にも一部機能をカーネルに置いたもの

カーネルとはOSの根幹部分であり、基本的なサービスを行うものでした。

このカーネルをどのようにして構成するかについて、大きく三つの方法があります。

モノリシックカーネルは、カーネルのすべての機能を一つに取り込んで構成するものです。マイクロカーネルは、それとは反対に最小限の機能をカーネルに置くものです。ハイブリッドカーネルは、両者の中間的なものになります。

以降で、この3つを説明します。

モノリシックカーネル方式とマイクロカーネル方式について、詳細を別途提供する「参考書_第2.7節.pdf」」 ー 参考書：「オペレーティングシステムの概念」からの抜粋（ただし、「システム呼出し」を「システムコール」、「一枚岩の」などを「モノリシックな」などと変更し、原著第9版の対応部分にある図2.14を引用して追加しています） ー
で確認してください。

基本的な構成法

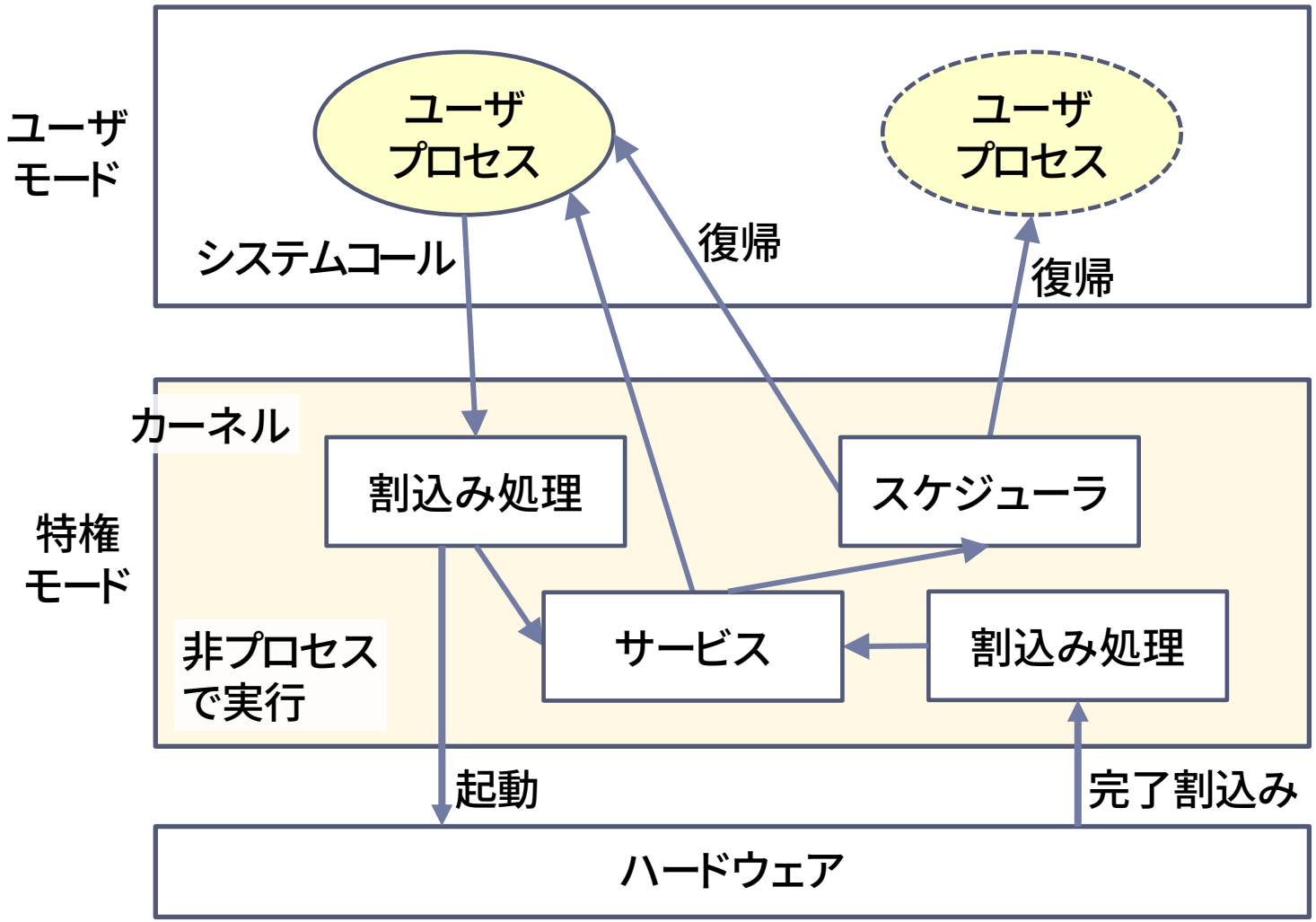
- モノリシック(monolithic)カーネル方式
  - 1つの巨大なカーネルが、カーネルの主要なサービスを実現する
  - カーネルの全機能を特権モードで実行する
  - カーネルデータは共有される
  - 高速
  - OSのバグにより不当なメモリアクセスがなされると、ユーザープログラムが正常に動作できなくなったり、システムダウンにつながる
  - カーネルは複雑で肥大化しやすく、保守が困難
  - 移植性が低い
- モノリシックカーネルのOSの例
  - Linuxや多くのUNIX系OS

モノリシックカーネル方式は1つのカーネルに機能のすべてを取り込んだもので、すべてのモジュールをリンクして1つのプログラムとしています。そのすべての機能は特権モードで実行され、カーネルのデータは共有されます。

このため、高速に実行できるという利点があります。
しかし、共有のデータにすべてアクセスできるので、OSのバグによって不当なメモリアクセスがあるとシステムダウンにつながったりします。(システムコールで実行モードを切り替えているため、ユーザプログラムの誤りによってカーネルへ影響が及ぶことはありません)
また、すべての機能を含むため、カーネルが巨大になり、常には必要でない機能もメモリを占有することになるので、有効利用できないということもあります。複雑になって変更や拡張が困難になるとも言えます。

Linuxや多くのUNIXはこの方式です。

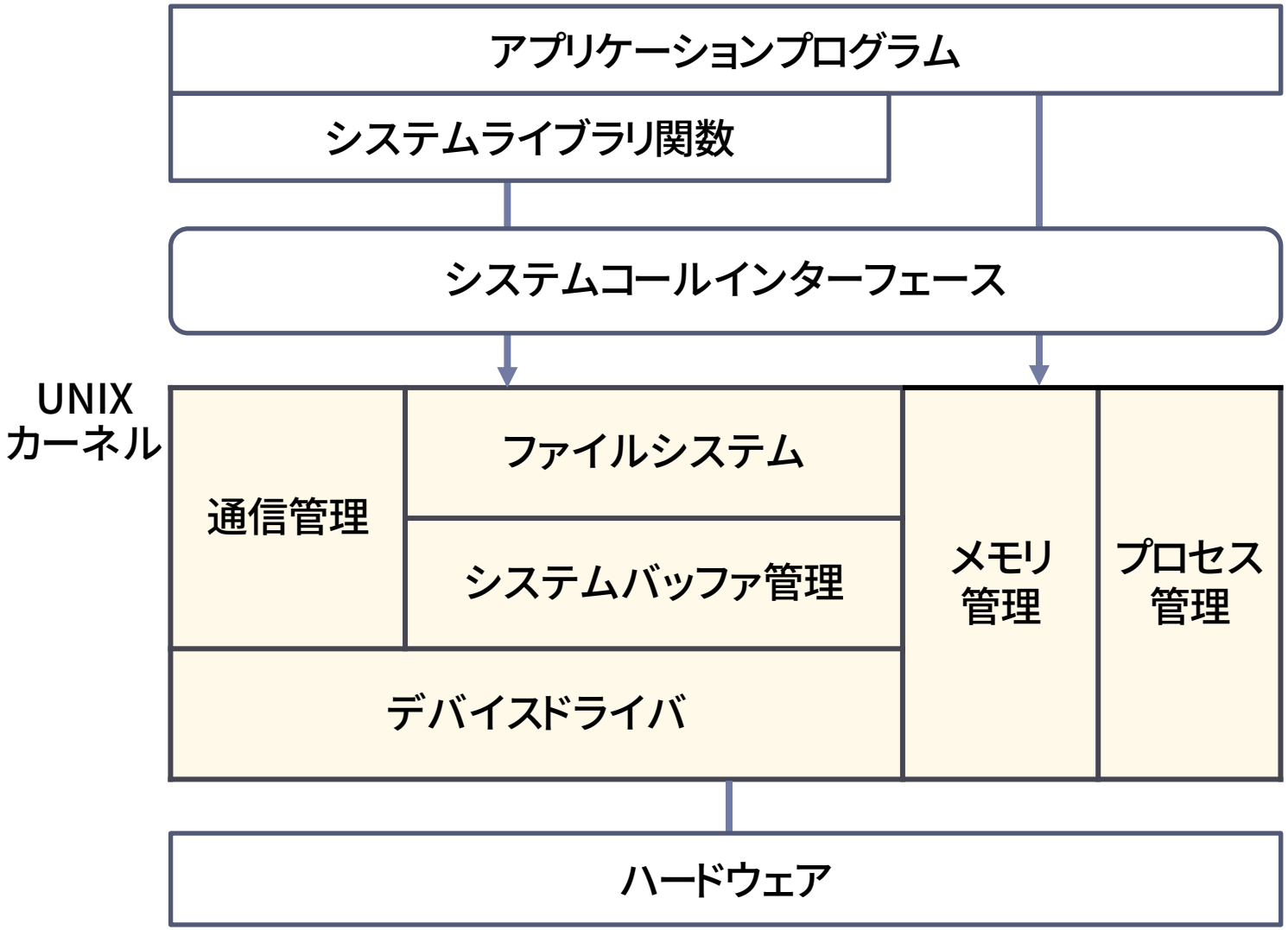
モノリシック構成によるカーネルと制御の流れ



モノリシックカーネル方式による制御の流れです。

ユーザプロセスがシステムコールを出すと割込みによってカーネルに制御が移り、カーネルはすべての処理を特権モードで実行します。OSのサービス（ファイル管理など）を実行した後、ユーザプロセスに戻るか、スケジューラでスケジューリングする（別のユーザプロセスが動く場合もある）かのように動作します。
入出力などはCPUと独立に動作し、完了が割込みでCPUに通知されます。

UNIX系カーネル構成の例



UNIXのカーネル構成の例です。第2回講義資料のp.20や教科書p.51、図4.4と対応づけられるものです。

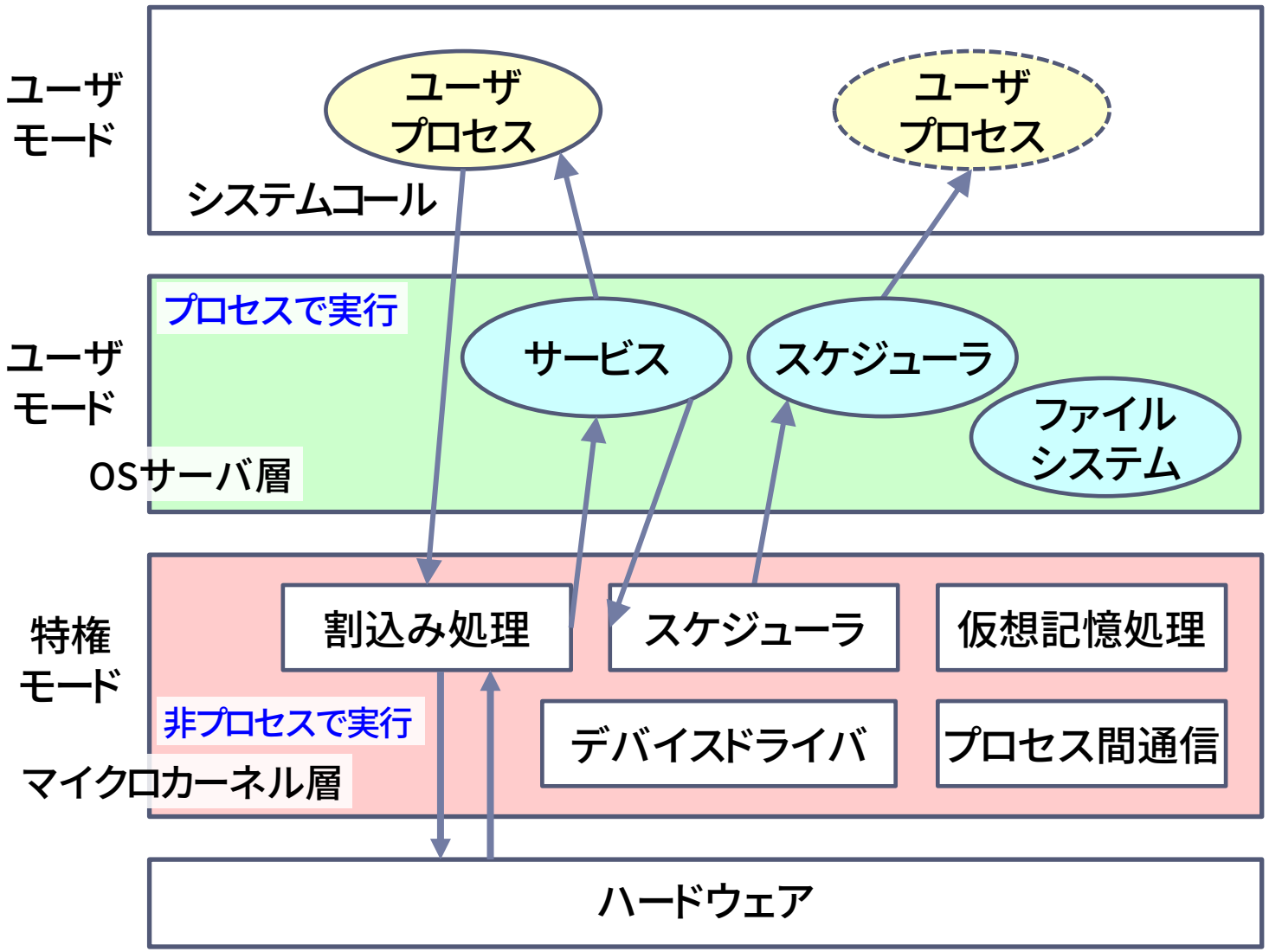
- マイクロカーネル（microkernel）方式
  - 必要最小限の機能を提供するカーネル
  - プロセス指向システム
    論理的なOS機能をプロセスとして実現
    - ユーザプログラムからのサービス要求は、基本的にそのサービスを提供するサーバが処理
      要求はプロセス間通信で伝えられる
    - カーネルは要求を伝えるだけで、実際の処理は行わない
    - プロセス指向システムの利点
      - システムの設計がしやすく、理解も容易
      - 機能の変更や拡張が行いやすい
      - 機能の分散化が行える
    - プロセス指向システムの弱点
      - 機能の呼出しをプロセス間通信で行うため、プロセス間の同期・通信に伴うオーバーヘッドが大きい

マイクロカーネル方式では、必要最小限の機能を実現するカーネル（マイクロカーネル）とし、残りの機能をユーザレベルのプロセスとして動作させます。

ユーザプログラムからの要求はシステムコールを使用しますが、サーバで処理されサーバがマイクロカーネルと連絡します。サーバ間やサーバとマイクロカーネル間はプロセス間通信を使用して連絡します。

マイクロカーネル方式では、マイクロカーネルとOS機能に対応したモジュール化が明確に行われ、見通しの良い設計が可能となり、変更や拡張も容易です。
その反面、サーバ間やサーバとマイクロカーネル間でのプロセス間通信が多発するため、それにともなうオーバーヘッド（実行が低速になること）が問題となります。

マイクロカーネルによるカーネル構成法



マイクロカーネルによる構成では、3層の構造を持ちます。

p.5のモノリシック方式との差を見てください。
特権モードで動作するマイクロカーネル層には、割込み処理、仮想記憶処理、プロセス間通信、スケジューラ、一部のデバイスドライバなどの基本機能だけが含まれます。
その上位に、OSサーバ層として各種のサービスを提供する機能がプロセスとして実現されています。この部分はユーザモードで動作し、モノリシックなカーネルで持っている各種の機能が配置され、アプリケーションのプロセスに対してサービスを提供します。

サーバ層で各種のプロセスが連携して動作するにはプロセス間通信を行う必要があり、システムコールを発行してマイクロカーネル層に行き、そこから戻ってくるという手順が、直接呼び出せるモノリシックと違って余分になりオーバーヘッドになります。

- マイクロカーネル層　ハードウェアとの接点
  - 最も基本的な機能のみ保持、特権モードで実行
    割込み処理、プロセススケジューラ、仮想記憶管理、デバイスドライバ
- OSサーバ層　OSサーバをプロセスとして実行、プロセス間はメッセージ交換で通信
  - カーネルの各機能を配置し、アプリケーション層のプロセスにサービス提供
- マイクロカーネル方式の例
  - Mach、MINIX、QNX

- ハイブリッド型構成法
  - モノリシック構成法とマイクロカーネル構成法の中間的な構成法（マイクロカーネルにモノリシックの特性を一部取り入れて拡張）
    … 非特権モードで実行できるOS機能を極力増やす
  - ハイブリッド方式の例
    - Windows、macOS

マイクロカーネルの代表的なものは、米国カーネギーメロン大学のMach（マーク）やMINIX、商用OSではQNXなどがあります。

ハイブリッド型は、名前のとおりモノリシックとマイクロカーネルの中間的な構成法で、モノリシックの欠点を補うマイクロカーネルの特徴を生かしながら、性能に影響のある部分をカーネル内に取り込んだものです。

非特権モードで実行できるOS機能を極力増やすため、たとえばファイルシステムは、論理的なインターフェース部分とディスク装置などの操作を行うデバイスドライバ部分で構成されるので、論理的なインターフェース部分を非特権で実行できるユーザランドで実行するのが妥当です。

仮想計算機（仮想マシン）

- 1つのコンピュータシステム上で複数のOSを（同時に）実行させたい
  - 古いOSから新OSに移行する経過措置として
  - 既存システムを使用しながら新システム開発をしているとき
  - OS依存のアプリケーションを複数動かさねばならないとき
- このため、複数台のコンピュータを導入することが望ましいが、高価であったり場所の都合などから不可能なことがある → 1台の実コンピュータ上に仮想的なコンピュータを複数提供
- 現在では、CPUパッケージの中に複数のCPUコアが配置され、メモリも大容量なので、1つのコンピュータ上で複数のOSを同時に実行させることが可能
  - 複数のコンピュータで動作していたシステムを1台に集約する
  - あるコンピュータ上で動作しているシステムを別のコンピュータ上に移動させる

動作する実コンピュータの資源をすべて論理化し、仮想的にそのハードウェアのコピーを作る

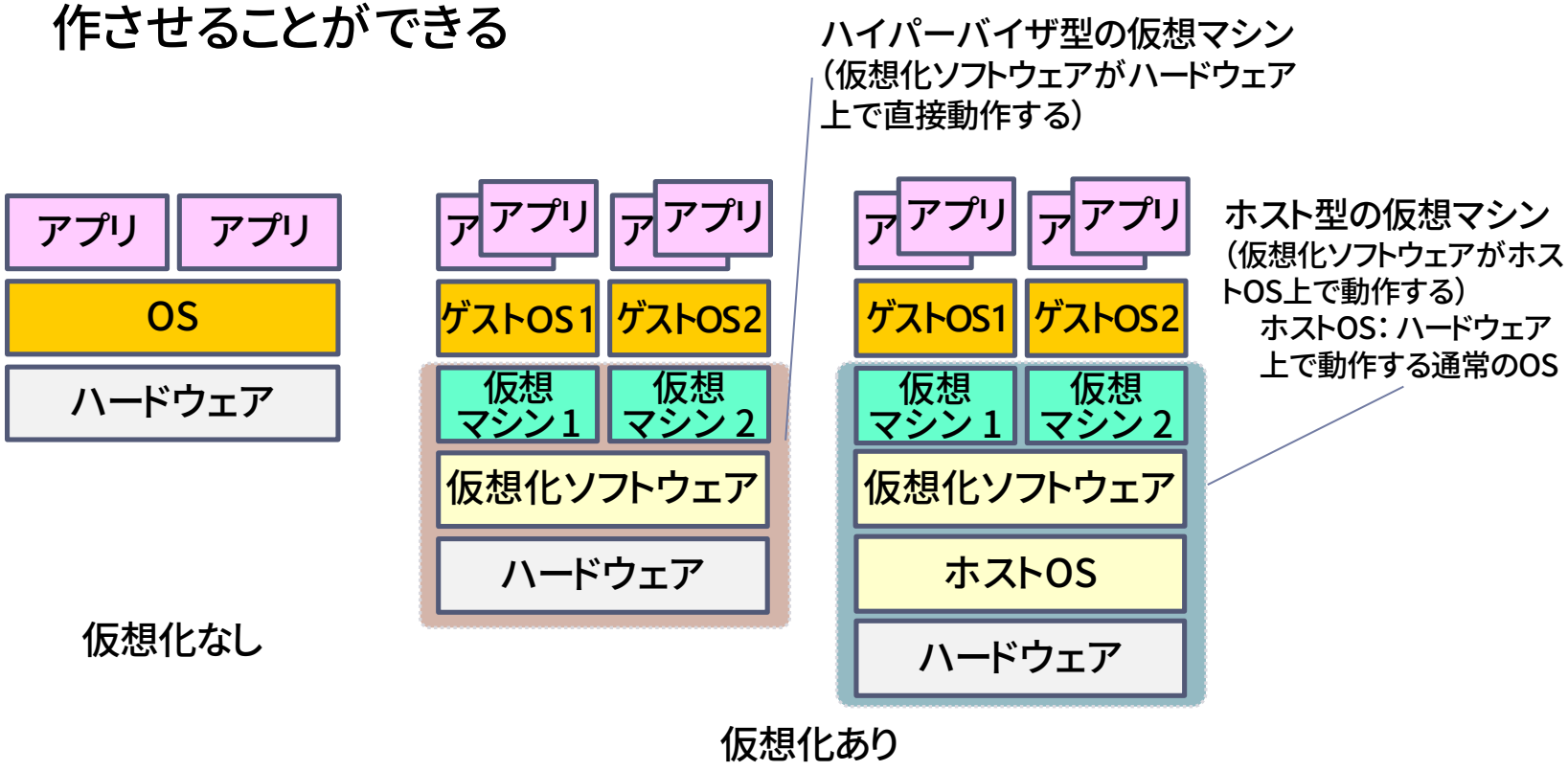
仮想計算機（仮想マシン、Virtual Machine：VM）

どのOSを利用しているかは、通常意識することが少ないかもしれませんが、別なOSを（同時に）使用することが必要になることもあります。このとき、1つのコンピュータ上で複数のOSを使えると便利なことがあります。

仮想マシンはこのような要求に対して、コンピュータの資源を論理化して、1台のコンピュータ上に仮想的なコンピュータが複数台あるかのように見せる仕組みです。

仮想マシン（VM）

- コンピュータの動作を仮想的に構築するソフトウェア
  また、構築された仮想のコンピュータそのもの
- 仮想マシンによって、1つのコンピュータ上で複数のコンピュータやOS（ゲストOS）を動作させたり、別のアーキテクチャ用のソフトウェアを動作させることができる



仮想マシンという言葉で、前のページのように仮想的なコンピュータを示すこともあり、またそのようなことを実現するソフトウェアを指すこともあります。

仮想マシンを構築するソフトウェアがあると、図の左側のように1つのコンピュータ（ハードウェア）の上に、複数の仮想マシンを実現して、そのそれぞれにゲストOS（異なる種類や同じ種類）を搭載し、そのうえでアプリケーションが動作します。
それぞれのゲストOSや仮想マシンは他方とは独立したものです。

ハードウェアを仮想化して、それぞれのゲストOSに仮想のコンピュータ（仮想マシン）を見せるように機能するソフトウェアがハイパーバイザとか仮想マシンモニタとか呼ばれるものです。

仮想マシンの動作原理

- 1台のコンピュータ上に、ソフトウェアによって論理的なコンピュータを複数作り、それぞれにOS（ゲストOS）をロードして実行させる
  - OSは同一でも別々であっても構わない
  - CPU、メモリ、入出力を仮想化（論理化）して提供する
- 仮想化を行うソフトウェア（仮想マシンモニタ：VMM、
  ハイパーバイザ　：Hypervisor）
  特権モードで動作、割込み処理も行う
- 仮想マシンでのプログラム実行はユーザモードで実行
  （ゲストOSはユーザモードで実行される）
- VMMは複数のゲストOSの上位に位置する制御プログラム
  従来のOSが担っている実資源の管理を VMM が行い、仮想マシンのゲストOSは仮想化された資源を取り扱うことになる
  - 従来のOSの考えからするとVMM がOS、VM がプロセスに相当、したがってVMM は各 VM をディスパッチすると考えられる

仮想マシンでは、1つのハードウェア（CPU、メモリ、入出力機器）を仮想化して、それぞれに独立した仮想のCPU、仮想のメモリ、仮想の入出力を提供します。

前のページで出てきましたハイパーバイザはそれを行うソフトウェアです。特権モードで処理が行われます。
その上で実行される仮想マシンはユーザモードで実行されます。ゲストOSもユーザモードで実行されます。すなわち、従来のOSの考えからすると VMMがOS、VMがプロセスに相当することになります。

CPUは時分割して割り当てます。複数のプロセスに対してCPUが時分割で割り当てられたのと同じです。

仮想化でのCPU管理

- 仮想マシンには、実CPUを仮想化した「仮想CPU」が割り当てられる

ゲストOSがユーザモードで動作することで生じる問題点

- 本来OSが扱うべき特権命令の処理
  ゲストOSにおいて特権命令違反の例外が発生した場合に、適切な処理を行って、（複数のゲストOS（VM）が動作していることもある）適切なゲストOSに戻す
- これをソフトウェアで処理するのに、「準仮想化」と「完全仮想化」がある
  - 準仮想化 ではゲストOSのコードの一部を変更し、特権命令が必要となる箇所のソースコードを書き換えて「ハイパーバイザコール」によってVMM（ハイパーバイザ）に処理してもらう
  - 完全仮想化 では、ゲストOSで特権命令違反が検出され例外が発生すると、VMM に制御が移行し、以降の処理を VMM がすべて行って（命令のエミュレーション）元のゲストOSに戻す
    準仮想化よりも処理オーバーヘッドが大きくなるが、ゲストOSを改変する必要はない
    - ハードウェア自体を仮想化するので、IA-32の上でARMなど別のアーキテクチャを動かすこともできる

VMで動作しているプログラム（ゲストOSでも）が特権命令を実行すると例外が発生してハイパーバイザに制御が移ります。（プロセススイッチのイメージです）

通常のOSではアプリで例外が発生すると、OS（カーネル）に制御が移り、カーネルが処理した後、アプリに戻ります。
完全仮想化の場合、ゲストOSがアプリ、VMMがカーネルとして、それに対応付けられます。

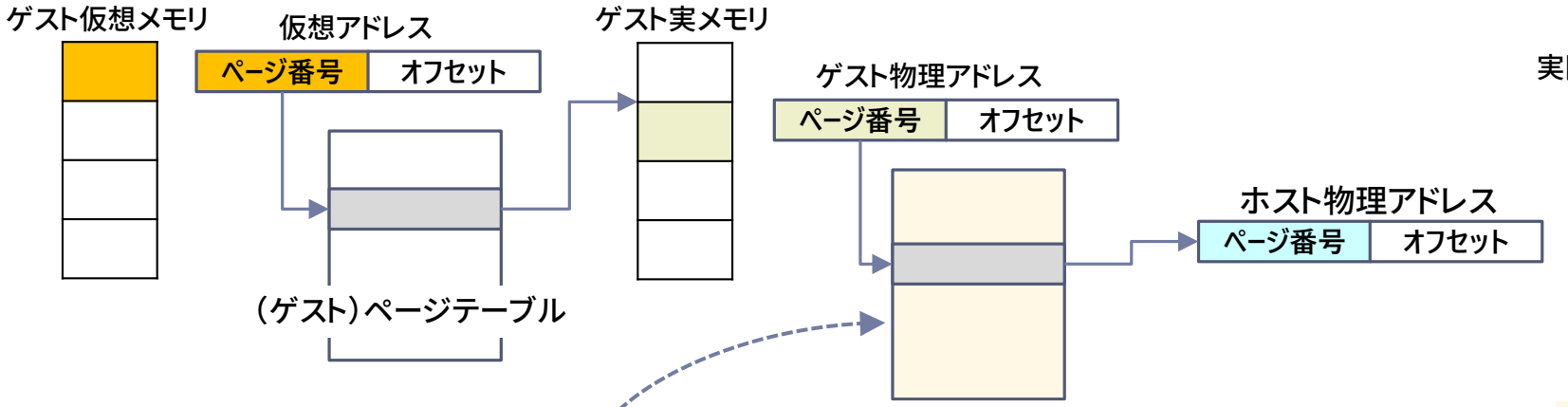
準仮想化では、ハイパーバイザコールを通じて、割り込みハンドラの登録やページテーブルの更新といった粒度の大きい処理単位とするため、ゲストOSとVMM間の遷移回数が少なく高速です。

ハイパーバイザコールという名称は、システムコールにならったもので、システムコールがOSを呼び出すのに対して、ハイパーバイザを呼び出すことからきています。
（教科書では「ハイパーコール」）

仮想化でのメモリ管理

- 仮想記憶方式のゲストOSの場合、2段階のアドレス変換が行われる
  - ゲストOSは「仮想アドレス（ゲスト仮想アドレス）」から、自身のページテーブルを使用して変換し「ゲストが提供する実アドレス」（ゲスト物理アドレス）を得る … このアドレスでは実際のメモリにアクセスできない
  - VMM が「ゲスト物理アドレス」を変換して「ホスト物理アドレス」…実際の実メモリのアドレス を得る

通常のMMUではCPUのアドレス変換は、ゲストページテーブルで行われ、うまく行かない



- VMM は、仮想アドレスをホスト物理アドレスに変換する専用のページテーブルをメモリ上に作る（シャドウページテーブル）
  ゲストOSでページテーブルを参照すると、特権命令実行で VMM に制御が移り、ゲストのページテーブルの内容をシャドウページテーブルに反映する

メモリについては、ゲストOSでのアドレス変換で得られたゲストの「物理」アドレスに対してVMMがさらに変換を行ってホストの物理アドレスを得ます（ここでの「物理」アドレスは、まだ実際のメモリのアドレスではない）

VMMがゲストOSに仮想的なメモリ空間を見せるわけです。（実際のメモリが分割されて割り当てられている）

VMMが管理する実際の物理メモリ空間上のアドレスに変換するには、もう1段階分のアドレス変換テーブルを用意することが必要です。
（1段階のアドレス変換では、2段や3段のテーブルを経由しますが、それをまとめて1段階の変換です）

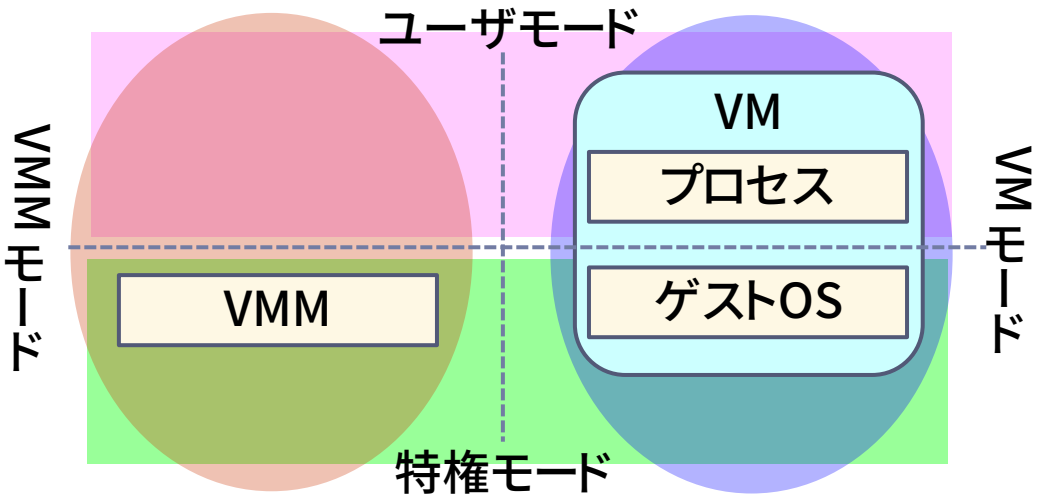
これをソフトウェアで行う手法として、ソフトウェアによる「シャドーページテーブル」があります。

仮想化でのI/Oの管理

- I/Oデバイスを仮想化してゲストOSに仮想デバイスとして見せる
  - 性能への影響が大きい
  - デバイスが多様
- 仮想ディスク … ゲストOSにホストOS上の一般のファイルをマウントする
- デバイス単位で割り当て（パススルー）
  デバイスを共有せず、特定のゲストOSが占有する
  ハードウェアとVMの対応付けはユーザの設定に従ってVMMが行う
- デバイスエミュレーション
  特定の形式のデバイスをVMMでエミュレートし、そのインタフェースをゲストOSに提供する
  （I/O処理は特権命令であるため、VMMに制御が移る）
  - 実デバイスからの割込みはデバイスエミュレータで処理し、適切なVMに送る
- DMAの場合、ゲストOSのデバイスドライバがゲスト実メモリのアドレスを設定すると、そのアドレスがVMに割り当てられているとは限らないなどの問題を起こすことがある

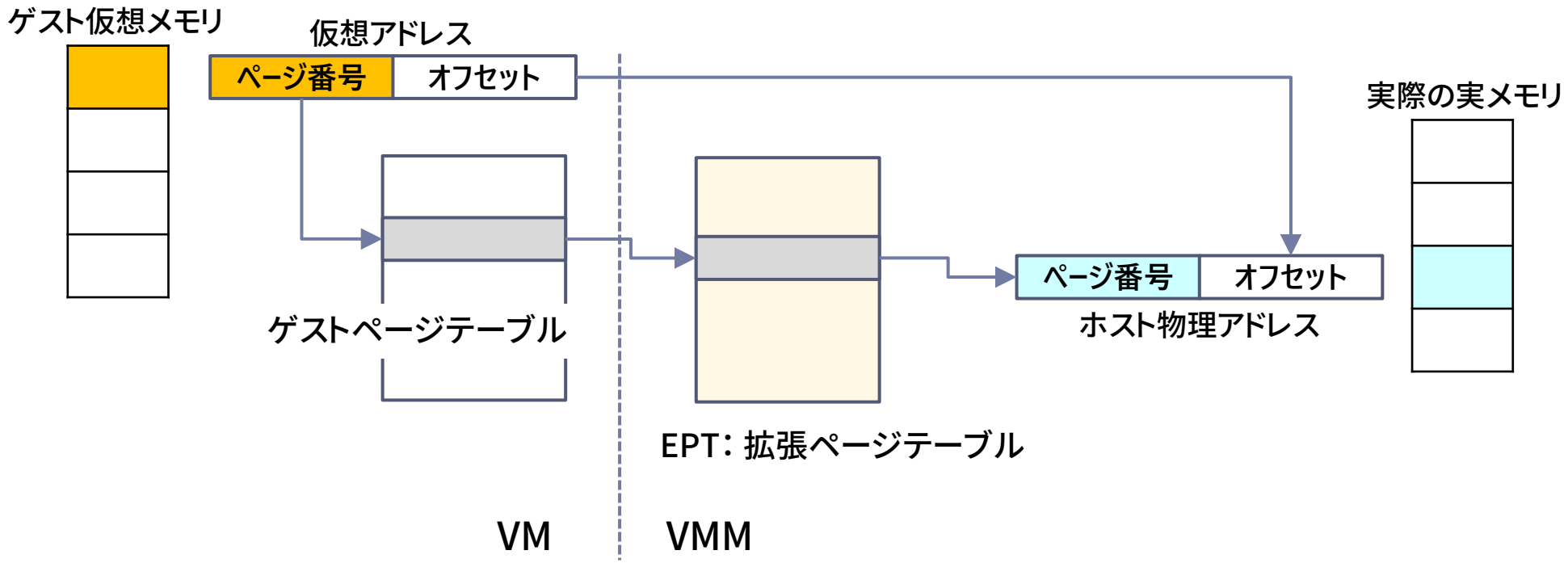
仮想化に対するハードウェア支援

- VM用の新たな実行モード
  - 特権モード/ユーザモードに直交するCPUの動作モード：VMMモード（ホストモード）/VMモード（ゲストモード）



  - プロセスはユーザ・VMモードで動作し、システムコールを実行するなどすると、特権・VMモードになる
  - ゲストOSが特権命令を実行すると、VMMへ遷移して 特権・VMMモードになって実行する
    終了するとゲストOSに復帰（VMM移行/復帰命令発行）
    - VMMは適切な処理を行って適切なゲストOSに戻すことができる

- ネステッドページングによる仮想アドレス変換
  - ゲストページテーブルにより仮想アドレスからゲスト物理アドレスに変換され、その後 VMM が管理する EPT（拡張ページテーブル）によってホスト物理アドレスに変換される
  - ページテーブルによるアドレス変換が2回必要
    シャドウページテーブル方式に比べオーバーヘッドははるかに小さい

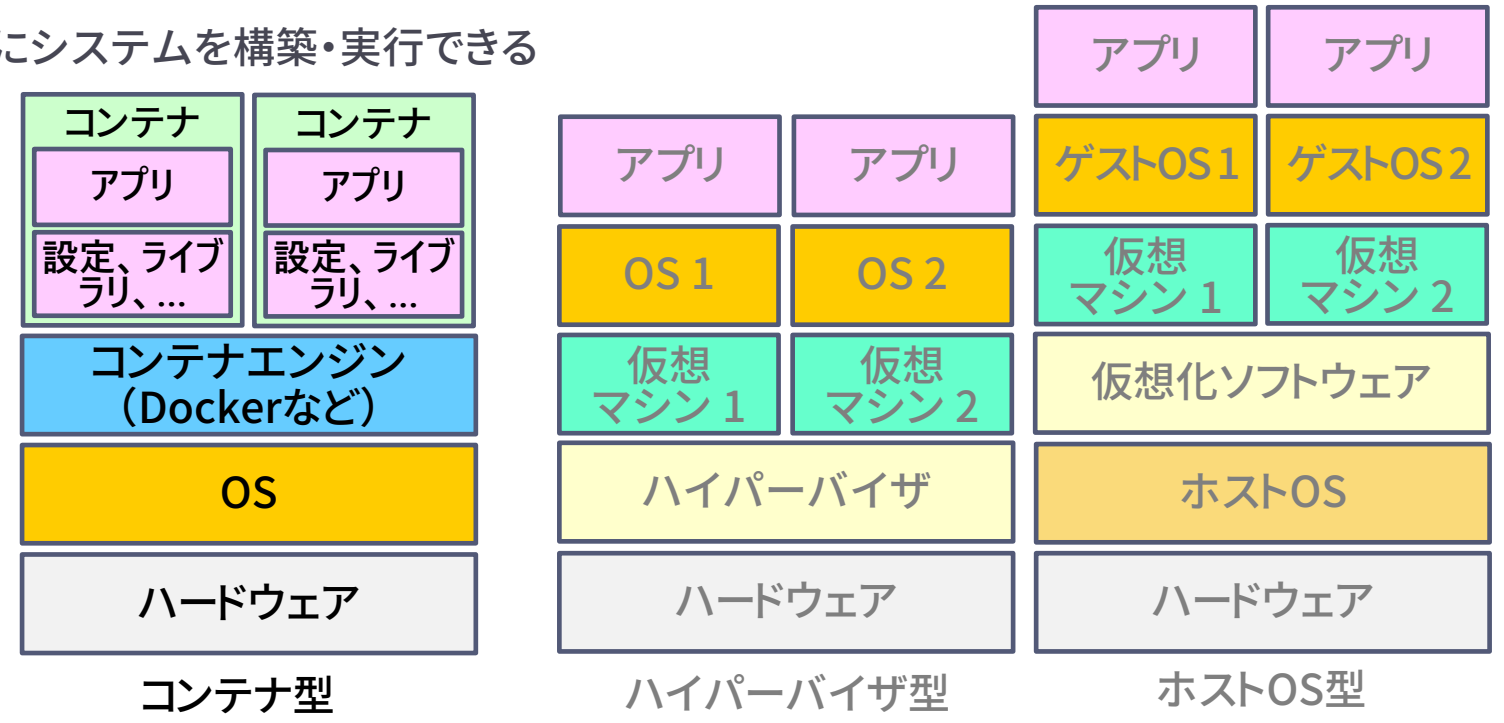


補足・参考：コンテナ型仮想化

- 仮想化技術には、ハイパーバイザ型とホストOS型に加えて、コンテナ型がある
  - 1つのOS上に複数のコンテナ（他のユーザから分離・独立した実行環境）を構築し、
    その中でアプリケーションを動作させる仕組み

    コンテナには、アプリケーションの実行に必要な設定ファイルやライブラリ、実行時ソフトウェアなどが格納され、
    それぞれのプロセスが独立して実行・管理される
    （ファイルシステムやOSリソースの隔離、プロセスのリソースや権限の制限、など）

    他のアプリケーションとの競合を意識せずにシステムを構築・実行できる
  - コンテナはOSを内包しないので、
    軽量で高速、
    リソースを効率的に利用



教科書との対応

- プロセッサ仮想化でのスケジューリングやセンシティブ命令のエミュレーション、メモリ仮想化での準仮想化ページング、バルーニング、入出力仮想化での準仮想化ドライバなどについて割愛しています
- コンテナ型の仮想化について追加しています
- OSの構成法について追加し「参考書_第2.7節」を参考資料とします

この資料では、

「2.7.1 単純構造」でモノリシックカーネル方式について説明し、

「2.7.2 層アプローチ」で階層化（スライド資料p.3）を説明しています

そして、「2.7.3 マイクロカーネル」で、カーネルから必須ではない構成要素をシステムやユーザレベルのプロセスとして階層化して実装したものとして説明しています

（教科書では、第4章のコラム p.51 でマイクロカーネルが説明されています）

なお、資料内にある「2.4.5項」は、この参考書でプロセス間通信について説明されている部分であり、講義資料では第9回の p.20 以降が対応します

教科書での説明と記載箇所が違うところがありますので、読むときに注意してください。

- 今回の課題はありません

事後学習

- 今回の講義資料に基づいて内容を振り返り、教科書などの該当箇所を読む

今回の講義内容の振り返りをお願いします。