オペレーティングシステム
（2024年 第３回）

OSとハードウェアアーキテクチャについて

前回の課題について

問1 は、OSのAPI（システムコール）の講義資料の説明（ソフトウェアからOSの機能を利用する）の通り （イ）

- （ア）は「ハードウェアを直接操作して」が当てはまらない
  （ウ）については、APIの意味としてアプリケーションプログラム間でデータを受け渡すときの決め事を含むが、本設問では「OSのAPI」と限定されているので、（イ）の方がより適切な説明になる

問2

- ある時点であるプログラム（A)に提供していたCPUを次の時刻で別のプログラム（B)に提供するためには、Aを中断してBに切り替える操作が必要になる
- その際、さらに別の時刻で、中断されていたAを、その中断時の状態から再開できるように実行の状況を保存する処置が必要になる

OSが前提とするハードウェア支援機構

- 割込み・例外とその処理
- 実行モード
- アドレス変換

WindowsやUNIX系のOSではこれらの機能がCPUに備わっていることを前提とする

- 組込み系のOSではこのような機構を前提としないものもある
  - その場合、実現されないOS機能がある
- ハードウェア機構も進歩し続けている
  - 仮想マシンの支援など

ソフトウェアで実装できるものはできる限りソフトウェアで実現するというのが基本的な考え方

CPUに備わっている割込み・例外、実行モード、アドレス変換と呼ばれる機構についてお話しします。

我々が一般によく使っているOS、WindowsやLinuxではこれらの機構がCPUに備わっていることが前提となっています。
このような機能がないと、仮想化や効率的な資源管理が実現できません。

今あげた3つの機構だけでなく、新しい機構、これは例えば仮想マシンを支援するためのハードウェア機構などですが、追加されたりしています。

ソフトウェアで実装できるものは、極端な性能劣化を引き起こさない限りソフトウェアで実装する。ハードウェアでないと実現できない基本機能をハードウェアで実現するというのが基本的な考え方です。

背景 (1)

- 入出力装置のプログラミングの問題点
- OSが存在しない場合
  - 昔 … プログラムから直接入出力命令を発行し、CPUと入出力装置との間でデータを転送
    入出力の実行が終了するまでCPUが停止している
  - 入力データの到着/出力データの転送完了を表すフラグを用意し、プログラムでそれを読んで確認する
    (「busy wait」と呼ぶ)

    ```
    while (入力データのフラグが未着)
        入力を行う
    ```

  - 問題点
    - CPUの利用効率が悪い(入出力が完了するまで他の処理が行えない)
    - 複数の装置から同時に入出力を行う場合、操作が煩雑になる

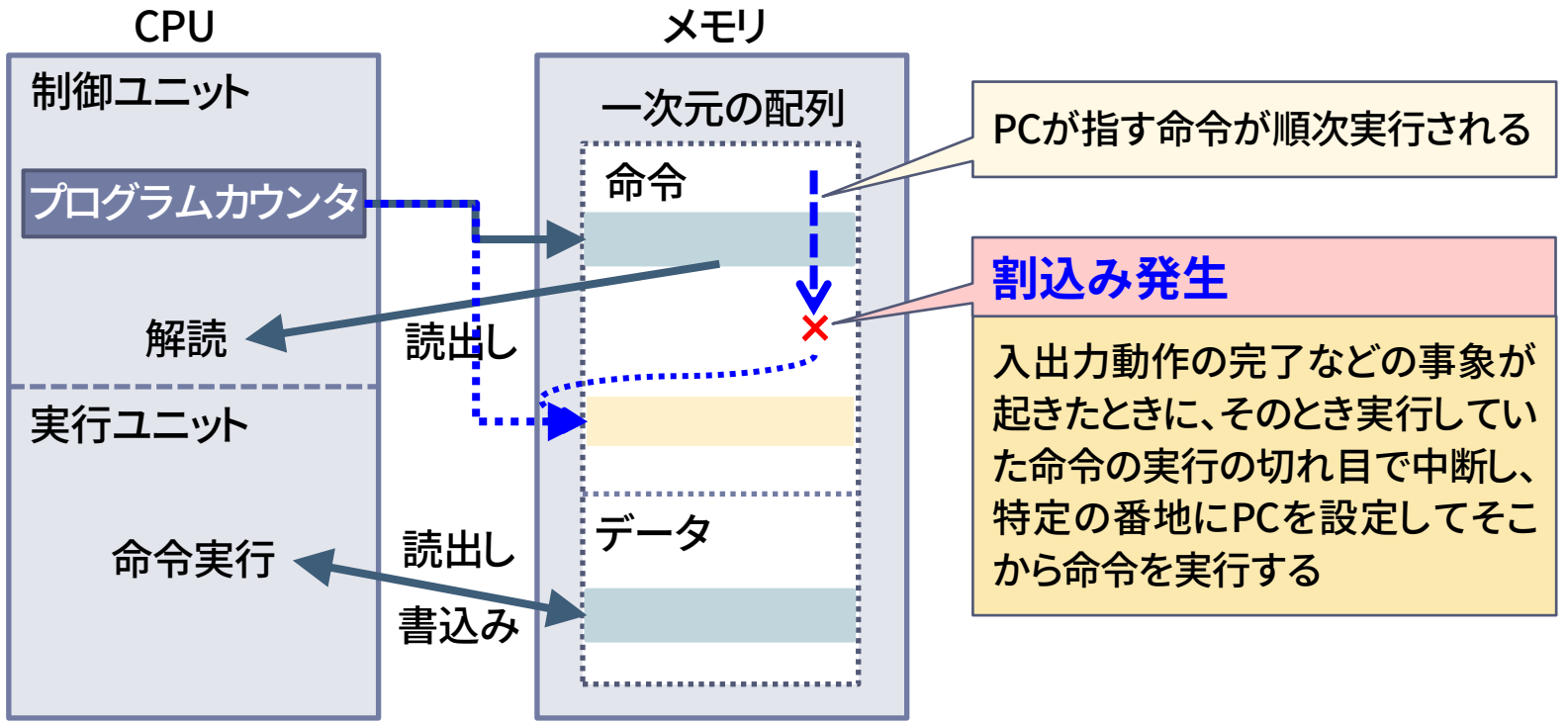
このようなハードウェアによる支援機構が導入された背景から考えます。OSを発展させるようハードウェアも進化したということになります。
まずは、入出力装置のプログラミングを行う上での問題点です。

OSの助けを受けられない場合を考えます。
OSがまだない昔では、プログラムの中から、直接入出力を行う命令を発行して、それによってCPUと入出力装置との間でデータを転送します。
ここで、そのデータ転送が終了するまで命令の実行が完了せず、CPUは停止した状態になります。

現在でも、OSが搭載されないコンピュータでは、入力の場合、 CPUが停止することはありませんが、データの到着を示す変数であるフラグ(名前のとおり旗で示すわけです)を用意して、それが設定されるまで確認することを繰り返します。
このように、CPUで繰り返しによって処理の完了を検査する方式を「busy wait」と呼びます。プログラムで書くと、フラグが設定されているかを見て、未だならそれをずっと繰り返すということが典型的なものです。
このbusy waitによる処理は単純で分かりやすいのですが入出力処理の完了をCPUによるプログラム実行で検査するので、完了するまで他の処理が行えず、CPUの利用効率が悪いことが問題です。
また、複数の入出力装置から同時にデータの入出力を行う場合、個々の装置からそれぞれが完了したことを検査することになり、操作が複雑になります。

割込みの動作

- 割込みの目的
  - プログラムが自ら入出力処理の完了を検査しなくてよいよう
  - 入出力処理の完了をbusy waitで行わなくてよいよう



このような方式の問題は、プログラムで自ら入出力処理の完了をbusy waitで行うことにあります。
これをなんとかしようと、割込み機能が発明されました。

CPUの構成で説明したように、プログラムは、プログラムカウンタの指す命令を読み出して実行し、プログラムカウンタを進めて次の命令を実行するという繰り返しで進行します。
この実行はプログラム内に命令が並べられて指定され、順序付けられたとおりに行われるものです。

割込みは、この命令実行を、順序付けを無視して特定の場所に強制的に移動させる、すなわち、プログラムカウンタ(PC)をそこに対する値に設定するものです。
ここにあるように、実行中の命令の切れ目で、その命令を中断して、特定の番地(どこかは、後で説明します)にPCを設定する、すなわちそこに制御を移すことになります。
割込みによって、現在実行していた命令系列を中断するということが可能になります。

OSを支援するためのハードウェア機能（1）

- 割込み（interrupt）
  CPUの命令実行順序の系列を突発的に変更するもの
  ハードウェア割込みとソフトウェア割込みに分けられる
- ハードウェア割込み（外部割込み）
  実行中のプログラムとは関係しない非同期な外部事象の発生によるもの
  - 入出力割込み … 以前に起動した入出力操作が完了した
  - タイマ割込み … セットした時間が経過した
  ・・・ など
  - CPUに外部から与えられる電気信号
  - 主な目的 … 周辺機器からの情報をCPUが他の作業をしながら
    取落とすことなく受取る
  - 効果 … CPU資源の有効利用
    - 周辺機器の側から割込みによって処理の終了を通知
      周辺機器が処理を行っている間、CPUが他の処理を行ったほうが効率がよい

割込みにはいろいろな種類があり、大きくハードウェア割込みとソフトウェア割込みに分けられます。
ハードウェア割込みは、入出力操作が完了したことを入出力装置が知らせる入出力割込み、設定した時間が経過したことを知らせるタイマ割込みなどで、
CPUにその外部から信号としてあたえられるものです。外部割込みとも呼ばれます。

前のページにあったように、CPUがプログラムを順次実行している中で、その実行とは無関係に、入出力装置やタイマといった周辺機器からの信号によって現時点の実行を中断させて、特定の場所に行く、すなわち、たとえば入力の処理を行うようにできるのです。

この効果は、入出力の完了をCPUがbusy waitでフラグを見るなどして検査しなくてよくなり、入出力を行っている間CPUは他の処理を行うことができるということで、CPUの有効利用です。

- ソフトウェア割込み（内部割込み）
  - CPU内部において、実行した命令や命令実行に関わるモジュール（例えばキャッシュ）の変化によって発生する
    例外 (exception)やトラップ (trap)と呼ばれることもある
    - カーネル呼出し割込み … CPUの割込み命令を実行
    - 演算例外割込み … 0除算、オーバーフロー、などの発生
    - アドレス変換例外割込み … アクセスしたページがメモリの中にない
    - …

    プログラム割込み
- 割込みは、プログラムの制御により、その原因となる事象が発生しても、割込みの動作を抑止（マスク）することができる
  割込み禁止状態
- 割込み処理中に割込みが発生することもある（多重割込み）
- 割込みには優先度がある（割込みレベル）

もう一つ、ソフトウェア割込みがあります。これは、CPUの内部で実行した命令によって変化する状態が引き起こすもので、内部割込みとも呼ばれます。
例外やトラップと呼ばれることもあります。

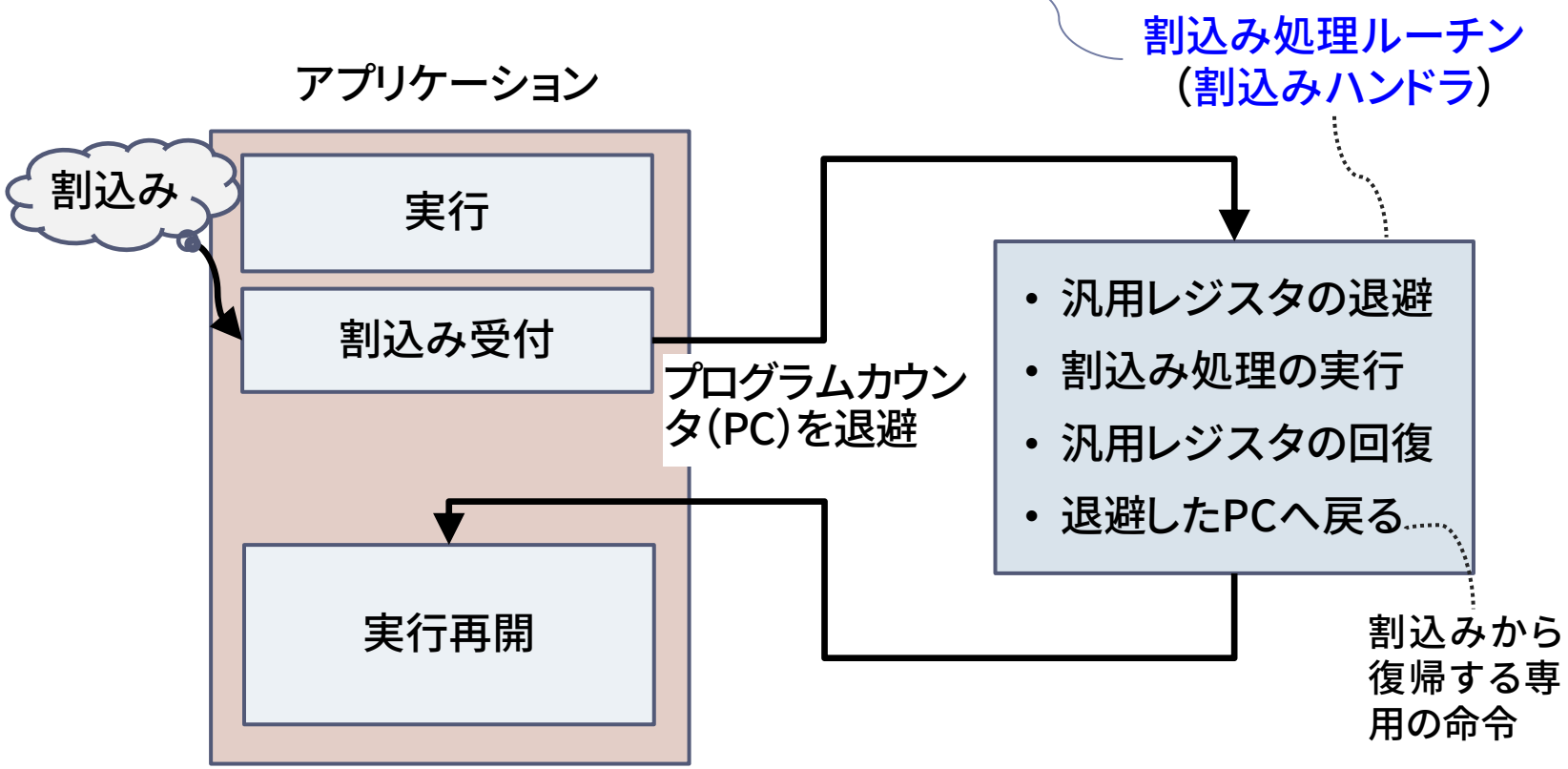
前回説明したカーネル呼出しを実現する特別な割込み命令、カーネル呼出し割込みや0で除算したとき、演算結果が表現範囲を超えてしまったときなど演算結果により発生する演算例外割込み、メモリにアクセスしたときに異常を示すアドレス変換例外などがあります。

これらは、プログラムの実行に伴って発生するので、「プログラム割込み」とも呼ばれます。

なお、割込みは、発生してもその動作を抑止することができます。ただし、電源異常を知らせる割込みのように抑止できない割込みもあります。

割込み処理・例外処理

- 割込みや例外が発生したときの処理
  - 割込みを受付けるとCPUは特定の番地に制御を移行する
    その番地から始まるように割込み用の処理を置いておく



それでは、割込みがどのように処理されるかについて見てみます。

割込みが発生してそれが受け付けられると、CPUはその時点で実行している命令の切れ目で、実行を中断し、PCがあらかじめ決められた番地に設定されます。
このPC設定の直前に、その時点のPCの値、これは「中断した命令の次」－すなわち、割込みがなければ実行するはずの命令のものを退避します。「退避」というのは、メモリの中に領域をとっておいて、そこに保存するということです。

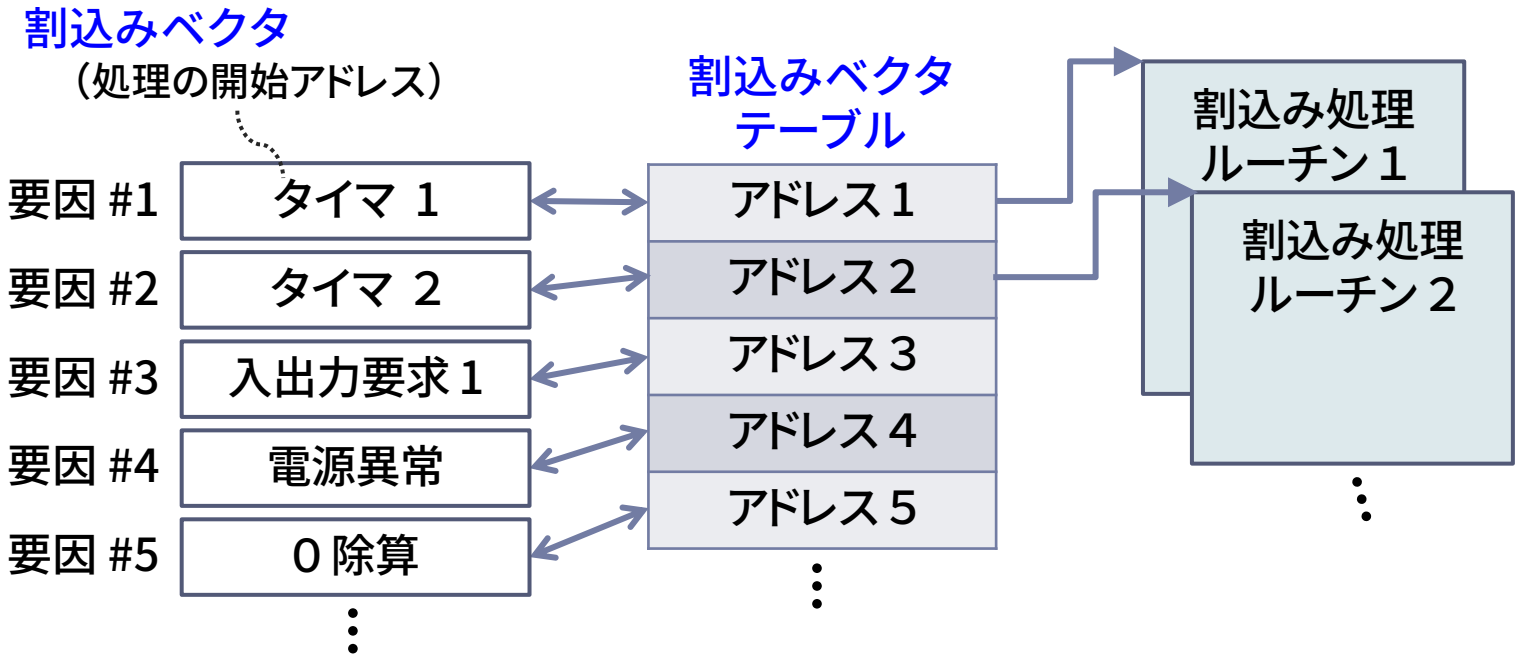
新たにPCが設定されたので、CPUはその番地から実行を続けることになります。そこに、割込みに応じた処理のプログラムを作成しておくのです。
この図の右側の箱、「割込みハンドラ」や「割込み処理ルーチン」と呼ばれるものがそれです。

割込みハンドラでは、まずCPUの汎用レジスタなどを退避した後、割込みに対応した処理(たとえば入出力装置とのデータ転送)を実行します。その後、先に退避した汎用レジスタを回復します。この退避・回復は、割込み処理の中で汎用レジスタなどを変更してしまうので復帰するために行います。
そして以前に退避しておいたPCの値を回復してそこに戻ります。

そうすると、実行が中断された次から再実行されることになり、汎用レジスタなどの値も中断時点のものにもどっていますので、本来実行されるはずであった命令が実行されることになります。

割込み処理ルーチン

- 割込み処理ルーチン(割込みハンドラ)の配置場所
  - 割込みを受付けるとCPUは特定の番地(アドレス)に制御を移行する
  - 割込みの事象(割込み要因)は複数ある
  - 割込み要因と割込み処理ルーチンの対応づけ
    割込みハンドラの開始番地(ベクタアドレス)を配列でもつ



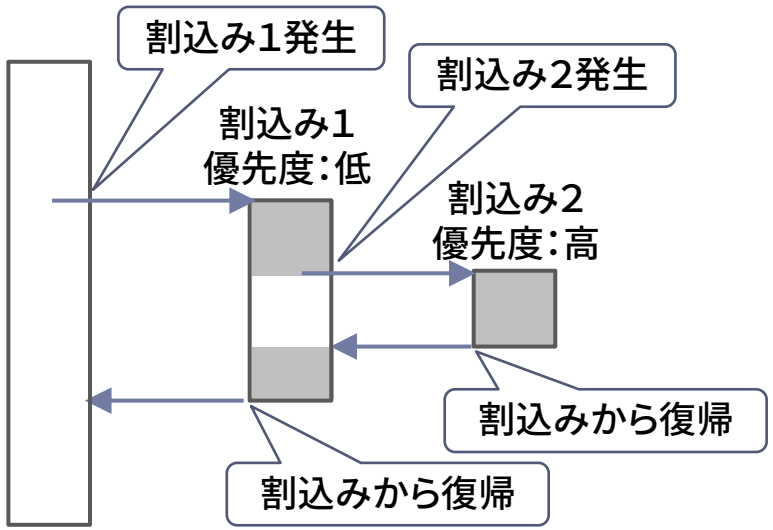
割込みの種類(割込み要因)ごとに割込み処理ルーチンを用意することで、それぞれに対応した処理が実施できます。
割込みによって特定の番地に飛ぶようになっているので、その番地が割込み要因ごとに設定できれば、実現できます。

要因に対する飛び先(割込みハンドラの開始点)のアドレスを「割込みベクタ」とよび、このアドレスを順番にメモリ上に配置したものを「割込みベクタテーブル」と呼びます。

割込みベクタテーブルがメモリの特定のアドレスから始まっていて、割込み要因を示す番号に従って、2番目の要因であれば、第2要素の中にあるアドレス2をPCに設定するような機構になっています。

多重割込み

- 同時に二つ以上の要因の割込みが発生した場合
  - 割込み優先度は同時に割込みが発生した時の処理の順序を定める
  - 同一レベル（優先度）の割込み要求が同時に発生した場合
    … 例えば番号の小さい割込み要因が優先となる
- 一つの要因の割込み処理中に別の要因の割込みが発生した場合
  - より高いレベルの高い割込みが発生した場合には、その処理が優先される
    その際、処理中である低レベルのものは待たされる
  - 低いレベルの割込みが発生した場合は、より高いレベルの割込み処理が終了するまで延期される



割込みは複数ありますので、複数のものがほぼ同時に発生することがあり、割込み処理はそれに対応できなければなりません。

複数の割込みが同時に発生する場合を考えます。
割込みには優先順位（割込みレベル）が付けられていて、優先度の高いものが処理されるようにします。同じ優先順位のものが同時に発生した場合は、番号順などがとられます。

また、1つの割込み処理中に別の割込みが発生することもあります。このときも優先順位によって処理し、より高いレベルの割込みが発生すると、現在処理中のものを中断して優先順位の高いものの処理を行います。

まとめ

- 割込みの利点
  - CPUの効率的利用
  - 入出力処理と計算処理の分離
- 割込み処理用の命令
  - 割込み処理ルーチンからの復帰命令、割込みの許可・不許可 など
- ソフトウェア割込みの利用
  - 仮想化での利用
  - バグの発見
  - 信頼性向上

割込みについてまとめますと、
割込みの利点は、割込みが来るまで別の仕事をしていることができ、もともと目的としていたようにCPUの利用を効率化できるということです。
割込み処理に関する専用の命令として、CPUには割込みから復帰する命令などが用意されています。

ソフトウェア割込みの利用法としては、仮想化に利用するものがあり、これについては今後、詳細を説明します。
ここでは、たとえばアクセスしたメモリが存在しないという状態をソフトウェア割込みで検出すると、メモリの仮想化が行なえるというような程度をイメージしてください。

背景 (2)

- バグによるプログラム破壊
- 保護と実行モード
  - 正当な利用者、プログラムに対して適切なアクセス権限を与える
  - 2つの区分 … 実行時のモード
    - 資源管理を行い、ハードウェアを操作する特権的なプログラム
    - 資源管理や入出力処理は行わず、計算などの処理を行うプログラム
      - 入出力命令の実行を禁止、メモリの参照領域を限定、などの機能制限

次の背景として、バグからプログラムを保護したいということがあります。
バグとは、プログラムにおける不備とか欠陥とかいった意味ですが、その結果、アクセスする場所を間違っているとか正しい命令でない内容を命令としてしまったなどの状況に陥ります。その結果、間違った計算になるとか、動かない、暴走するといったことが起こり得ます。

バグがどこにあるかを発見することは困難です。
先に説明した割込みや例外などを使うと、バグによる不都合な状態を検出することができる場合があります。しかし、割込み処理ルーチンが破壊されていたりすると、割込みによる通知を受け取れなかったり、どこまで暴走しているかわからないといった可能性があります。

このような状況のうちのある部分をハードウェアの機構によって回避できるようにしたいというものが、この背景です。
プログラムに対して、メモリをアクセスする権限や命令を実行できる権限といったものを与えると回避できるのではということです。
単純には、2つのモードがあって、一つは特権的で、すべてのアクセスと命令実行ができるもの、もう一つはアクセスできないメモリの領域があり、また実行できない命令があるものです。(p.14の特権モードとユーザモードを見てください)

OSを支援するためのハードウェア機能（2）

- 実行モード（CPUモード）
  CPUが実行できる操作を制限する動作モードを設ける
  保護が目的（カーネルの機能はユーザモードでは実行できない）
  - 特権モード（カーネルモード、スーパバイザモード）
    無制限のCPU動作を許し、全ての操作が可能
  - ユーザモード（非特権モード）
    CPUの動作に制限が加えられ、一部の命令の実行や入出力操作ができない、メモリ空間の一部にアクセスできない
  - アプリケーションはユーザモードで実行され、例えば、直接ディスクにアクセスする権限はない。ディスクのアクセスは必ずカーネルを通して操作を行うようにする
- 実行モードの実現
  - 実行モードはCPUのレジスタに格納される
  - 実行モードや演算結果の状態を格納したレジスタをプログラム状態語（Program Status Word：PSW）と呼ぶ

第1回で説明したプログラム内蔵方式のコンピュータアーキテクチャでは元来、保護は考慮されていませんでした。すべてのプログラムはすべての命令を実行でき、すべてのメモリを読み書きでき、すべての入出力装置を操作できました。
しかし、すべてのプログラムがそのような機能をもつことは、バグによるプログラムの破壊からの保護という観点からは適切ではありません。

プログラム、すなわちそれを動作させているCPUが実行できる命令やアクセスできるメモリを制限できれば、ということが考えられ、それが実行モードと呼ばれるものです。
カーネルの機能＝資源管理を行ってハードウェアを操作する特権的なもの、とユーザ（アプリケーション）の処理＝資源管理や入出力は行わず、計算などを行う、の最小限2つに区分するモードがとれるようにします。

特権モードではすべてのCPU動作、メモリ操作が行えますが、ユーザモードでは、CPUの動作に制限が加えられ、一部の命令の実行や入出力操作ができない、メモリ空間の一部にアクセスできないといった制限があります。

実行モードはCPU内のレジスタに格納されます。

実行モードと例外処理

- 3つ以上の実行モードを持つものもある
  - 複数のユーザモードをサポートし、階層を形成する
    → リング保護
  - ハイパーバイザモード用のモード
- ユーザモードにおいて、実行できない命令やアクセスできないメモリを操作すると、例外が発生する
- 例外処理の実行時にOS（カーネル）が実行される
  - 割込みやシステムコール命令の実行などで、実行モードが特権モードになり、OSが実行される

モードを3つ以上もつCPUもあります。x86では4つのモードをもち、教科書にあるように内側から権限の高いレベルの同心円の階層をなしていて、「リングプロテクション」と呼ばれます。
今日のWindowsやLinuxなどでは、特権モードとユーザモード2つのレベルしか使っていません。
1台のコンピュータ上で複数のOSを同時に実行するための機構であるハイパーバイザ用のモードを使用することがあります。

実行モードと例外処理の観点では、ユーザモードにおいて、実行できない命令やアクセスできないメモリを操作すると、例外が発生します。
割込みや例外は必ず特権モードで実行されます。割込みや例外処理の実行時にOS（カーネル）が実行されるということになります。

システムコール用命令の実行、割込み発生、エラー検出など例外処理、これらすべてで、p.9のようにアプリケーションの実行が一時中断され、実行モードが特権モードになり、OSが実行されます。

背景 (3)

- メモリへの要求
  - メモリは常に不足している
    有効利用することがOSの課題
- プログラムとメモリ
  実行のためには、プログラムもデータもメモリ内にあることが必要
  - 複数プログラムの同時実行にはより大きなメモリの提供が必要
  - 保護の必要性
    プログラムで使用できる領域から離れた領域の破壊からの保護 など
- 主記憶装置（メモリ）のアドレス空間

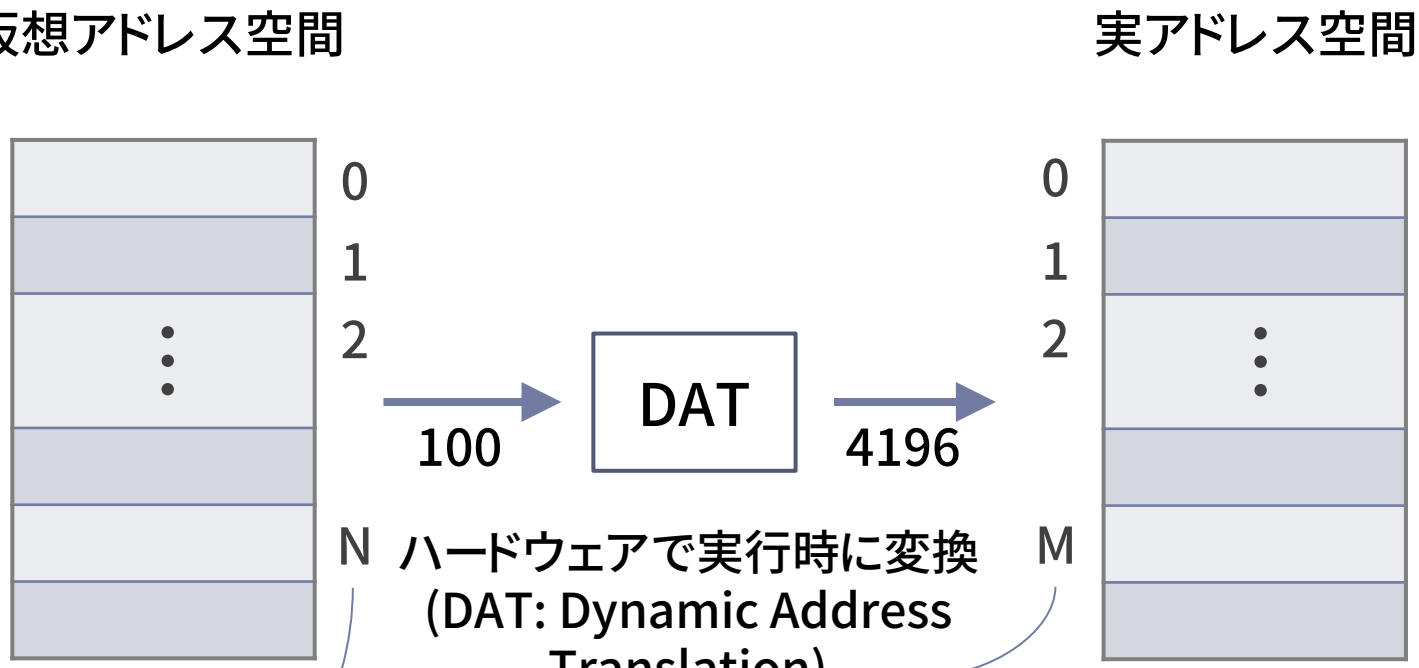
  アドレス指定（番地づけ）によってアクセス可能であるメモリの範囲のこと
  - メモリのアドレス空間を実アドレス空間と仮想アドレス空間に分ける
  - 仮想アドレス空間は、実行するプログラムごとに個別に定義する
  - アドレス変換によって、仮想アドレスを実アドレスに変換する

次に、メモリに関してです。
プログラムの実行では、命令語もデータも一緒にメモリに保持していることが前提（プログラム内蔵方式）ですので、メモリの大きさで解ける問題のサイズが決まることになります。
現在のような半導体メモリが発展する以前では、メモリは高価な装置であり、大量に実装することは困難でした。
現在でも、グラフィカルなユーザインタフェースや動画、音声の処理などはメモリに対する要求が高く、メモリは常に不足しているといえます。
複数プログラムを同時に実行させるには、必要となるメモリは増加します。

容量への要求だけでなく、バグなどによりメモリの内容が破壊された場合、不具合として現れる症状は多様で原因を発見することが困難であり、このようなメモリに対する破壊を検出して対処することも必要です。

このように、資源の有効活用と保護の観点からメモリに対するハードウェアでの支援が考えられてきました。実際のしくみについては、「メモリ管理」で説明しますが、ここでは、以下を概要として見ておきます。
まずアドレス空間ということを考えます。
メモリの空間を実アドレス空間（実際に装備されているメモリのアドレスによる空間）と仮想アドレス空間（実行するプログラムが使用する論理的なアドレスによる空間）に分けて考えます。
仮想アドレス空間は、実行するプログラムごとに個別、独立に定義されます。（たとえば、どのプログラムも論理的なアドレス0番地から開始するといったようなことです）

ハードウェアによるアドレス変換



プログラムの実行時に、仮想アドレスから実アドレス（物理アドレス）へのアドレス変換が、ハードウェアで行われます。
アドレス変換を行う機構は、動的アドレス変換機構：DAT（Dynamic Address Translation）と呼ばれ、CPUの中にあるメモリ管理ユニット：MMU（Memory Management Unit）で行われます。
この図では、プログラムの実行時に、プログラムでは100番地にあるものとする変数を参照するのに、CPUが「100」を出力すると、それがDATで「4196」と変換されて実際のメモリの4196番地を指定するようになることを示しています。

このような変換を行うことで、仮想アドレス空間では同じアドレス（複数のプログラムのものなど）が異なる実アドレスに対応できるようになります。

教科書との対応

- 割込み、実行モードについては、教科書4.1節
- アドレス変換機構については、教科書10.3節

教科書ではハードウェア機構との関連が分かれて記載されていますので、読むときの参考にしてください。

第３回の課題

ある割込み処理中にほかの割込みを許す場合、その処理には何を注意するべきか

今回の課題です。クラスウェブのレポートで提出してください。

4/27の午前中までに提出してください。

事後学習・事前学習

- 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
- 以下の項目を主に、教科書7章（7.1～7.3）、8章（8.2）に目を通す
  - プロセス
  - スケジューリング

今回の講義内容の振り返りと次回の準備をお願いします。