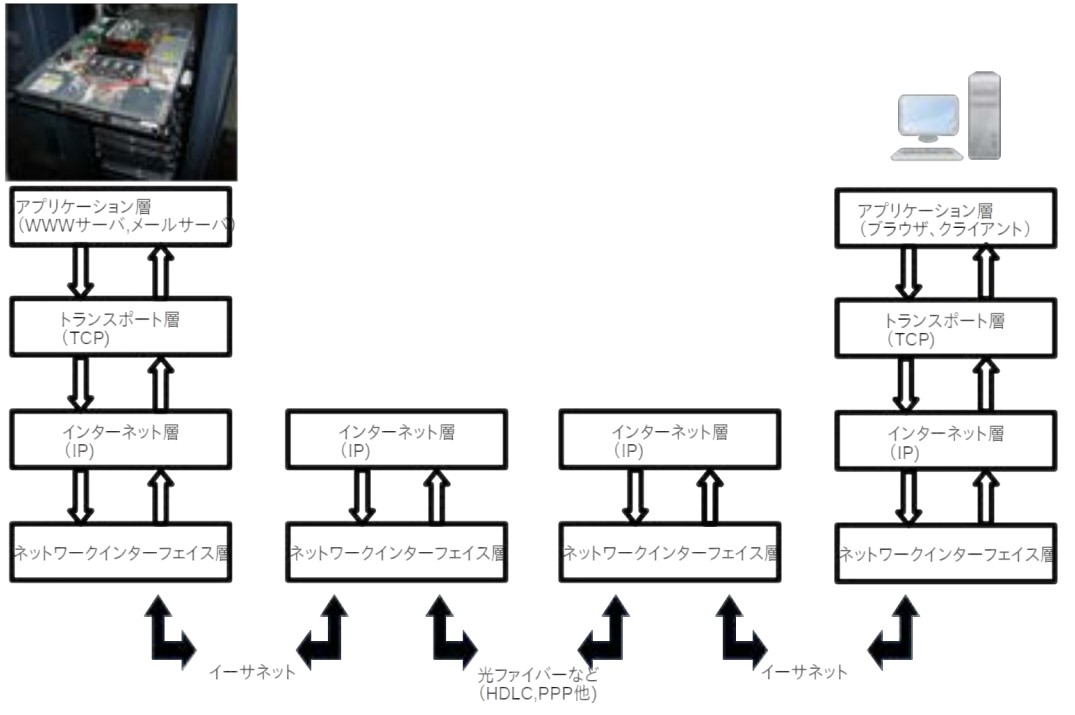
情報技術応用特論 第04回
OS(1): OSの概要

WWWサーバを例にOSの動作イメージを解説する

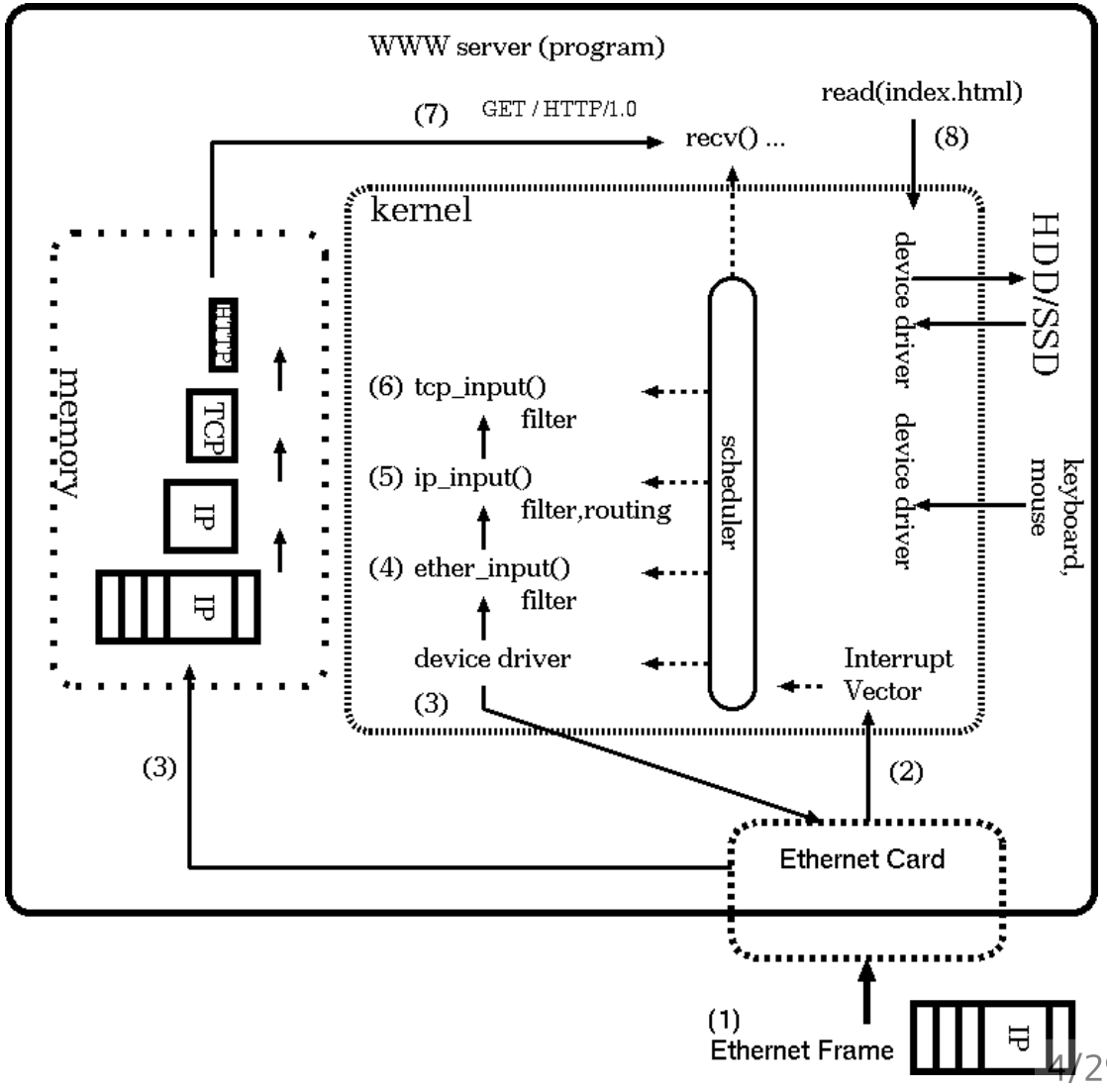
ネットワークの全体像(復習、networkのスライド再掲)

- 右図: PC(サーバ)とPC間の通信
  - END TO END (端〜端)通信
  - 用語: 階層(Layer), 〜層
  - 用語: プロトコル(取り決め,約束事)
    - TCP,IP,Ethernetなど
  - 用語: サーバ,クライアント
- 右図: サーバ(左)とクライアント(右)
  - サーバ (図左上の業務用PC)
    - サービスを提供する側
    - WWWサーバ,メールサーバ
  - クライアント (右端のPC)
    - サービスを受ける側(お客様)
    - ブラウザやメールソフト
    - 人間が操作しているPC



サーバの全体像とキーワード

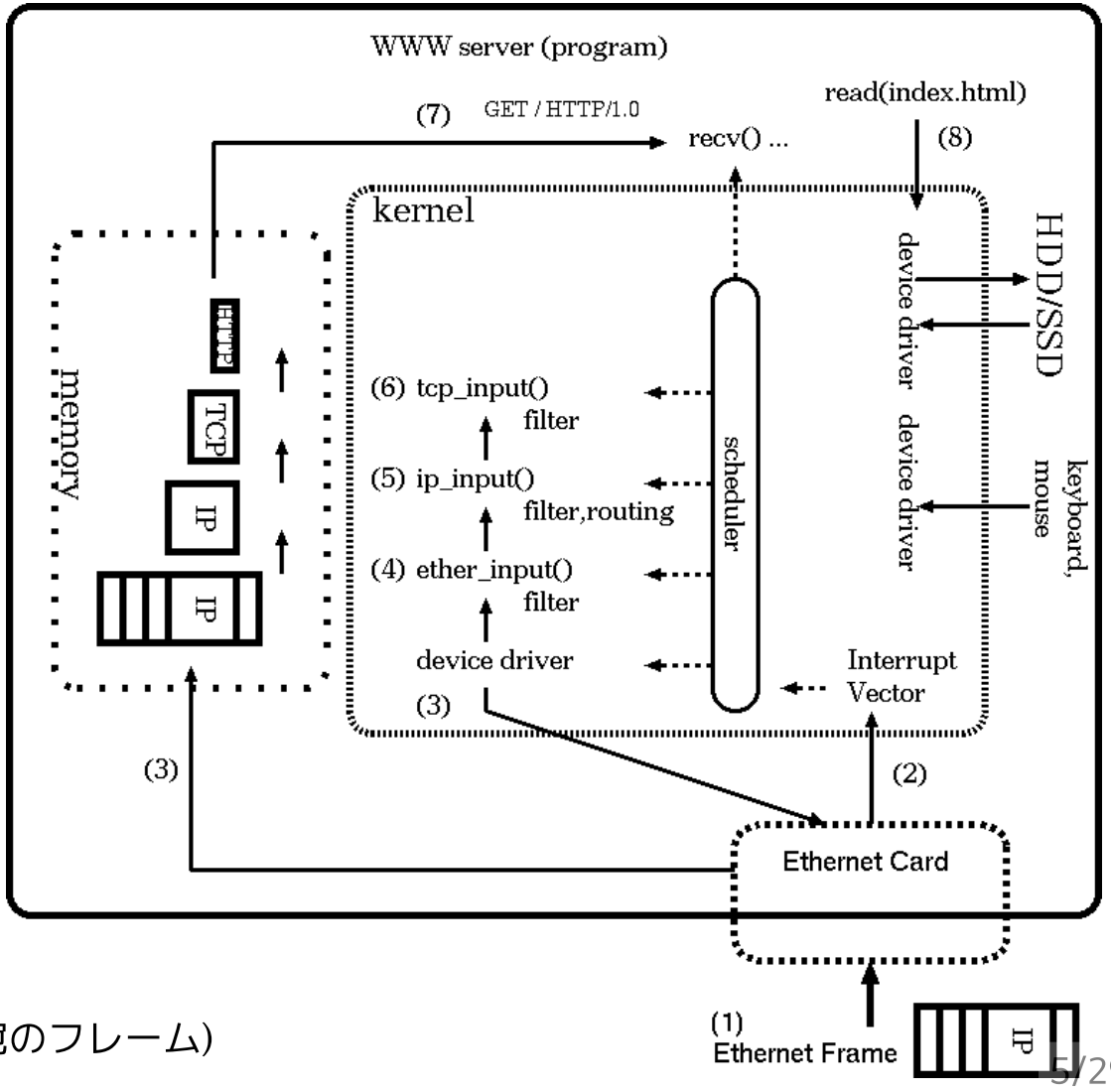
- 前頁のサーバ部分(図左端)を題材にして**OSの重要キーワード**を紹介します。 次頁から各ステップを簡単に見ていきます
- 右図はWWWサーバの内部動作です。すでにTCPの3wayハンドシェイクが終わり、 WWWブラウザからリクエスト（GET / HTTP/1.0）のパケットが送られてきた時点を想定してください
- 用語
  - カーネル(kernel) ... OS本体と言うべき**特権**プログラム



(脚注) WWWサーバプログラムの例: apacheやnginx、thttpd

サーバの動作(1)

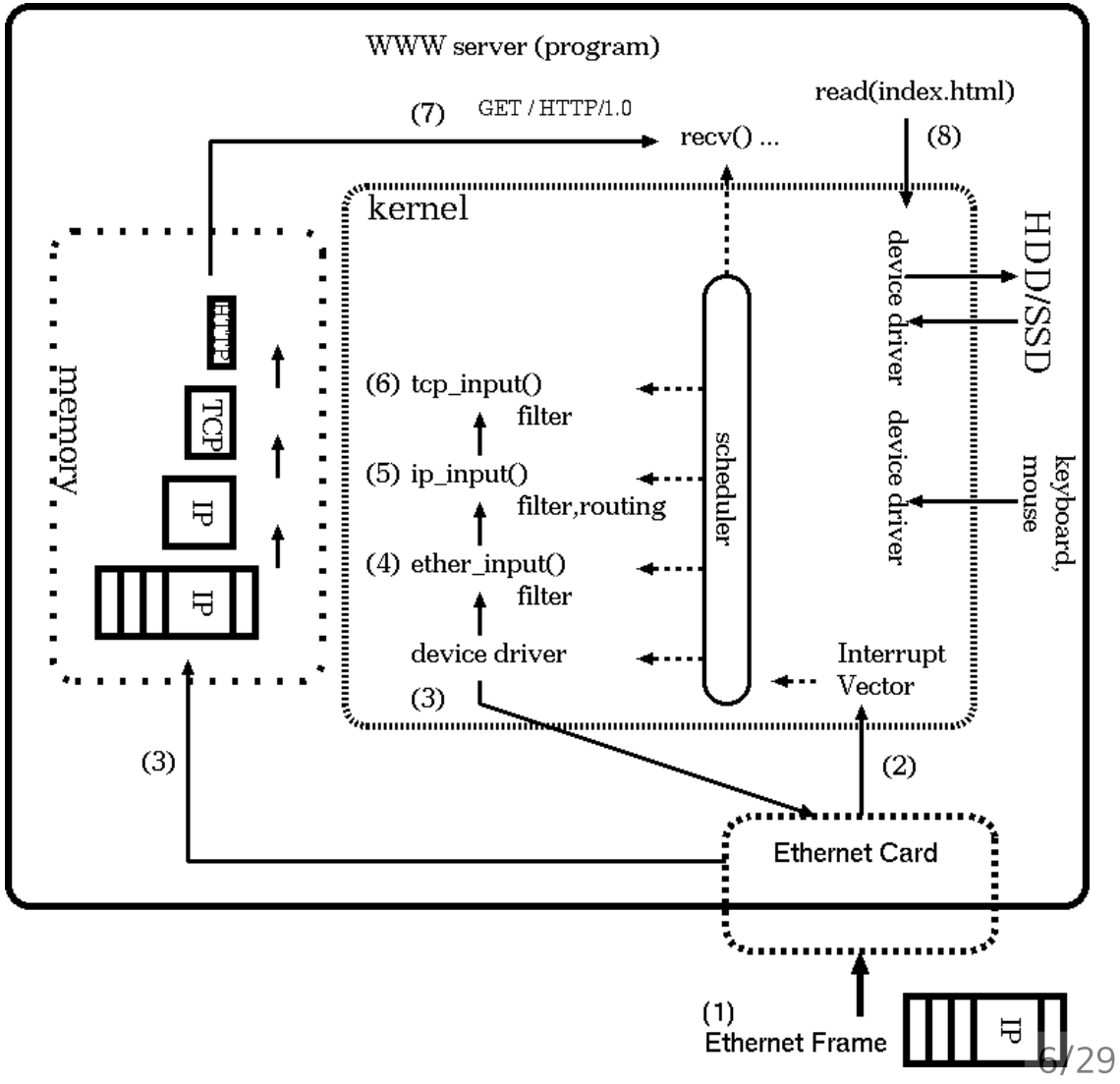
- サーバ宛のイーサネットフレームをネットワークインターフェイス（**イーサネット**）カードが受け取ります
- フレームヘッダの宛先MACアドレスがカードのMACアドレスと一致すれば自分宛ということなので、 カードはフレームを受け取ります。 宛先が異なれば無視します



(脚注) もちろんブロードキャストも受け取ります(FF:FF:FF:FF:FF:FF宛のフレーム)

サーバの動作(2)

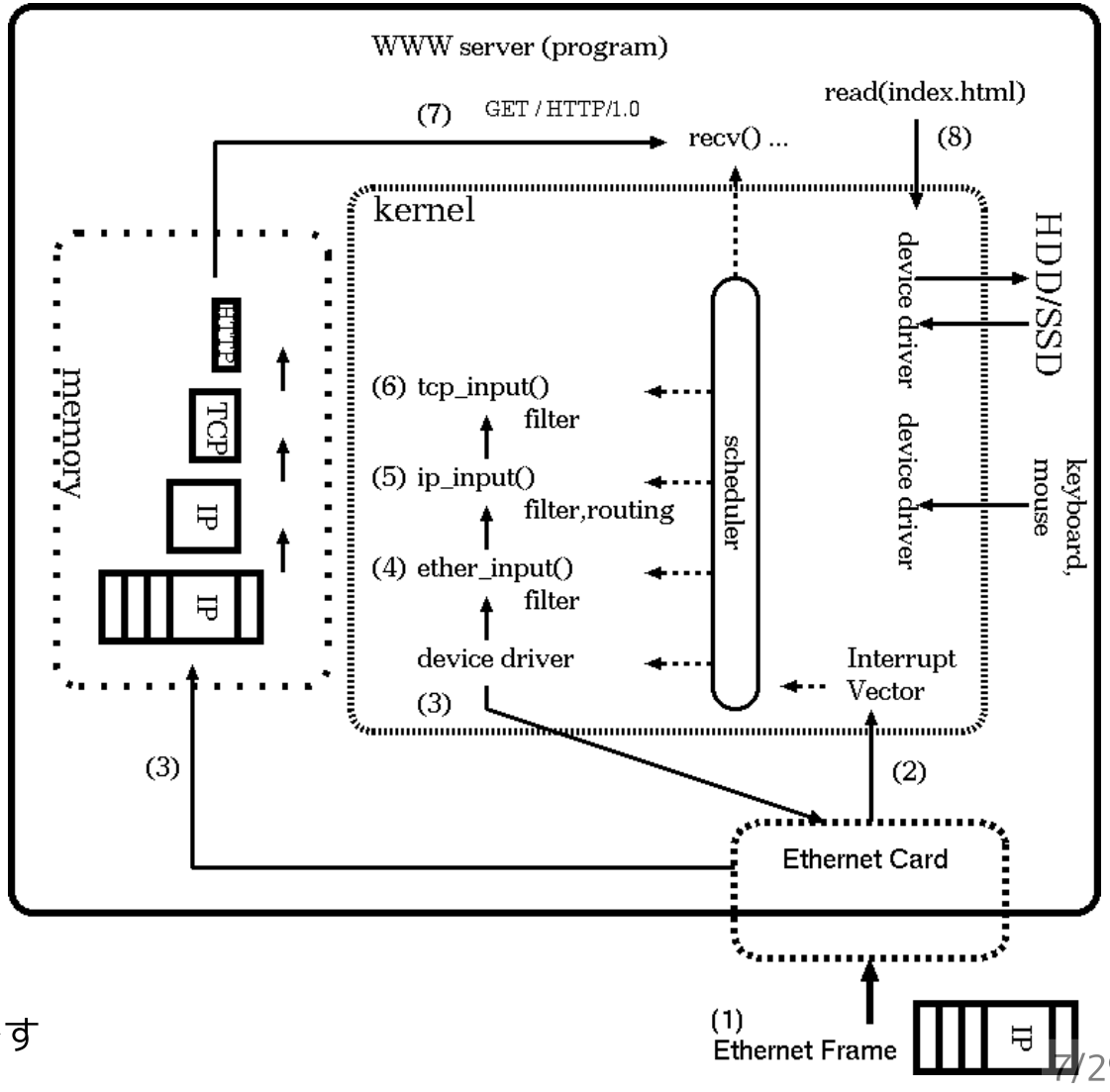
- イーサネットカードのコントローラが、**ハードウエア割り込み**をかけます
- OSが事前に設定した**割り込み**の設定表(**割り込みベクタ(interrupt vector)**)を検索し、 該当する処理関数が呼ばれます
- この**ハードウエア固有の**処理関数を**デバイスドライバ**と呼びます
  - ハードウエアそれぞれに専用の**デバイスドライバ**が必要です
  - カーネルに該当するデバイスドライバがない場合、 そのハードウエアは利用できません



(脚注)各デバイスにあるコントローラも小さなコンピュータです

サーバの動作(3)

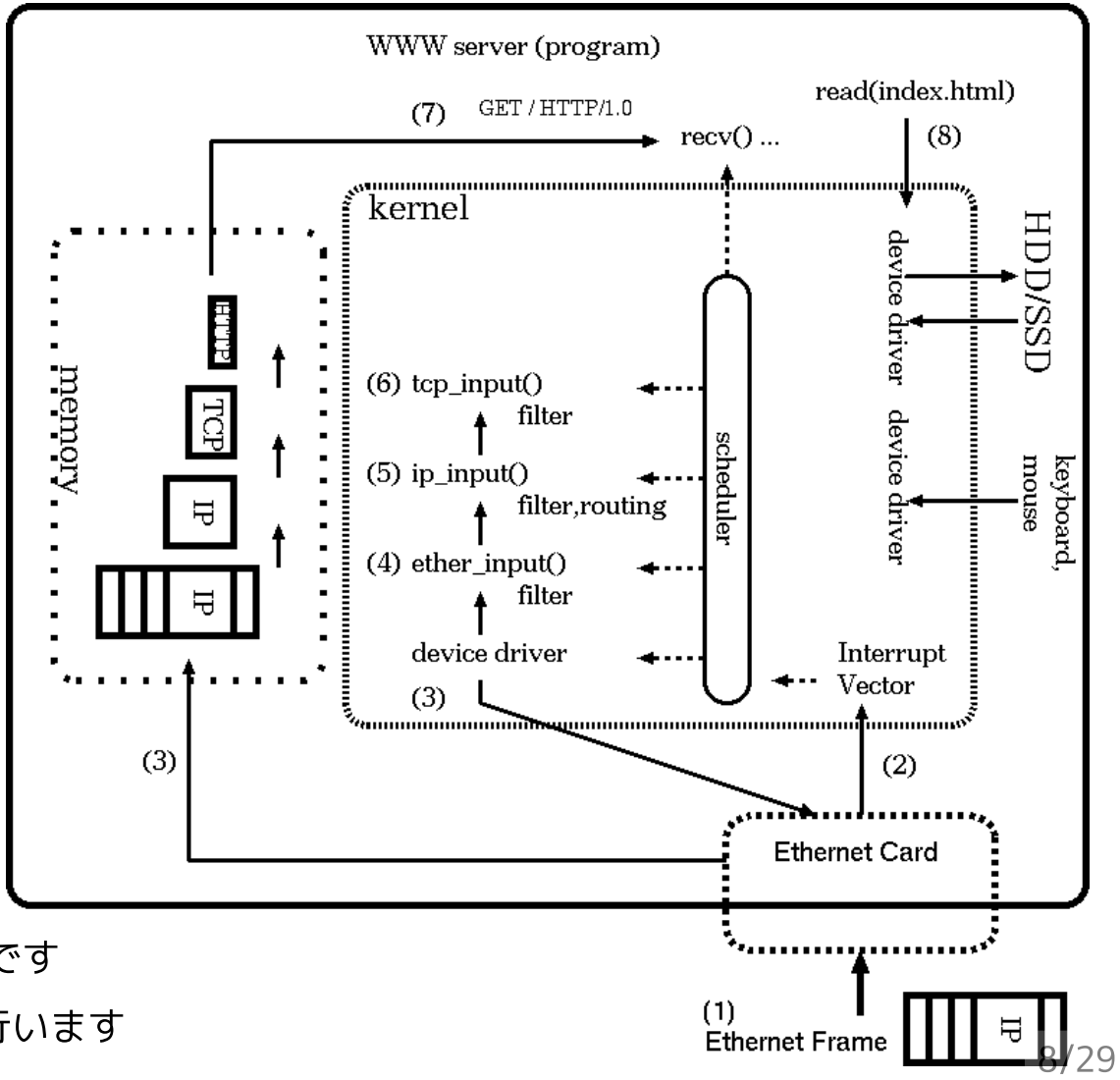
- 呼ばれたデバイスドライバは図のイーサネットカードの扱い方を知っています
  - 例: メモリ上の0170000から読む（後述）
- カードからイーサネットの情報を取り出し(メイン)**メモリ**にコピーします
- デバイスドライバは、この先の処理をイーサネットの処理関数 ether_input() にまかせます



(脚注)ここで出てくるXXX_input()はBSD Unixにある実際の関数名です

サーバの動作(4)

- ether_input() は、 イーサネットフレームのヘッダを見て、 フレームの中身がIPパケットだと分かるので、 IPパケットの処理関数 ip_input() を呼び出し、 先の処理をゆだねます

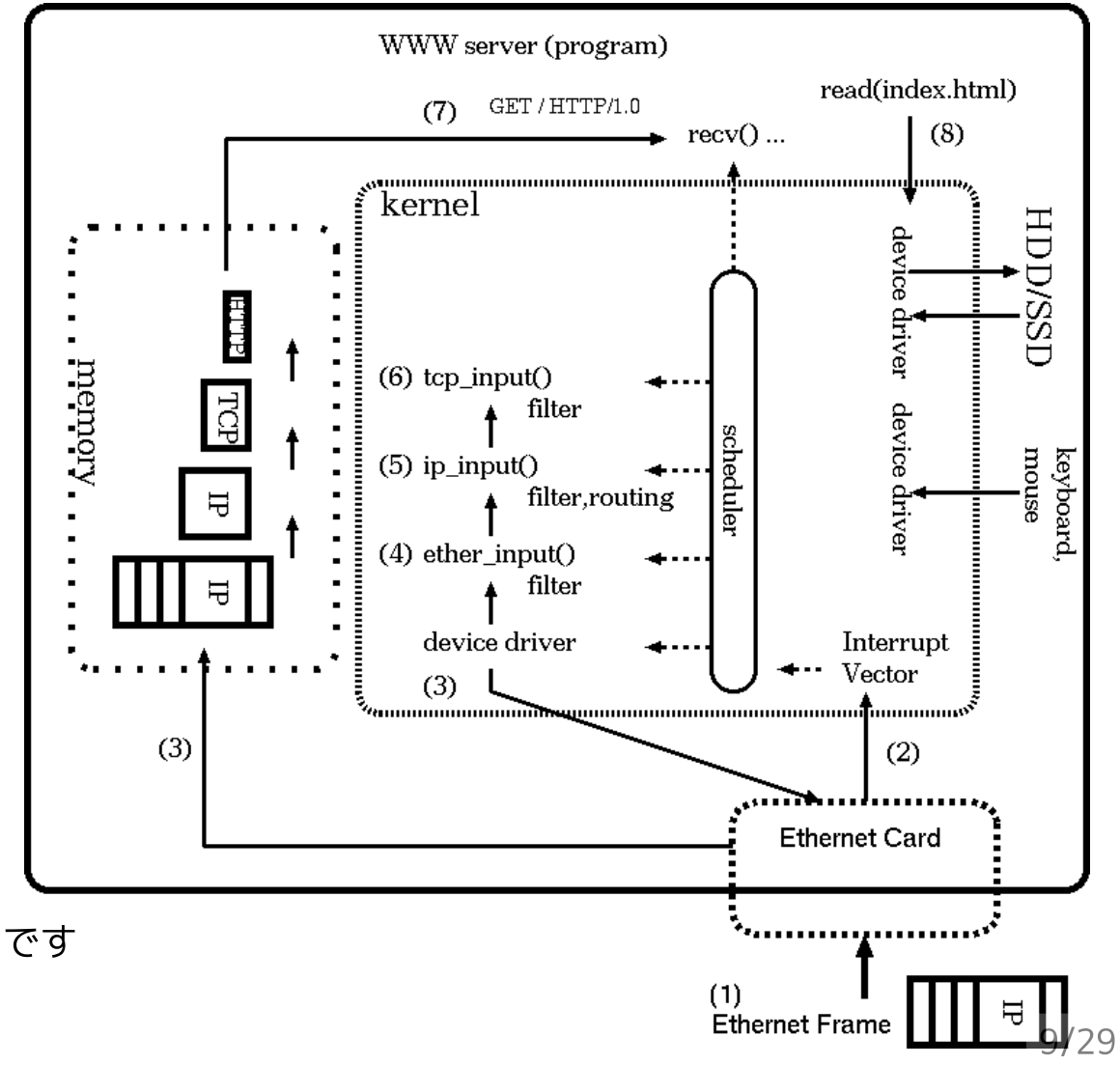


(脚注1) ここで出てくるXXX_input()はBSD Unixにある実際の関数名です

(脚注2) ether_input() は、必要があれば、フィルタリングの処理も行います

サーバの動作(5)

- ip_input()はIPパケットの処理を行います
- IPヘッダを見れば、ペイロード(運んでいるデータ)がTCPだと分かるので、 tcp_input()を呼び出します

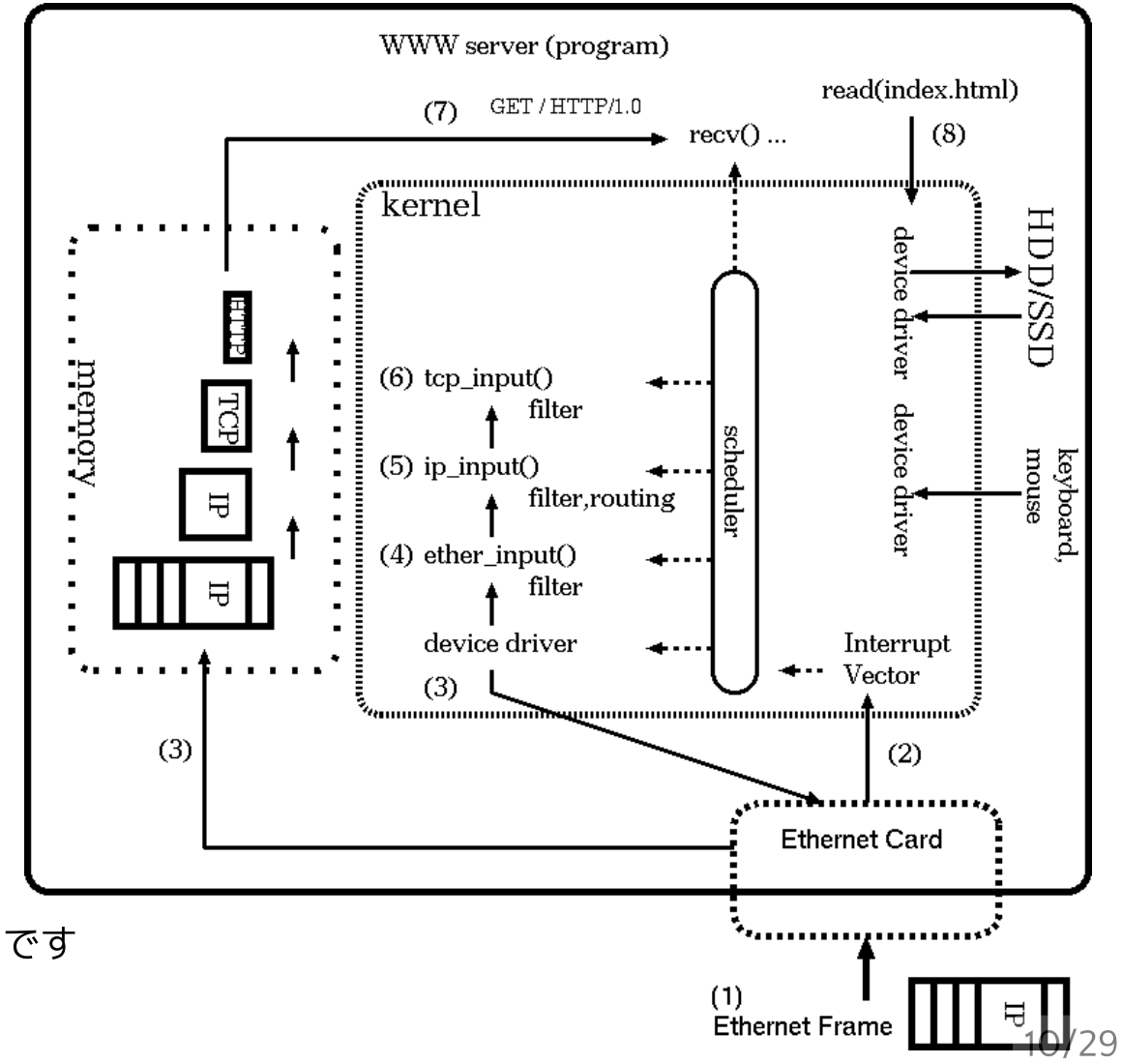


(脚注1) ここで出てくるXXX_input()はBSD Unixにある実際の関数名です

(脚注2) 適宜ルーティングやフィルタリングの処理も行います

サーバの動作(6)

- tcp_input()はTCPパケットを処理します
- 最終的にパケットのペイロード(データ)が取り出され、 WWWサーバ側でデータを待っている関数を呼び出します。 例：recv(2)

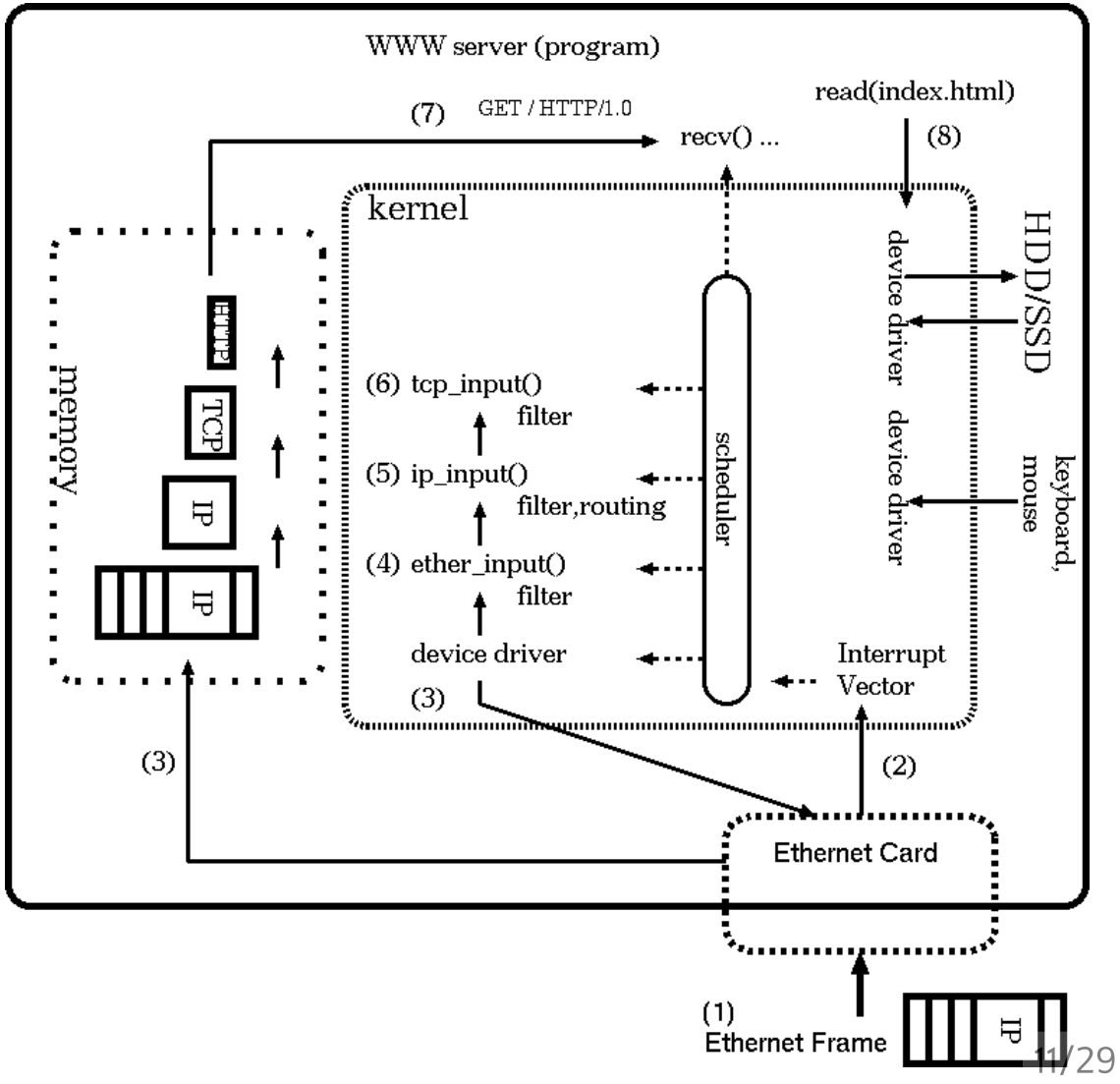


(脚注1) ここで出てくるXXX_input()はBSD Unixにある実際の関数名です

(脚注2) 必要であればフィルタリングの処理も行います

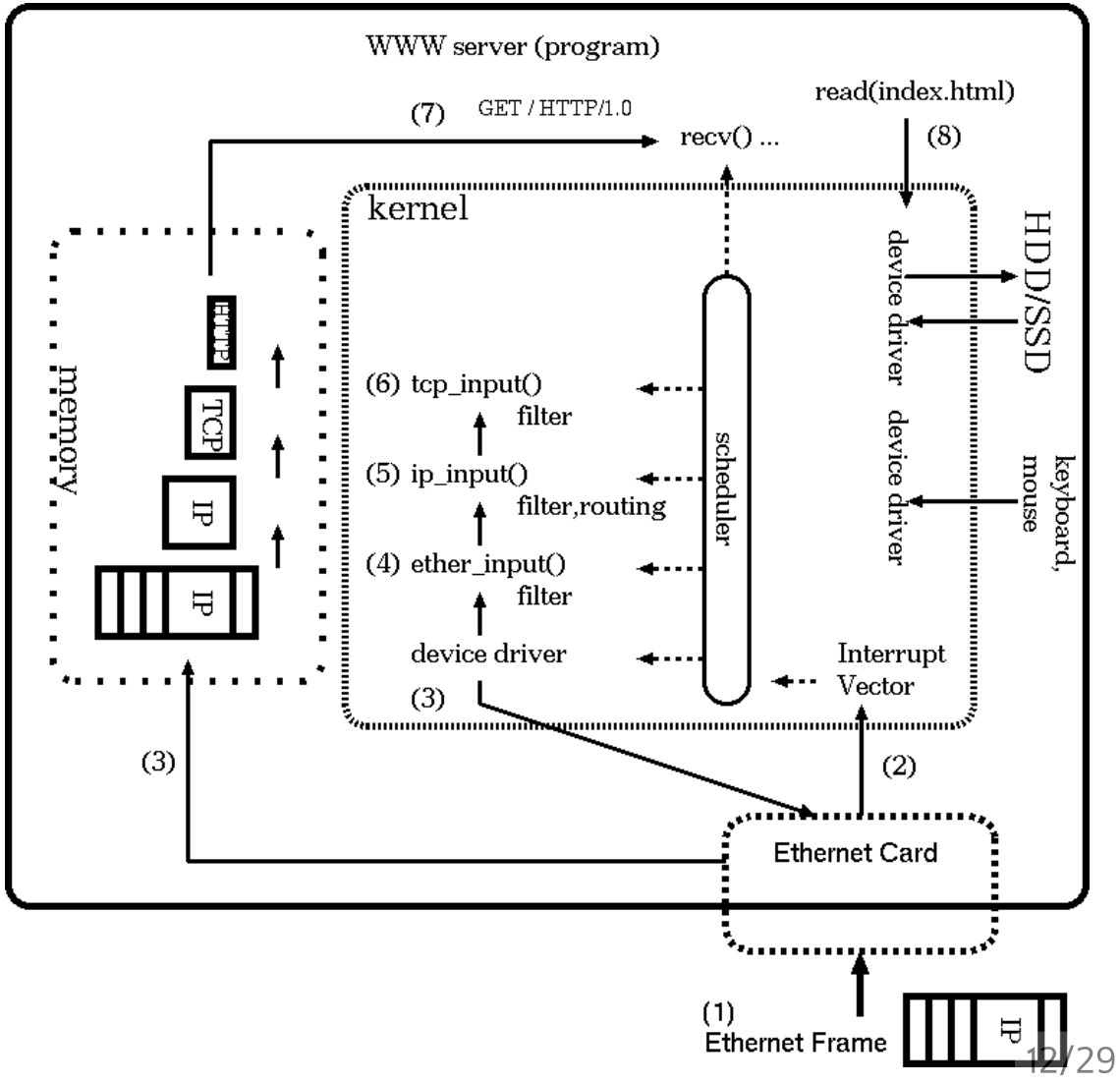
サーバの動作(7)

- WWWサーバの**プロセス**は、 クライアントからのデータを待つ関数(例:recv())により、 メモリからデータ`GET / HTTP/1.0`をよみこみます
- このあとWWWサーバ内の処理が続きます
  - 上のGET …をparseすれば `/` つまりサーバのトップページ(index.html)をリクエストしていることが分かります



サーバの動作(8)

- クライアントからのリクエスト「`GET / HTTP/1.0`」を解析して、 index.htmlファイルを読み込む必要があると分かりました
- （図では省略されていますが） (1) **ファイルシステム**上のindex.htmlファイルをopen**システムコール**で開き (2) read**システムコール**でファイルを読む、これらの処理をカーネルに依頼します
- HDDやSSDからファイルを読み出すのは、 そのHDD/SSDハードウエアの**デバイスドライバ**です。 カーネルが適切な**デバイスドライバ**を呼び出します
- HDDやSSDは**ストレージ**と呼ばれます

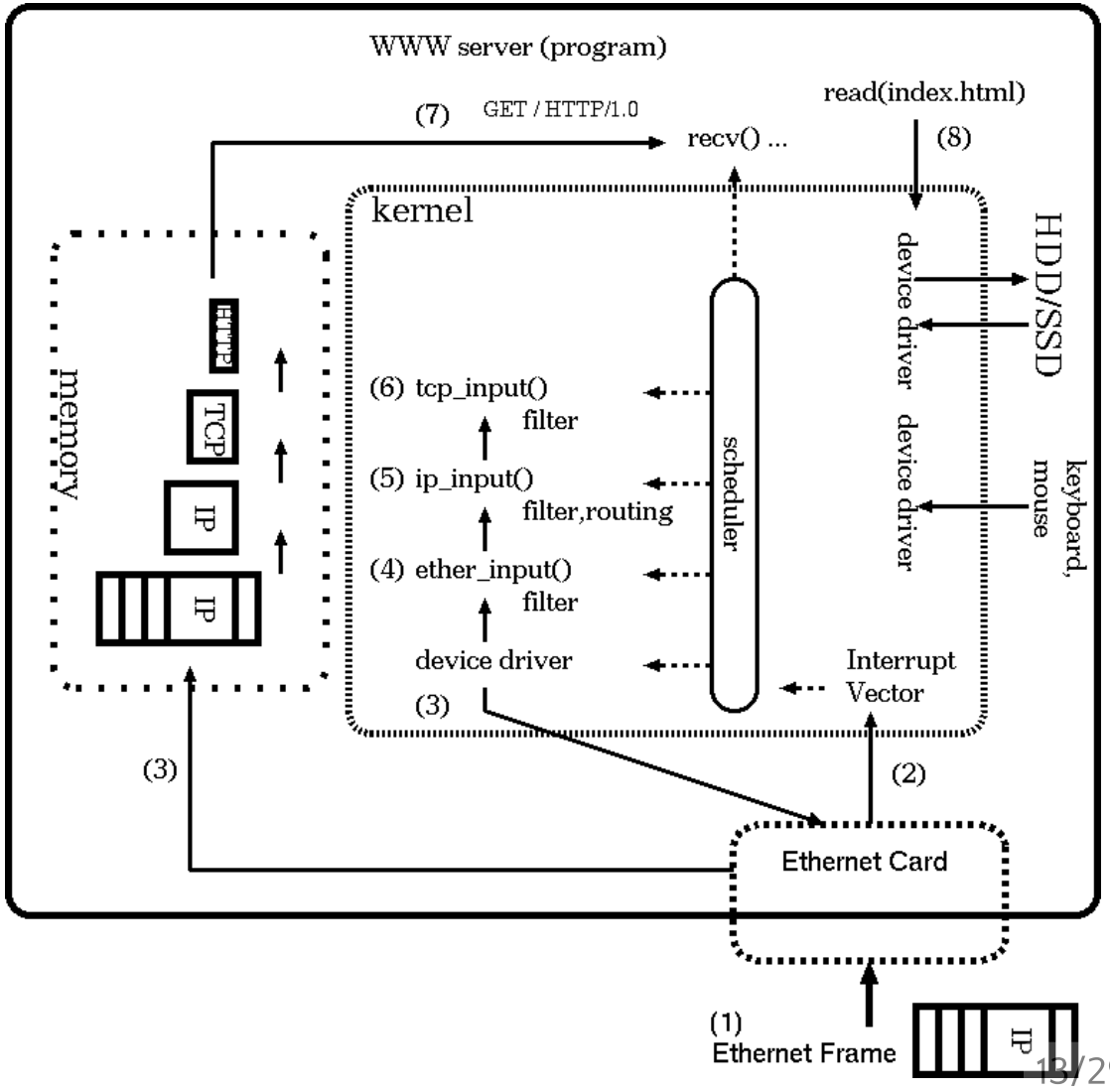


(脚注1) システムコールも割り込みの一種です

(脚注2) システムコールは**ソフトウエア割り込み**と呼ばれています

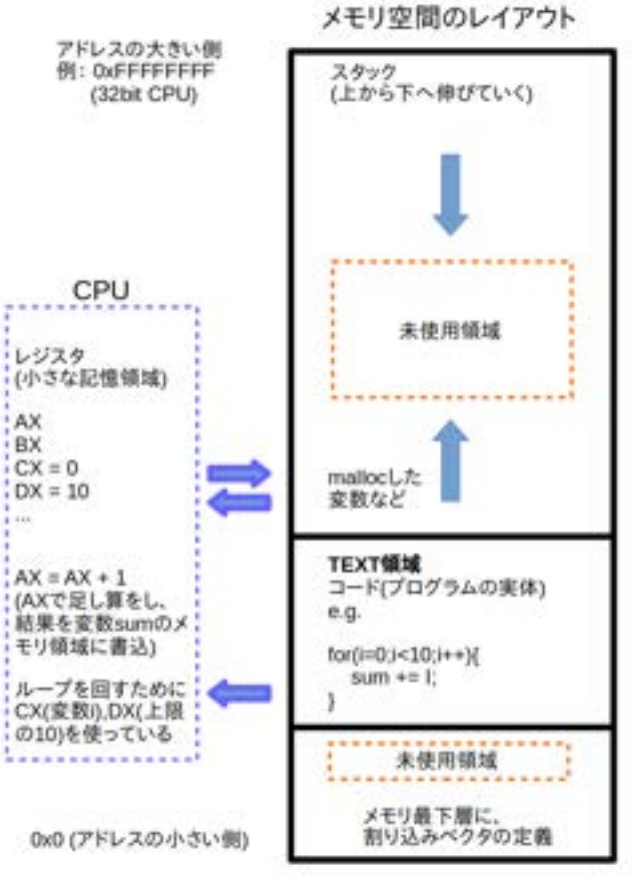
サーバの動作(送信、詳細は省略)

- クライアントへの返事は逆順です
- 送信するデータを用意し、send()などの関数(**システムコール**)を呼び出します
- 逆順に tcp_output -> ip_output() -> ether_output() と進み、 最後はイーサネットの**デバイスドライバ**が呼ばれ、 イーサネットフレームが送信されるといった具合です



プロセス

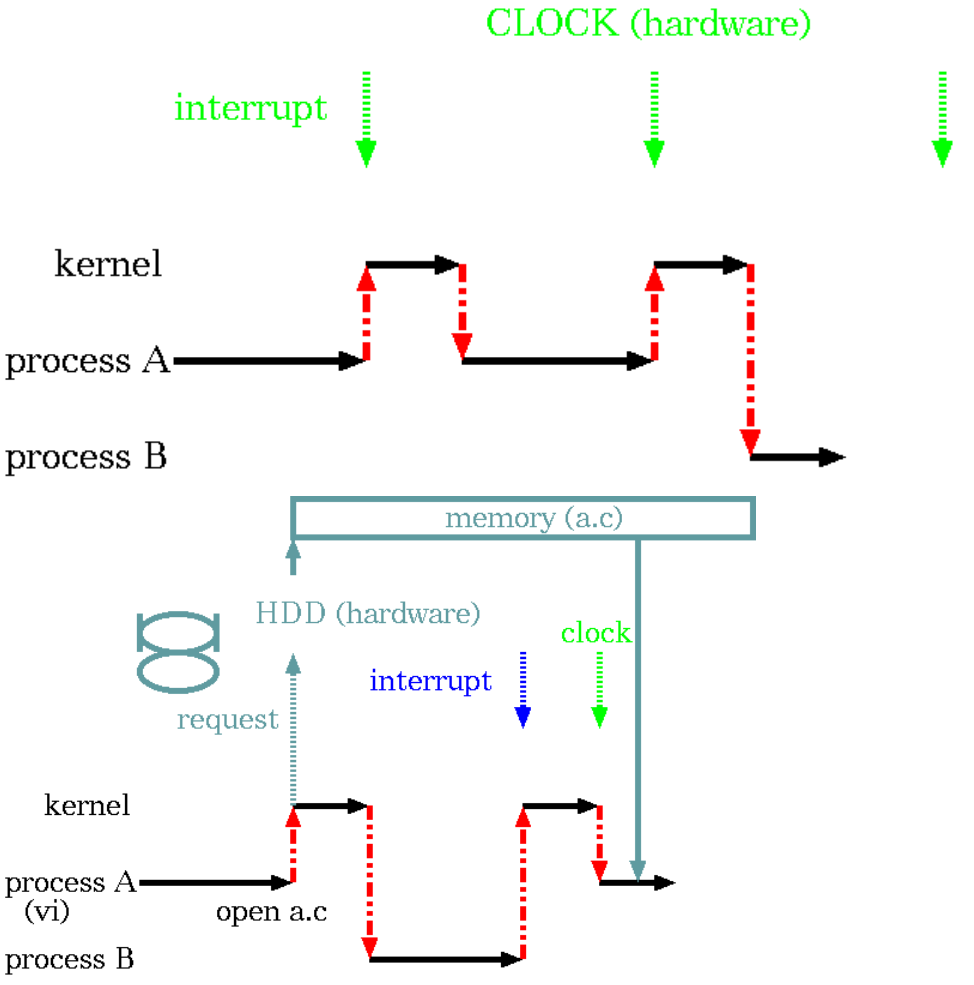
- この例ではWWWサーバというプログラムが動いていますが、 コンパイルしたファイル(たとえばa.out)を、ただメモリに置けば動くわけではありません
- プログラムが動く際には次のようなデータやハードウエアの設定が必要です
  - **メモリ**上に置かれたコード(プログラムの命令の実体,TEXTと呼ぶ)
  - **メモリ**上の変数や**スタック**領域
  - **仮想記憶**システムの管理情報
  - **レジスタ**
- これらの総体を**プロセス**と呼んでいます この図では「WWWサーバのプロセス」が動いて(走って)います



(脚注) 変数まわりの解説を少し端折っていて不正確ですが、雰囲気を説明するスライドです

スケジューラ、リソース制御

- このスライドでは単に「XXX_input()を呼び出す」と説明していますが、 ソースコードが見たとおりに順番に実行されるわけではありません
- **論理的にはコードのとおり**ですが**時間的には飛び飛びに**実行されています
- プロセス群は**（擬似的に）並行処理**されます
  - コマンド、シェル、ネットワーク、カーネル、みな別のプロセスです
  - カーネルが1/100秒ずつプロセスを切り替えて並行処理に見せかけます。 どのプロセスに切り替えるか?を考えるのが**スケジューラ**
  - 同じ関数を（擬似的に）同時実行するため、**ロック**(一般には**リソース制御**)機能が必須



(脚注) つい並列と言いがちですが、並列処理(Parallelism)ではなく、並行処理(Concurrency)という言い方が正しいです

キーボードやマウス

- (サーバには直接関係ありませんが)
- キーボードやマウスからの入力が発生するたびに、 対応するデバイスドライバが呼ばれます
- こういった1バイトずつ転送するデバイスを**キャラクタデバイス**と呼びます。 逆にHDDやSSDなどのブロック単位(歴史的には512バイト、最近は4KBとか8KB) で読み書きするデバイスを**ブロックデバイス**と呼びます

コンピュータの基礎

基本情報処理試験のシラバスを基準に、 用途や特徴などを分類するところからはじめます。
どうしても大枠の用語だらけですが、 基本情報=技術者の常識レベルなので覚えてください

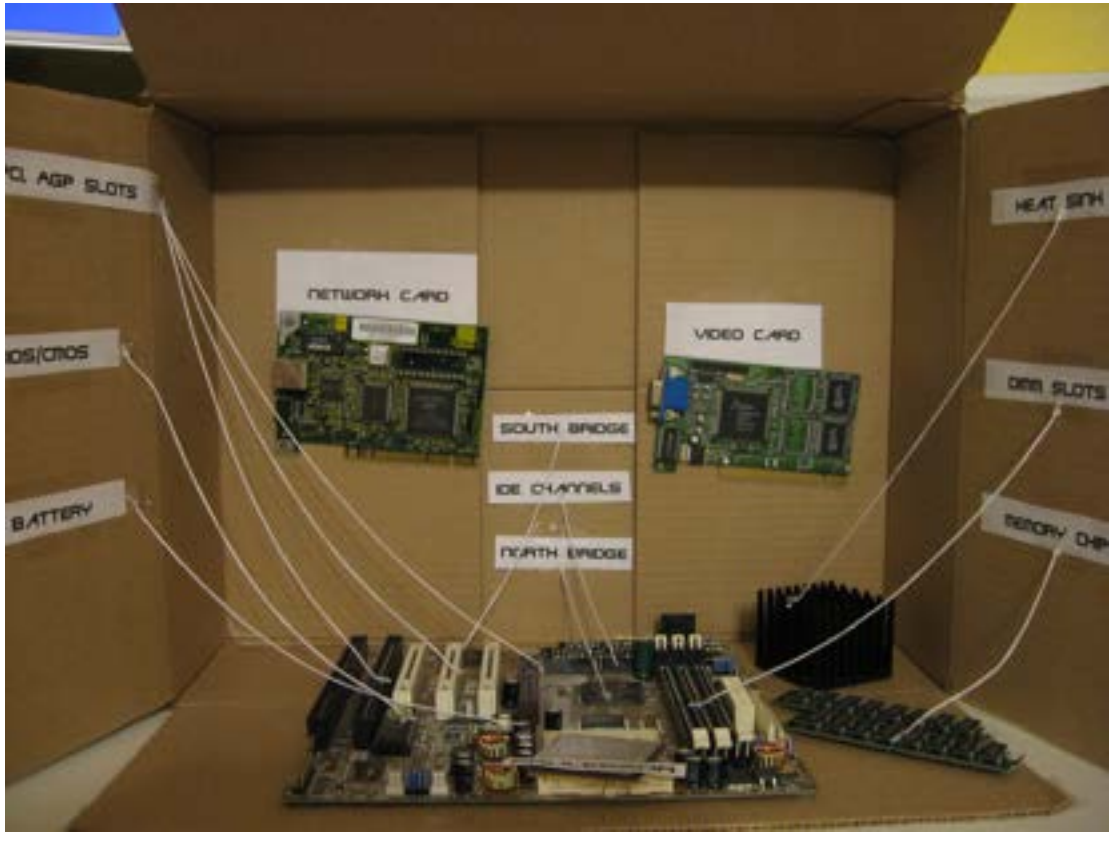
コンピュータの種類

種類	用途	直接操作	身近	具体例
スーパーコンピュータ	科学技術計算	NO	NO	富岳、京、TSUBAME
汎用機	汎用、業務基幹(事務,経理)	NO	NO	銀行、事務処理、業務システム
組み込み	デバイスの制御	NO	YES	家電製品、自動車、工場、発電所
パソコン	汎用、一般のアプリ	YES	YES	デスクトップ、ノートパソコン
ワークステーション	汎用ながら、特定業務利用	YES	?	CADを使う設計、映像処理
携帯端末	汎用、一般のアプリ	YES	YES	スマートフォン、タブレット

- 特徴
  - スーパーコンピュータは科学技術計算に特化した一品物の超高価な製品 (ただし近年のスパコンは、PCやスマホのパーツを元にカスタム化したものが普通。 それら**身近なパーツの価格性能比が高すぎて特注品を作っても太刀打ちできない**から)
  - 組み込みの特徴は特定用途で確実に動くハードウエアの信頼性とリアルタイム性
  - コンピュータの高速化・信頼性の向上が著しく、 各種コンピュータ間の違いは縮小中

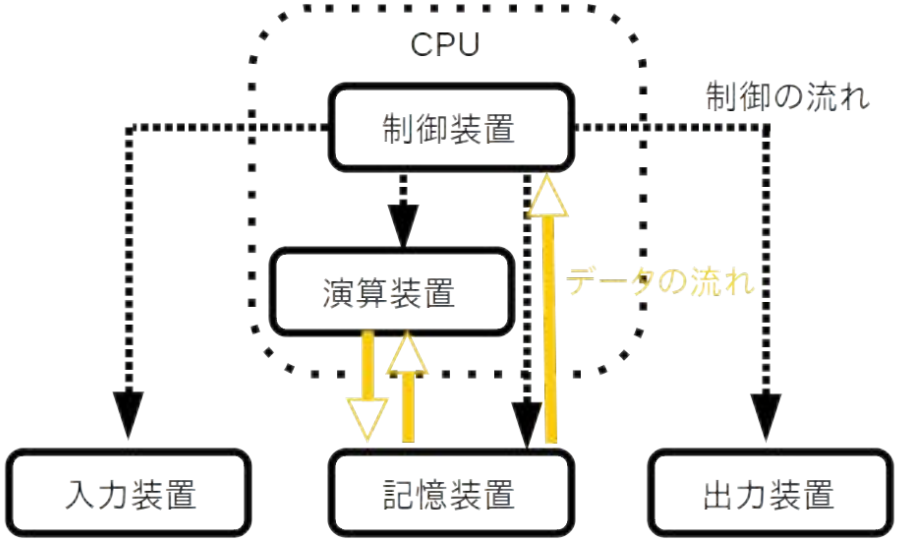
コンピュータを構成する部品

- マザーボード
  - 写真で一番下の緑のボードのこと
  - CPU,メモリ,各種カードを挿すスロット群
  - バスという各部品間のデータ転送をする通信路でパーツ群が接続されています (右図ではスロットの下に隠れて見えてません)。 電力供給ラインも見えませんね (ちなみに直流)
- CPU
- メモリ
- ヒートシンク(CPUを空冷,写真の右下奥)
- ビデオカード
- ネットワークカード



コンピュータの五大要素とプログラム内臓方式

- **五大要素**(右図を参照)
  - **制御、演算、記憶、入力、出力**
- **プログラム内臓方式**
  - 格納方式やストアードプログラム(stored program)方式とも言う
  - まず主記憶装置に読み、それを実行
    1. プログラムを**入力装置**から
    2. メモリ(**記憶装置**)へ読み込み
    3. メモリ上のプログラムを順次実行
       - HDD/SSDなどの**記憶装置**から読みこんだ命令が**演算装置**(四則演算や比較など)や**制御装置**(各装置への指示)で実行される
  - (ハードウエアの変更なしに)**ソフトウエアを変更するだけで、 さまざまな処理が可能**



システムの基礎

ソフトウエアの分類

大分類	中分類	例および小分類
システムソフトウエア		
	基本ソフトウエア	OS
		(a)カーネル本体
		(b)標準ユーティリティ(ls sort システム管理)
		(c)コンパイラなど言語処理系
	ミドルソフトウエア	データベース
応用ソフトウエア		オフィス製品(ワープロ、表計算)

- OSは**基本ソフトウエア**
  - コンピュータはOSがないと非常に使い勝手が悪い=基本のソフトウエア
    - 商用OS: Microsoft社のWindows, Apple社のMacOS X, IBM社のz/OS, 商用UNIX
    - フリーソフトウエア(OSS)のOS: Linux, BSD Unix

(脚注) オープンソース(OSS)の話は（できれば）第7回に取りあげます

システム処理形態の分類: 利用方法、即時性、対話性

処理形態	動作	例
バッチ	一定期間もしくは一定量ためてから**一括処理**	月末の事務処理
		売上データの集計
オンライントランザクション	データの発生と同時に処理を開始(**即時性**)	銀行のATM
リアルタイム制御	センサーで状態を監視し対応(**厳しい即時性**)	自動車のエンジン

- 普段のパソコンやスマートフォンの使い方は、 **対話型**(コンピュータと対話しながら進める)処理が主流
- 上の例でも「銀行のATM」は対話型ですが、他は非対話型

(脚注1) コンビニやスーパーでは全店舗の売上を集計して夜中に集計処理などをしています。 これが**バッチ**処理。 夜中にバッチ処理といえば、汎用機の世界というイメージ。主力言語はCOBOL(偏見?) (脚注2) バッチやオンライントランザクションの処理中枢はデータセンターなどにあるサーバ(群)です。 別途、 対話型の入出力用コンピュータ(補助,本体より低価格)が必要です。 これが本来の**端末**ですが、 いまは普通のPCからサーバにログインすることが多いでしょう。 ちなみに端末(termianl)の訳がターミナルです みなさんの黒い画面もターミナルと呼ばれますが、 正確にはterminal emulationをするソフトウエアです

システム処理形態の分類: 集中vs分散

- **集中処理**
  - データセンターなどに集中管理されたサーバ(群)で処理するため、 **管理**しやすく**機密保護**もしやすい反面、 一部の機材の故障が全体に影響をおよぼすことが多い(分散システムほど冗長化されていない)傾向にあります
  - 性能を向上させるには(機材の入れ替え = **停止をともなう作業**)が必要です（-> **スケールアップ**)
- **分散処理** (いわゆる**クラウドの裏側**)
  - 一部の機材故障であれば（全体には）影響ありません。障害に強い反面、 多数のコンピュータ群からなるため設計も管理も障害対応も難しくなりがちです。 性能の向上は物量戦です（-> **スケールアウト**）
- 性能の向上
  - スケールアップ ... **古典的な少数のサーバ**と多数のクライアントの場合、サーバのアップグレードで対応
  - **スケールアウト** ... **サーバ数の増加で対応**,**無停止**で**ユーザ数に応じて自動増減可**

(脚注1) サーバクライアントモデルは分散処理の一種と考えられます (脚注2) 分散システムは古典的な開発手法では作れません。 分散を前提にサーバソフトウエアとアプリが設計・作成されていなければ、 きちんとシステムの自動拡大・縮小に連動して動作しません

OSの基礎

OSの目的

1. ハードウエア資源の有効活用
2. 処理能力の向上
    - スループット(仕事量/単位時間)、 ターンアラウンドタイム(全部終わるまでの時間)、 レスポンスタイム(ユーザに結果を出し始めるまでの時間)
3. 信頼性の向上
    - 信頼性(正常動作)、可用性(稼働率)、保守性、保全性(壊れにくい)、安全性(機密性;セキュア)
4. 開発効率の向上
5. 操作性の向上

- **移植性**の向上(**開発効率**向上の一部)
    - Unix以前のOSはハードウエアごとの一品物でした。 たいてい、**アセンブリ言語**で書かれていました
    - 1970年代に、**C言語で書かれたOS(Unix)**が**移植可能**なことが証明されました。 これ以降、新しいコンピュータ(ハードウエア)でも、 OSおよびその上で動くミドルウエアやアプリもそのまま動かすことが可能になり**開発効率の向上**につながりました

(脚注) アプリ互換性: アプリ互換性はUnixが最初ではありません。 60年代に発売されたIBMの汎用機がすでにアプリ互換をうたっています。 ただ、アセンブリ言語で書かれたOSなのでIBM以外では動きません。 メンテも大変そう...IBMだから出来た力技?:-)

処理というより利用形態によるOSの分類

- タイムシェアリングシステム(TSS)
  - **コンピュータと対話しながら利用**する形態です
  - 操作画面がある**身近な機器の大半**はTSSなので、サーバもパソコンも**TSS**です
- リアルタイムOS (RTOS)
  - **組み込みシステム = 特定の機能を実現するために機械に組み込まれるシステム**に使われます。 例：家電製品,自動車,医療機器,産業用の機械など、 **身の回りに多数**(さほど対話的ではない)あります。 よって**組み込みOS**は同義語です
    - ちなみに一昔前まで携帯電話もリアルタイムOSが主流でした
  - TSSとの主要な違いはスケジューラで、 高速な反応/**一定時間内に確実に特定プロセスを実行する**ことが求められています。 例: 自動車のエンジン(反応が遅いと致命的)

(脚注1) いまや一人で占有できるコンピュータを複数台もっている時代なので、 タイムシェアリングの**シェア**の意味が分かりにくいです;-)が、 本来TSSは高価なコンピュータを**多数のユーザで共有(シェア)して使う仕組み**のことでした。 20億円のコンピュータを30人で使えるなんてすごい！これが世界最先端だった時代の話です コンピュータ開発史では、 人間がコンピュータをどう使うか? 人間とコンピュータの共生とは? といった観点が出発点にありました (脚注2) 本科目はTSSが前提です

OSの使い分けの例

- インターステラテクノロジズ(株)(大樹町)
  - 大樹町で打ち上げている(いわゆるホリエモン)ロケットを例に使い分け方を説明します
  - 本体の姿勢制御とか噴射のタイミングは、 **処理速度1/1000秒レベル**で行う世界なので**リアルタイムOS**
  - 周辺機器(通信とかタンク内をかき回すとか?)は、そこまでcriticalではないので、 ラズベリーパイ(LinuxつまりTSS)で動かしているそうです

"Apollo 11 liftoff" by Apollo Image Gallery is marked with CC PDM 1.0

(脚注) 2021年には100kmまで上昇してペイロードを放出=宇宙に行けた、そうです。おめでとう！

構成法によるOSの分類

- **モノリシックカーネル**
  - カーネルが一つの巨大なソフトウエア、 代表例はUnixファミリー(商用UNIX, BSD Unix, Linux)
  - **マイクロカーネルにくらべ(a)動作が高速(b)作成も容易(c)デバッグも容易**
  - デメリット
    - 小さなセキュリティホールがカーネル全体に波及しうる
    - カーネルをアップグレードする際はOSの再起動が必要
- **マイクロカーネル**
  - カーネルは必要最低限の機能だけをもつ小さなもの(だからmicroなkernel)とし、 多くのプロセス（サーバプログラム）群が協調してカーネルの機能を提供するモジュラー(modular)な作りです。
  - **OSを小さな部品群から構成できるか?というOS研究の産物**です
  - 80年代の有名な**Mach**(マークと発音)はWindowsやMacOS Xの御先祖にあたります

(脚注1) つまり身の回りで良く見るPCやデバイスは、ほぼ全部UnixかMach系と言えます （注：組み込み機器は別です）

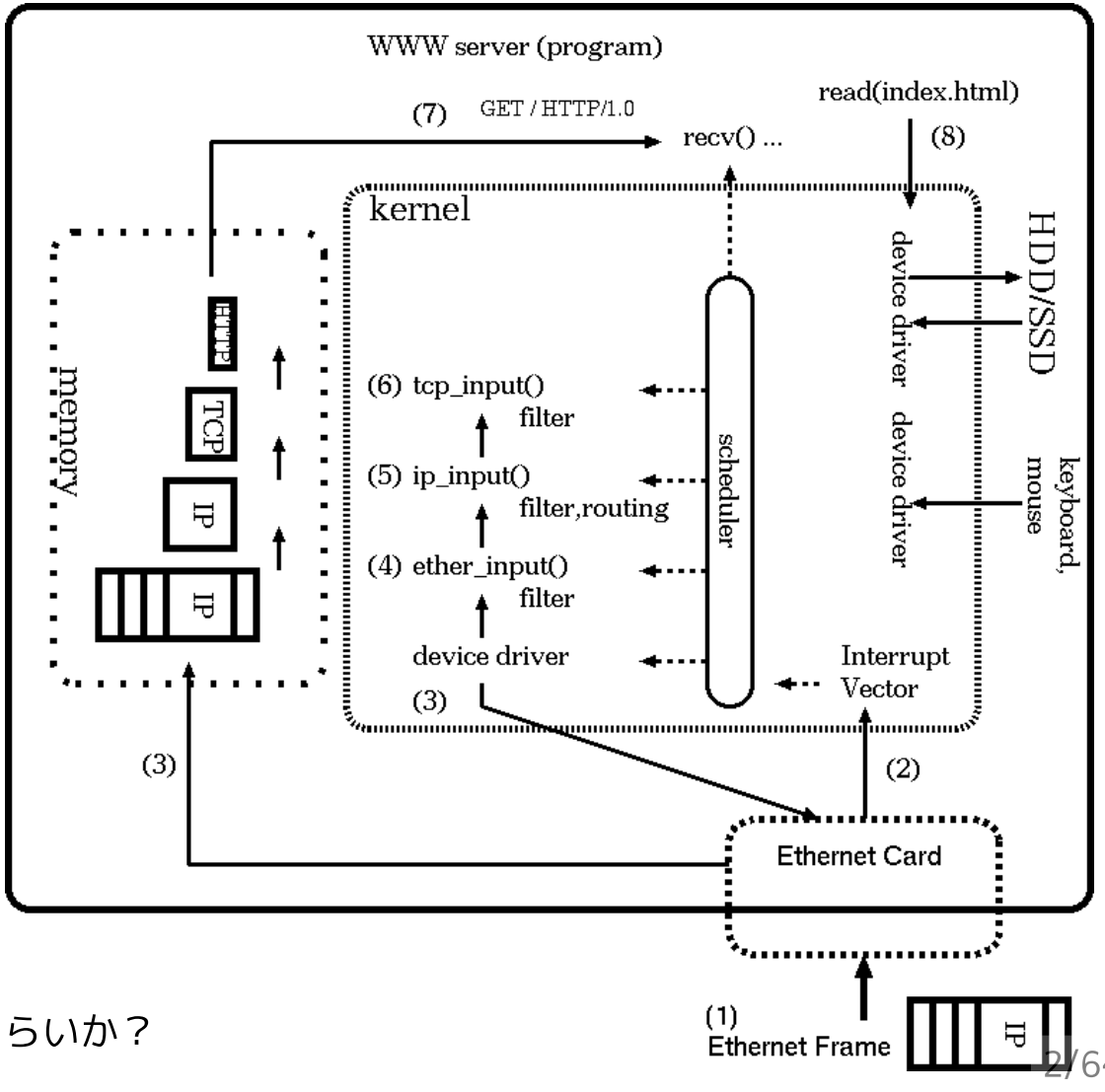
(脚注2) Machなどは遅くて使えませんでした。 1990年代以降にあると、 マイクロカーネル再評価が起こりますが、 その話は省略

(脚注3) WindowsはMachを設計の参考にしました。 MacOS Xは、カーネルがMach直系の子孫、ユーザランドはBSD Unixのゴッタ煮

情報技術応用特論 第05回
OS(2): OSの中核部分

シリーズ構成、サーバの全体像との関係

- シリーズ構成
  - 第05回
    - CPU
    - プロセス
    - メモリ（仮想記憶）
    - 割り込み
  - 第06回
    - 競合状態と排他制御
    - デバイスドライバ
    - ストレージ
- 今回（第05回）
  - 前半2/3（CPU～メモリ）は図の裏側にある
    ハードウエアや仕組みの話と言えます



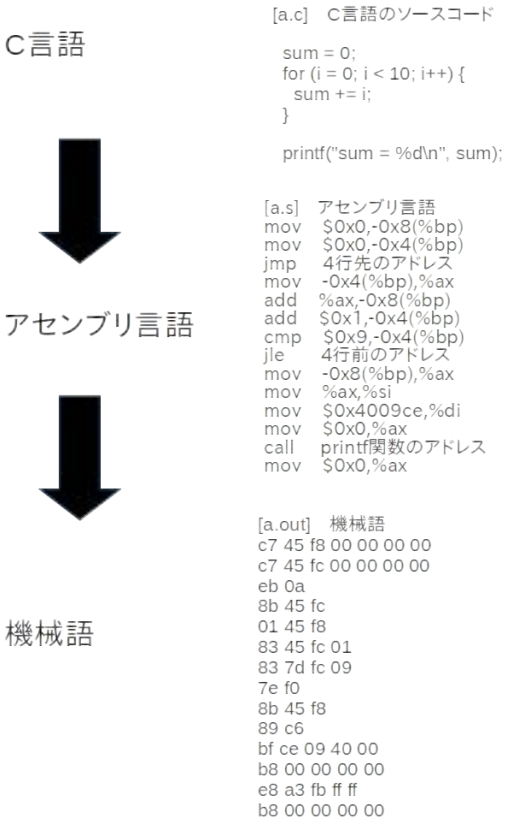
(脚注)「カーネル」「WWWサーバプロセス」はプロセスの一つ

「スケジューラ」はプロセス制御のくだりで登場しますが、それくらいか？

CPU

CPU動作の概要

- 演算と制御装置(五大要素)
    1. プログラムを**入力装置**から
    2. メモリ(**記憶装置**)へ読み込み
    3. メモリ上の**機械語**を順次実行
        - 実行時の一時領域などには**レジスタ**（CPU内の小さいが高速な記憶領域）を使い様々な処理をしています
- CPUが解釈できるのは**機械語**だけです
    - **コンパイル**とは機械語への翻訳です
        1. ソースコード(例:i=0) ->
        2. アセンブリ言語(例:mov $0,ax) ->
        3. 機械語(マシン語,例:0xb800)



(脚注1) レジスタがあるので、CPUは電卓より少し賢い動作ができますが、 大雑把にはCPU=超高速な電卓というイメージです。

(脚注2) 電卓だけあっても、記憶装置などの周辺機器が無いと、ぜんぜん便利な機械では無いです

機械語を1対1翻訳したものがアセンブリ言語

- 機械語は数字(前頁参照)ですが
- 人間が機械語を直接読み書きするのは辛い
- アセンブリ言語で読み書きします/できます
    - これは**機械語を1対1翻訳した**言語

(脚注1) 授業では、これ以上の詳細は割愛しますが、 基本情報処理試験の範囲は意外と広く、 レジスタやアセンブリ言語も、もう少し細かいところまで求められます。 例: アドレス指定の直接,間接,相対(アドレス指定) ... C言語のポインタのポインタといったポインタまわりの機械語表現のようなもの (脚注2) 基本情報の午後の部では簡易化したアセンブリ言語でも受験できます。 実はアセンブリ言語を選択するのが(難しい文法がないし)一番簡単というううわさ有り

(脚注3) 昔の映画のように「科学者が紙テープを読む(右図)=機械語を生読み」なんてしません(w)

例: アセンブリ言語

```
例: siレジスタへaxレジスタの値をコピーする
機械語 <-> アセンブリ言語     // 解説（機械語より、だいぶ読みやすいよね?)
89c6        mov    %ax,%si    // movが命令部、%ax,%siがオペランド部
                              // これはAT&T記法、Intel記法(この例はmov si,axになる)もあります
                              // 授業で使っているGCC(GNU Compiler Collection)はAT&T記法

[むりにC言語っぽい記述をすれば]
register ax, si;
register_copy(ax, si);
```

- 機械語（マシン語）: a.outを生で読むと 89c6 のような数字が並んでいます
- アセンブリ言語では、CPUの1命令が1行ずつに翻訳/出力されています
  - 89がmov命令(注：moveの略ですが実際の動作は**コピー**です)
  - c6の部分がaxレジスタとsiレジスタ
  - あわせると`mov %ax,%si`(movが関数名、引数が%axと%siと思えばよい;スペース区切り)

(脚注) アセンブリ言語でループを解説する例が付録にあるので参照してください

クロック周波数とコア数、その歴史的展開

- **クロック周波数**
  - 電圧の上がり下がりが処理の基本単位になっていて、 各部品間のタイミング合わせをしています。 これが**クロック**
  - CPU内部の周波数が**内部クロック**、 CPU外部(いわゆるバス)は**外部クロック**で内部より数倍おそいです
- **内部クロックをあげる=高速化**につながりますが、 2000年代後半以降は**コア数**を増やして性能を向上
  - **クロック数による高速化は3GHz付近で限界点に達した**(脚注2)
  - このあとの世代、いわゆる**Intel Coreアーキテクチャ**では、CPU単体のクロック数は3GHz付近までで、 プロセッサ(手のひらサイズのLSI)に（旧）CPU複数個分をいれるようになりました （この個数が**コア数**）

(脚注1) （後述するパイプラインやRISCの工夫でも速くなりますが） クロック数=1秒間に実行できる命令数が増えれば、そのぶん確実に速くなります (脚注2) 2000年代前半のIntel Netburstアーキテクチャ（製品例：Pentium D)。 クロック数を上げることはできても、3GHzあたりを越えると、 使うエネルギー(電力)=発熱のわりに高速化しなくなってきたというのが当時の見解です。 いまのCPUの最大クロックは、もう少し大きいですが、それでも数GHzが最大です (脚注3) 個人向けPCではコア数個でしょう。 業務用サーバでは、コア数が64〜128個などの製品が使われています。 仮想環境を提供するサーバ（いわゆる仮想母艦）では、 サーバの物理的な体積あたりのパワーが大きい方が良いです。 データセンターへのサーバの収容効率が良くなるから（-> プロバイダの経営の話）

CPUの大設計方針

- **CISC = 多数の命令語**があるCPUの設計
  - 複雑な処理が出来る命令群
- **RISC = 命令語が少なめ(厳選)の設計**
  - 単純な命令群
  - このほうがパイプライン（後述）に適しています
  - 最適化はコンパイラが頑張ります
  - （集積密度があがり複雑化する一方の）CPUの設計が楽になります

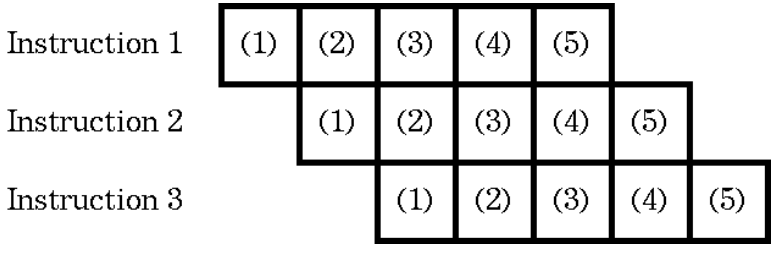
(脚注1) CISCはCompilicated Instruction Set Computer、RISCはReduced Instruction Set Computer の略

(脚注2) かつてコンピュータの性能が低かった時代には、 一発の命令でハードウエアつまりCPUに複雑な仕事をやってもらうほうが良いと考えられた時代があって、 CISCは、その産物と考えられます。 しかしながら、 CPUの使い方を実測してみると、複雑な命令の使用頻度は低いことが分かりました。 大変な思いをしてCISC CPUを作りこむメリットが無いことが判明したわけです

(脚注3) 商用インターネット時代の主流はRISC系です

CPUの高速化技術

- 逐次制御 ... 機械語を順番に実行します
- **パイプライン** ... 機械語の命令を平行して実行します
  - 命令の内容によっては必ずしも逐次実行する必要が無いことを利用します
    - 例：「メモリから読み込む」「加減乗除」は全然別の処理です
  - 可能な限りCPUの全部品に仕事をさせたい（遊ばせない） -> 効率化=速度向上
  - うまくいけば並列処理をしているかのように速くなります



(脚注) さらにステージを細かく分けるスーパーパイプラインや、 パイプラインを複数もつスーパースカラという技術も使われています。 これらは基本情報の範囲ですので、各自で勉強してください:-)

GPU

- GPU
  - 画像処理に特化した浮動小数点演算を並列実行できるプロセッサ
- GPGPU (Generic Purpose GPU)
  - 画像処理以外の分野に流用すること
    - 機械学習
    - 暗号解読
    - データマイニング

- [応用]スーパーコンピュータ
  - 科学技術計算は浮動小数点演算なので、GPUボードを束ねればスーパーコンピュータになります
  - 電力効率が高い(**省電力**)
  - コストパフォーマンスがよい(?)

(脚注1) GPUこそRISC的な発想の路線上にあると言えるよね？RISC CPUよりはるかに単純で並列度が高い部品ですけど

(脚注2) いまどきのGPUボード上位機種なら並列実行できる演算器を数千個〜数万個は搭載しています

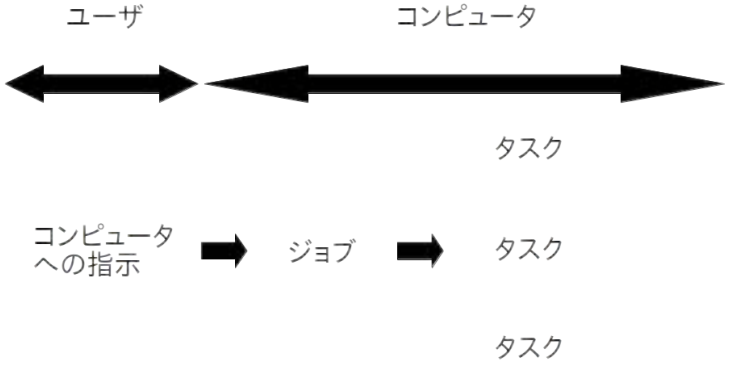
(脚注3) 電力効率の高いスパコン ... 例: 数世代前の話ですが、 東工大のTSUBAMEはNVIDIA GPUを大量に並べたスパコンで、GREEN500(速度/電力の勝負,2013)でTOPでした (TSUBAMEは「みんなのスパコン」で、 東工大 の人は誰でも使えたらしい、いいな〜

(脚注4) 本当に「コストパフォーマンスがよい(?)」かはGPUベンダからの仕入れ値しだい。 そもそもスーパーコンピュータ自体が一品物/特殊案件で、 普通の仕入れ値ではないはずだから、正直よくわからないです。 法人向けビジネスなんて全部そんなものなので:-)

プロセス

【用語】仕事の処理単位

- コンピュータ->**ジョブ**->**タスク**と細分化
- **ジョブ**=ユーザがコンピュータに指示する単位、**タスク**は**ジョブ**を分解したものでコンピュータからみた仕事の単位 (そういう用語があります)
- 以下、本稿ではOS上での仕事の単位を**プロセス**と呼びますが、 OS内部を解説する本稿の文脈ではタスクとプロセスは同義語です (Unixではプロセス、(Mach由来の)MacOSXやWindowsではタスクと呼んでいます)

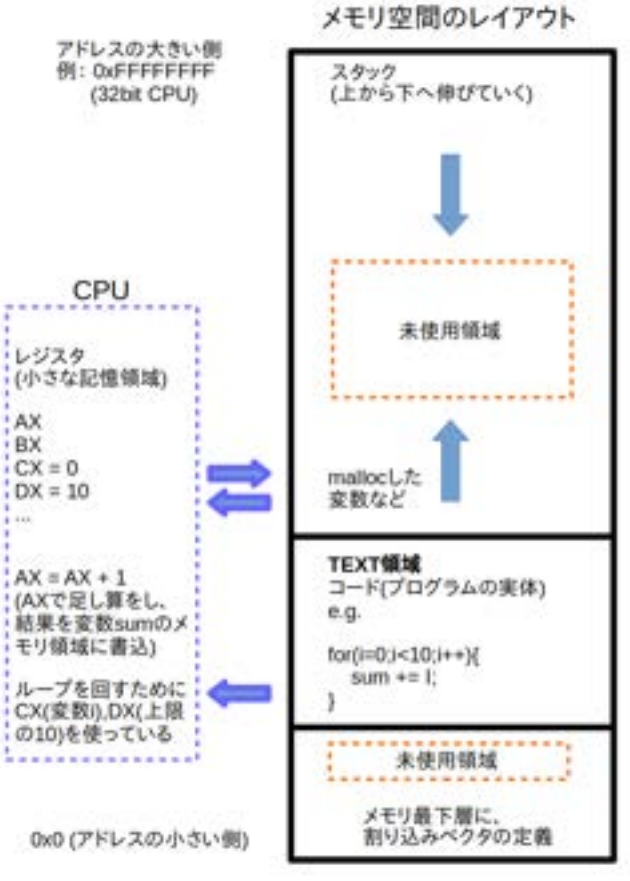


(脚注) 汎用機文化には「ジョブを流す」と言う言い回しがあります。 この手の用語の話（いわゆる宗教戦争っぽい話）は、 どちらかというと汎用機とか規格を決めたがる人たちの文化の雰囲気があります。 実際、Unix用語ではjobとtaskとprocessをきちんと分けて使い分けてないですし... (例: processから起動したjobのcontrol)

【用語】Unixにおけるプロセス

- プログラムが動く際には次のようなデータやメモリ領域、CPUの設定、管理情報が必要で、 これらの総体を**プロセス**と考えてください

- プロセスの構成要素（右図については後述）
  - **メモリ**上に置かれたコード(プログラムの命令の実体,**TEXT領域**)
  - **メモリ**上の変数や**スタック**領域
  - **仮想記憶**システムの管理情報
  - CPUの**レジスタ**
  - 開いているファイルの情報など

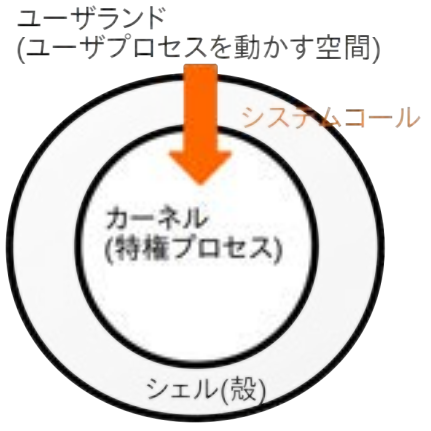


(脚注1) プロセスの詳細は覚えなくてOK。 若干の解説あり -> 付録を参照

(脚注2) ユーザプロセスを動かす環境を**ユーザランド(userland)**と呼びます。 システム構築演習とは、ほぼ**ユーザランドの操作**です

【用語】Unixにおける特権プロセス

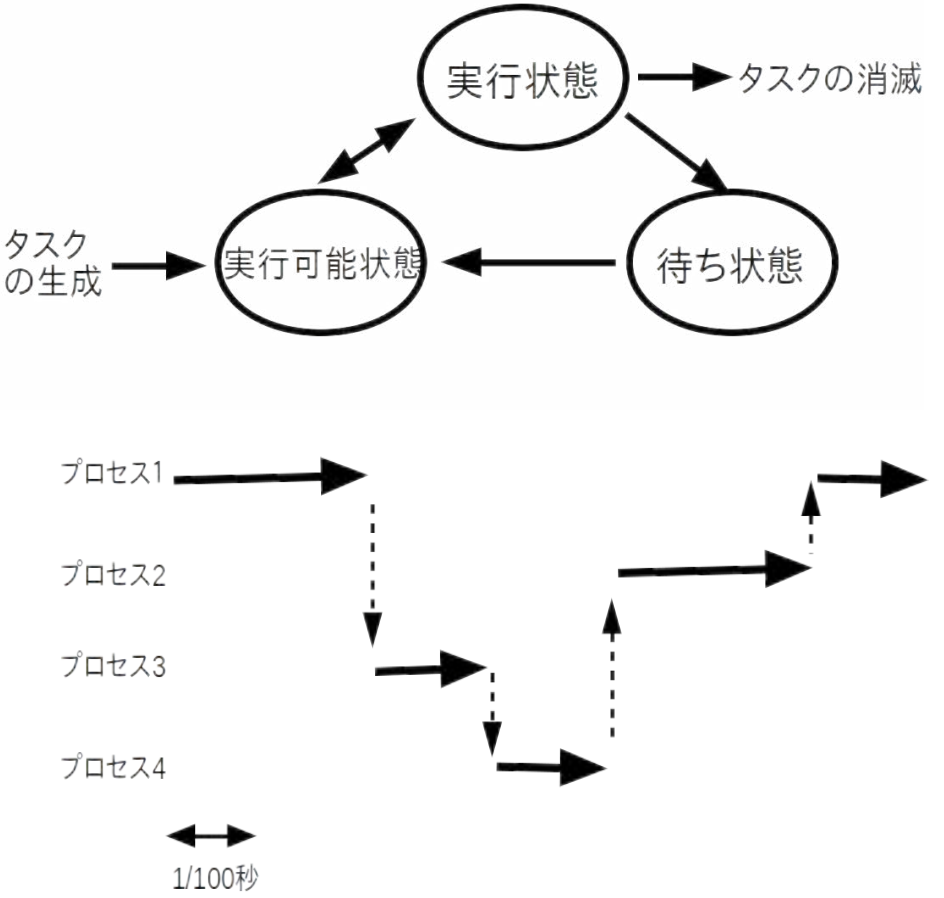
- プロセスには**特権**あり/なしの2種類
  - 図(右)のような階層モデルで**内側が特権(中央がkernel、外がuser)**と考えています
- **カーネル (kernel)** ... 特権あり
- ユーザ (user) ... 特権なし
  - たとえ話：システム構築とはイースターエッグに色を塗っているようなものではないでしょうか？:-)



(脚注1) ユーザプロセスを動かす環境を**ユーザランド(userland)**と呼びます。 システム構築演習とは、ほぼ**ユーザランドの操作**です

プロセスの状態遷移とマルチプログラミング

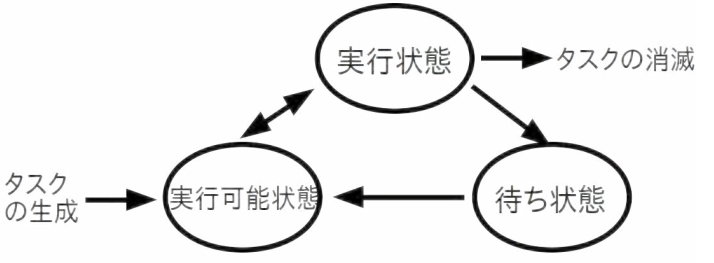
- プロセスには3つの状態があります: **実行可能状態、実行状態、待ち状態**(右上図)
- ある瞬間の**実行状態プロセスは一つだけ**です
- （カーネルの中の）**スケジューラ**が、どのプロセスを実行するかを決定しています
  - たとえば1/100秒ごとに実行状態プロセスを切り替え、 人間の目にはプロセスが同時実行されている幻影をみせています。
    この動作を**マルチプログラミング**もしくは**マルチタスク**と呼びます(右下図)

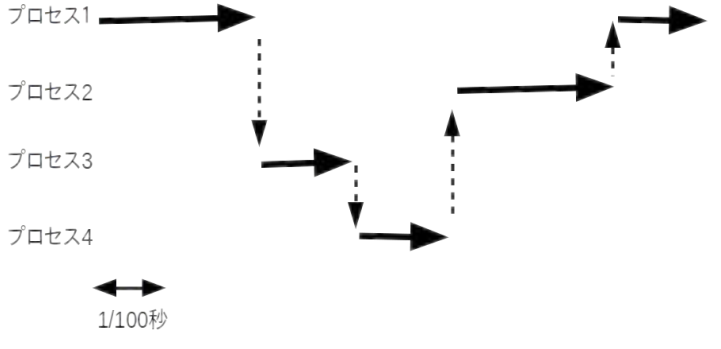


(脚注1) いまどきは「実行状態プロセス==コア数」です（コア数が1の場合、プロセスは一つだけ） (脚注2) つまりアニメと一緒です:-)

スケジューラとスケジューリングアルゴリズム

- 選ぶ方法=**スケジューリングアルゴリズム**の例
  - ラウンドロビン方式 ... 順番に一定時間で代わりばんこ
  - **優先度方式** ... プロセスごとに優先度という概念があり、 再計算して適切なものを選ぶ。 「**対話的プロセスを優先**」などの細かい調整が出来るのでTSS向き





(脚注) 教科書的な説明をするなら、 schedulerが順番を決めて、 切り替えをするのがdispatherとか、 きちんと用語を使い分けないといけないのでしょうが、 実際のカーネルコードが綺麗に書き分けてるとも限らないので、 このへんは端折ってます

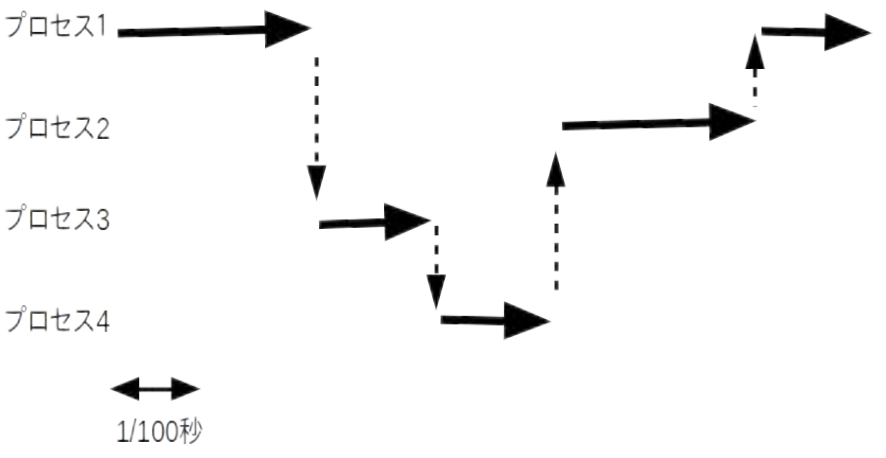
マルチプログラミングのメリットとデメリット

- メリット：OS全体での**利用効率向上**です
  - あまりにも違う**デバイス間の速度の差**を相殺できます。
  - プロセスAが待ち状態のときにプロセスBを実行すればCPUが遊んでいる時間がなくなります
    例：プロセスAが「キーボード入力待ち」や「HDDへの入出力待ち」をしている時、プロセスBを実行できます
- デメリット：プロセスの切り替え（**コンテキスト切替**）(後述)のオーバヘッドです
  - TSSでは、 このオーバヘッドがあっても、メリットのほうが大きいと判断されているわけです

(脚注1) タイムシェアリングシステム(TSS)はコンピュータを対話的に使う方法です。 OSはキーボードやマウスからの入力を待ち、処理し、結果を出力しています。 人間の操作（キーボードやマウス）、HDD、CPU間には数億倍の速度差があります。 例えば、キーボード入力を一文字待つあいだにCPUは数億回の足し算ができます

コンテキスト切り替え

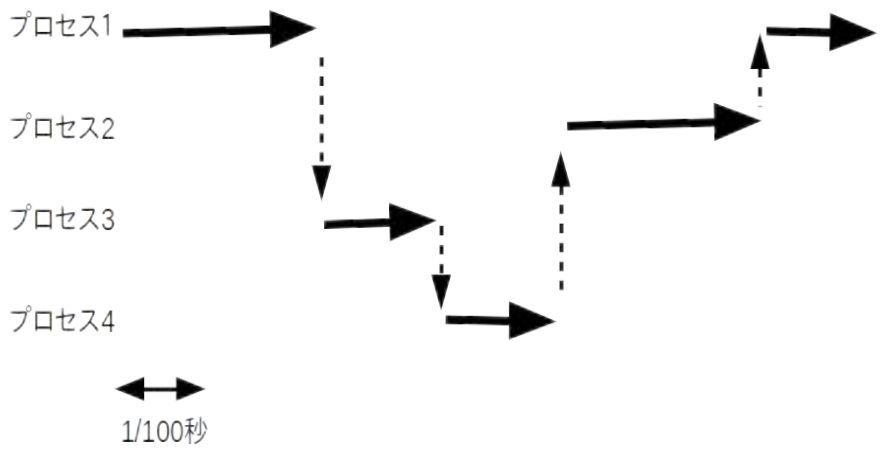
- プロセスの構成要素をコンテキスト(context)と呼んでいます。 これはプロセスを再現できる制御情報のすべて(プロセスの総体と同義)です
  - CPUレジスタの値,メモリ管理情報,カーネルのもつプロセス制御情報,ファイル他
- **コンテキスト切り替え**とは
  - プロセス1のコンテキストを保存
  - プロセス3のコンテキストを読み出し再設定
  - プロセス3を実行する(= 3に切り替える)



(脚注1) コンテキスト切り替えは、英語でcontext switchです (脚注2) 基本情報レベルでは**太字の用語だけでOK**です。 (脚注3) この説明も端折ってますが、 実際のカーネルコードを読まないとcontext switchの技術詳細は分からないと思います （でもオオゴトになるので割愛）。 モヤモヤする人はUnix kernelのソースコードを読んでね。 細部はOSバージョンやCPUごとに微妙な差異がありえます

プリエンプション(Preemption)

- **プリエンプション**(Preemption)
  - Preemptとは強引に横取りすること
- 割り込みの例
  1. (割りこみがかかると) プロセス1がPreemptされ**強制的に処理が一時中断**されます
  2. 割り込みの仕事をするプロセスへ切り替えが起こり ...
     (以下略)

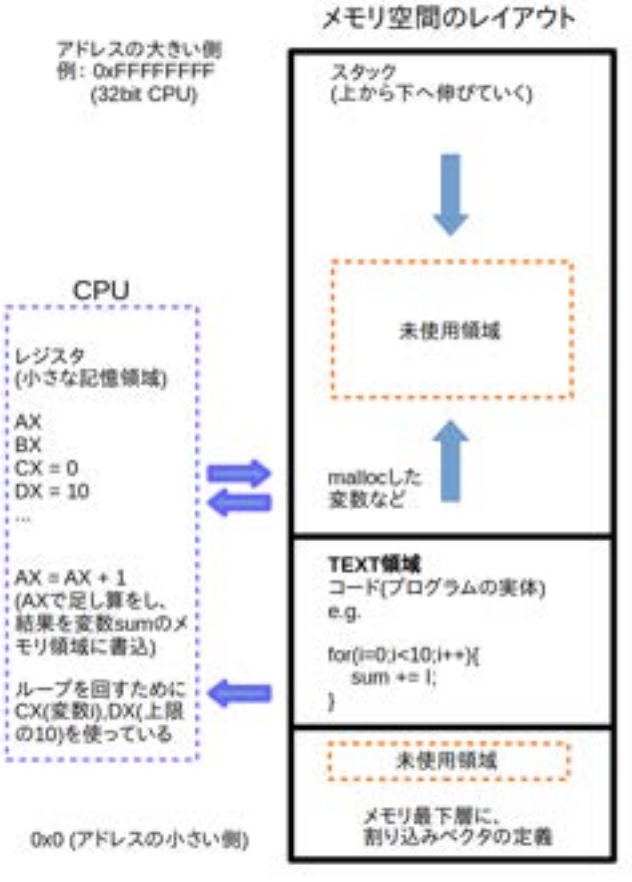


(脚注1) **割りこみ**(interrupt)は、このあと解説します

(脚注2) カーネルも（特権を持っていますが）単なる一つのプロセスです。 カーネルプロセスに切り替えないとOSの管理が出来ません

プロセスのメモリ空間レイアウトの例

- 右図のアドレスは下が小さく(0)->上が大
  - 理論上の最大値は32ビットCPUが4GB(=32bit)、64ビットは16EB
- レイアウト(上から下へ解説)
  - 上から下へ**スタック**領域が伸びる(関数呼び出しやローカル変数で利用)
  - 真ん中あたりは未使用
  - データやローカル変数(mallocするもの)の領域
  - アドレスの小さい側にTEXT領域
    - みなさんのa.outの実体はここ
  - 0バイト付近には割り込みの定義
- **どのプロセスでも同じレイアウト**です。 衝突しないの? これは物理ではなく**仮想メモリアドレス**なので衝突しません (-> 「仮想記憶」節を参照)



(脚注1) 応用情報のレベル (脚注2) **メモリレイアウトの詳細を気にしなくてOK(OSの有り難味)**

【参考】【余談】きみはコンテキスト切り替えがイメージできるか？

- 次の動作をイメージしながらkernelソースコードが読めるか?/納得できるか?次第で、 **OS動作の分かりづらさ**が変わってくると思います
  1. 並列実行されているかもしれない
  2. 突然、任意の場所で中断/再開がありうる

```
sum = 0;
for (i = 0; i < 10; i++) {
  sum += i;
}
```

```
c7 45 f8 00 00 00 00
c7 45 fc 00 00 00 00
eb 0a
8b 45 fc
01 45 f8
83 45 fc 01
83 7d fc 09
7e f0
8b 45 f8
89 c6
bf ce 09 40 00
b8 00 00 00 00
e8 a3 fb ff ff
b8 00 00 00 00
```

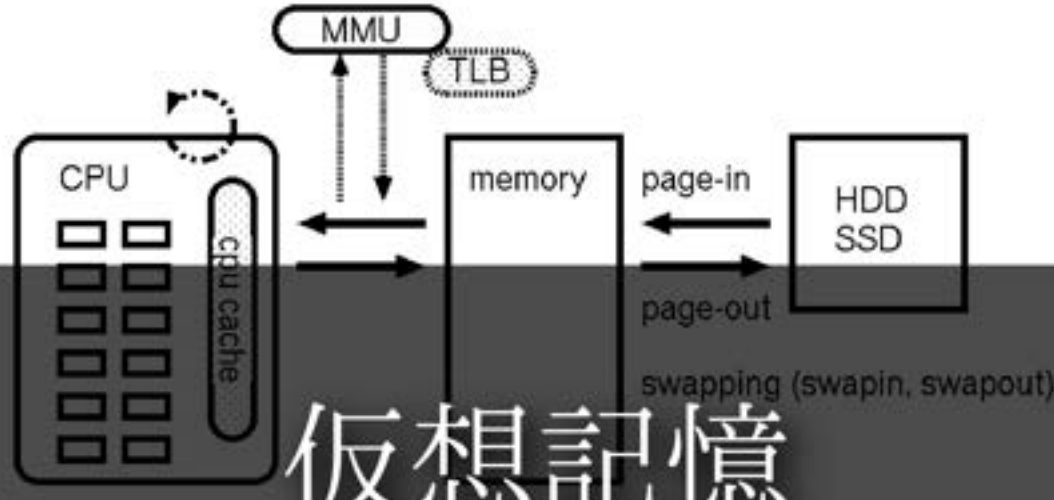
- いまどきのOSはFully Preemptiveに書かれていて、 アセンブリ言語 の任意の場所(行単位)で中断がかかります
- C言語で表現すると、 一行単位どころではなく一行の途中で突然中断し再開がいつか?も不明な振る舞いです (C言語をアセンブリ言語に翻訳すると一行が数倍に膨れ上がるのが普通で、 そのアセンブリ言語の任意の行で中断/再開します。左のコード例でいえばforループの条件部`i < 10`ですら条件判定の途中で中断がありえます)

(脚注) このページは中間試験に出ません:-)

(アーカイブ動画で見ている人も)一時停止して休憩

「30分ごとに雑談（休憩）をしろ」というのが指導教官の教え

正確には「アメリカでは30分ごとにジョークを言わないといけない」だけれど、そんな洒落乙なこと無理:-)



仮想記憶
(Virtual Memory)

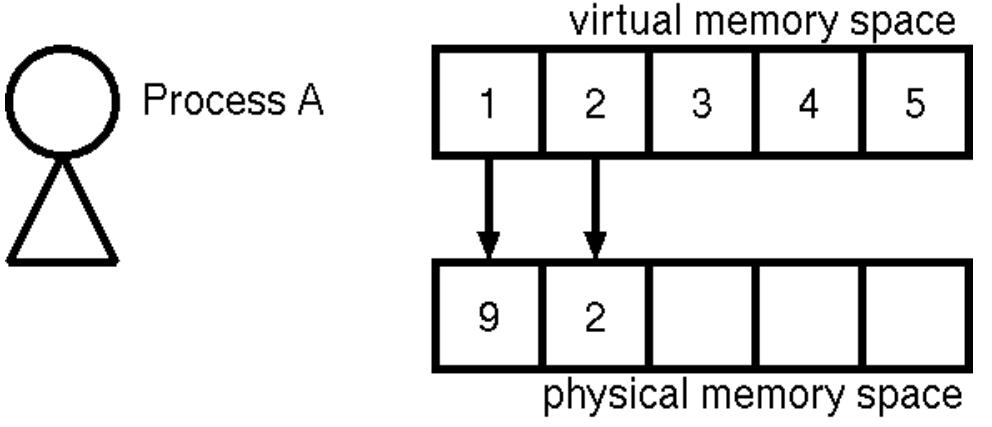
メモリと仮想記憶システムの意義

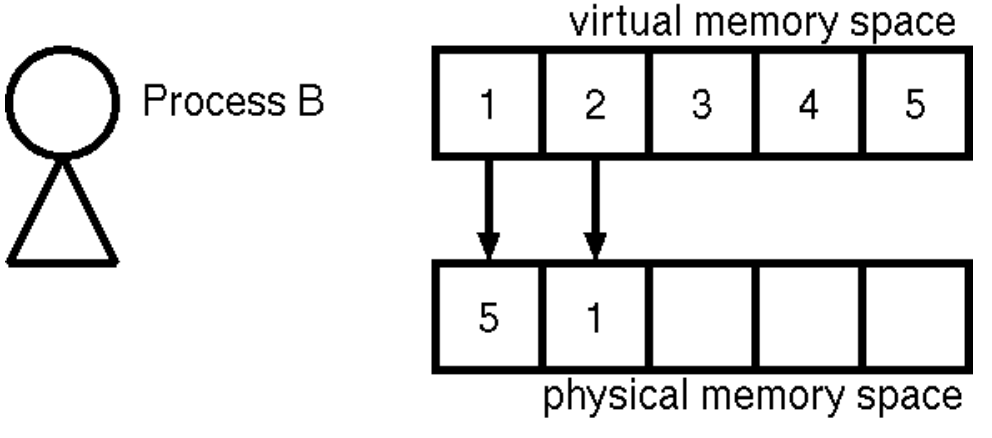
- メモリ(main memory)つまり主記憶装置(ハードウエア)と、 仮想記憶(virtual memory)システムについて取り上げます
  - タイムシェアリングシステムのような複雑なOSでは**仮想記憶**は必須です
  - 1980年代以降のOSで主流の**ページング型仮想記憶システム**だけを解説します
- 仮想記憶システムのおかげで、 システムの**物理的な詳細を気にせず、 一続きの大きなメモリを扱え、 プログラミングが楽**になります

(脚注1) 基本情報の出題範囲には、ページング以外にセグメンテーションやオーバレイなどの手法も含まれます (脚注2) 逆に組み込み機器の場合、仮想記憶が不要なこともありえます (脚注3) 仮想記憶は「**開発者の時間はコンピュータ（ハードウエア）より貴重**」という思想を実現するものの一つと考えられます そもそも、 **ビジネス**目線の発想は「**(納期によるけど)開発しなくてすむなら、それは大勝利**」です。 プログラマが頑張って10%速くチューニングしても、 一年以内に出荷される次期Intel CPUが10%くらい速くなるので、 ハードウエアアップグレードのほうが確実です(ソフトのバグも増えない)。 なにしろ人件費が一番高いですからね！ (脚注4) **なにかを開発する**ことが一番費用がかかる方法なので、それは最後の手段。「DXがほげ〜」と叫ぶ人の大半が理解していないことがコレ

仮想記憶システムの要諦

- 右図の**箱(1〜9の数字)はメモリの概念図**です
  - **ページ**が処理単位（詳細は次頁以降）
- プロセスAもBも仮想ページ1〜5(上側)を使っていますが、物理ページ(下側)は異なります。 各プロセスは同じ(仮想)メモリ空間(1〜5)を使っていると**錯覚**していますが、 実際に読み書きする物理メモリの場所は異なります
- **物理的な詳細を気にせずプログラミングできるのは、この仕組みをOSが提供しているから**です

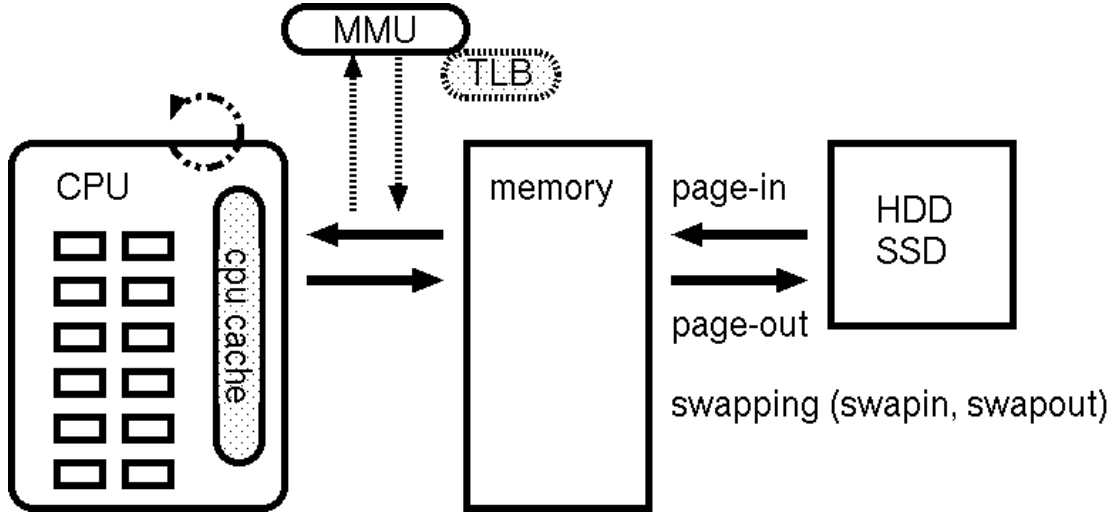




(脚注) 今回の白眉がコレ。 正確には**ページング型仮想記憶システム**の解説です

仮想記憶(Virtual Memory)とは?

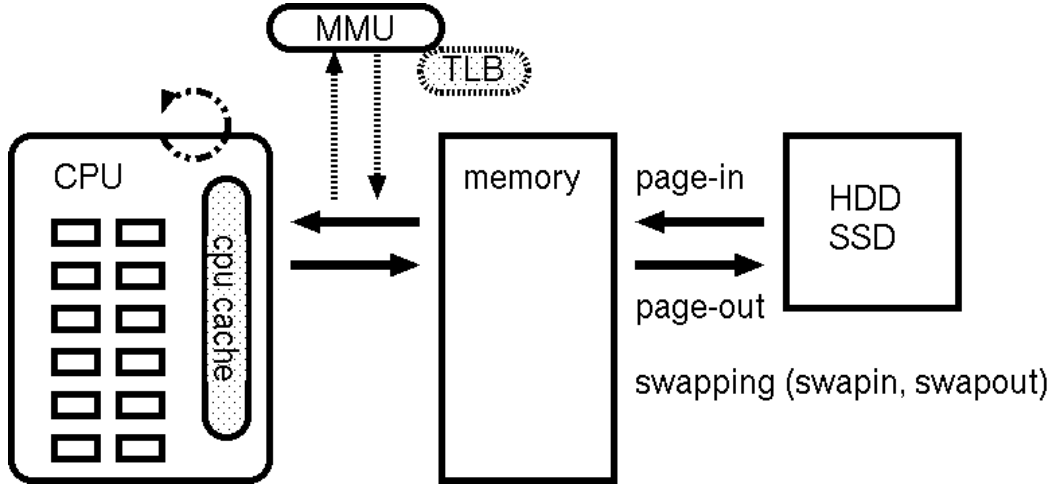
- プロセスに見せているのは**仮想アドレス**空間
- 仮想記憶システムを使うことで (a)**物理メモリの詳細を気にせず** (b)**実際の物理メモリより大きなメモリ空間**を扱えます
  - (a)によるメリット: **プロセスには仮想アドレス空間がリニア(直線的,一続き)**に見えます
- 実装: CPUがメモリへアクセスする際、 (こっそり)**MMU**という回路が**仮想アドレスを物理アドレスへ自動変換**します。 通常MMUは**CPUに搭載**され、 **ページテーブル**(仮想->物理アドレス変換ルール表)の管理をカーネルが行います



(脚注1) 仮想アドレス = virtual address (脚注2) MMU = Memory Management Unit (脚注3) かつて（70年代〜80年代あたりでは） 「(b)実際の物理メモリより大きなメモリ空間」に意味がありましたが、 現在の64ビット環境では逆の条件になってしまったので忘れてOK

仮想記憶のメリット、デメリット

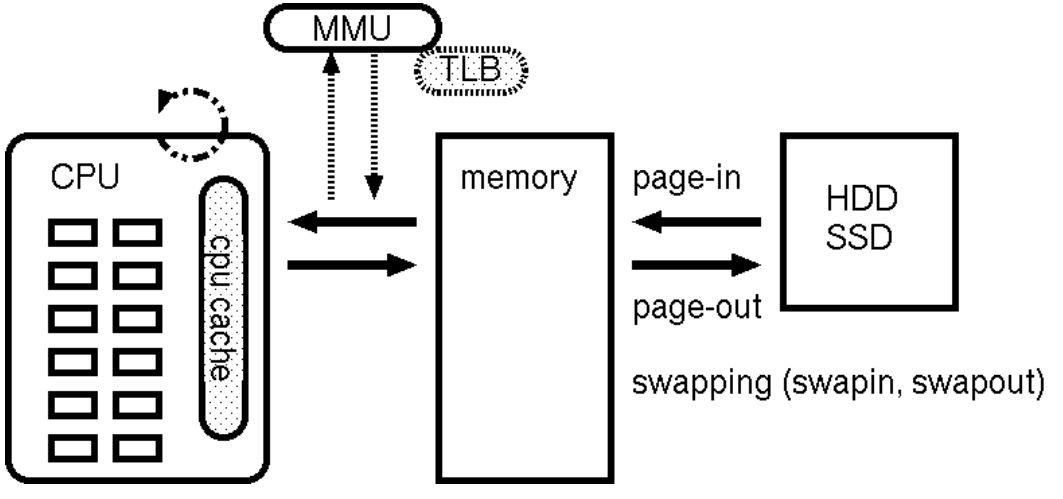
- メリットは開発効率の向上です。
  - (a)**メモリの詳細を気にせずにプログラムを書ける**ことが重要です
  - (b)物理メモリより大きな仮想アドレス空間が利用可能にもなりますが、 遅くなるので現在では重要視しません(脚注も参照)
- デメリット
  - (a)実装が複雑になること
  - (b)ほぼMMUが必須であること
  - (c)アドレス変換オーバヘッド



(脚注1) 物理メモリを直接あつかうMS-DOS上のプログラミングは大変です(googleしてみて:-) (脚注2) 大昔(70年代末〜20世紀の間?)はメモリ容量も小さかったため、 巨大な仮想メモリが使えることに意義がありましたが、 **物理メモリから溢れた分はHDD,SSDに置いてアクセスするため、ものすごく遅く**なります。 いまどき物理を越える仮想メモリが必要なプログラムを走らせたら負けで、 必要なメモリをお金で調達=解決するべきです

ページング型仮想記憶システム

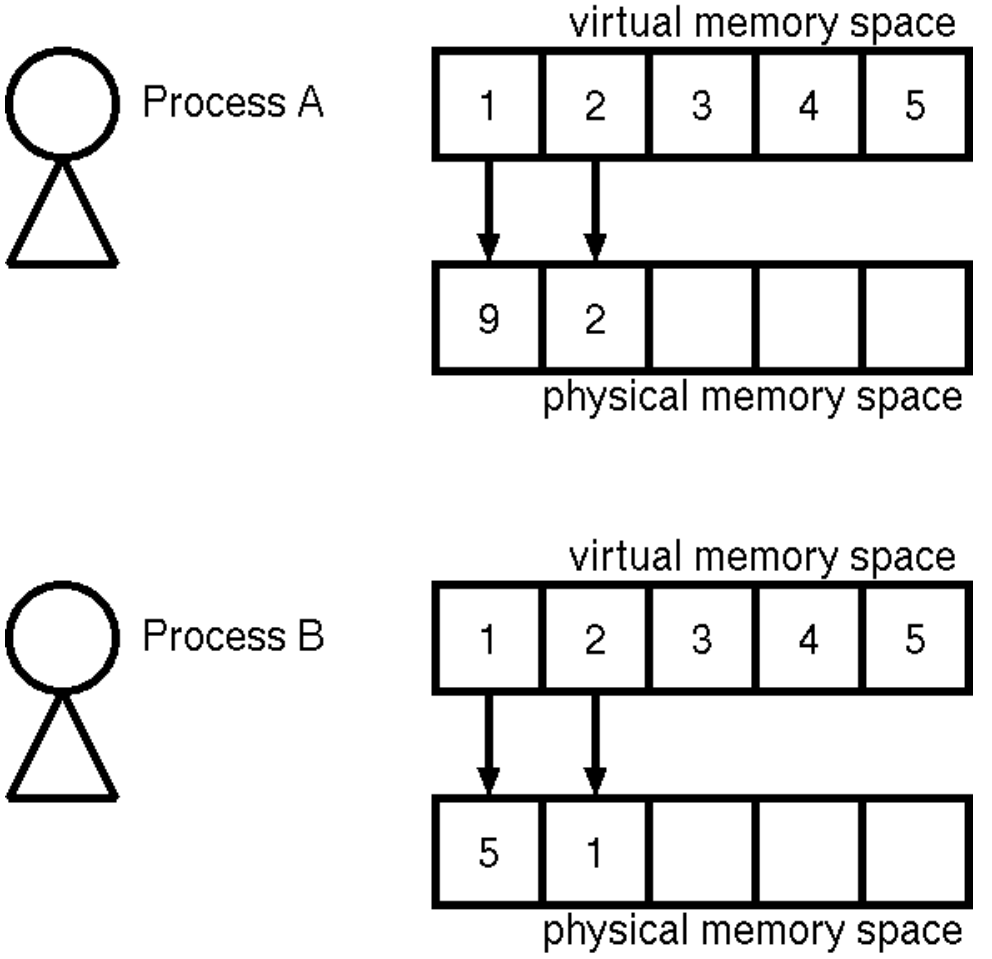
- アドレス変換はバイト単位ではなくKB単位の固定長で行います。 この処理単位を**ページ**と呼び、大きさは4KBか8KBあたりが定番です
- メモリをページで分割するタイプの仮想記憶が**ページング(paging)**型で、主流です
- **ページテーブル**(アドレス変換ルール表)とは、仮想->物理ページ番号の変換表です。 表の要素(変換ルール)を**PTE**(Page Table Entry)と呼んでいます
- HDD,SSDとメモリ間でスワップ(swap)します
  - **ページイン** ... ストレージからメモリに読みこむ
  - **ページアウト** ... メモリからストレージに書き戻す



(脚注) 実際にアクセスする場所は「アドレス = ページ番号x単位(4KBなど) + ページ内オフセット」です。 ページ番号だけを変換します

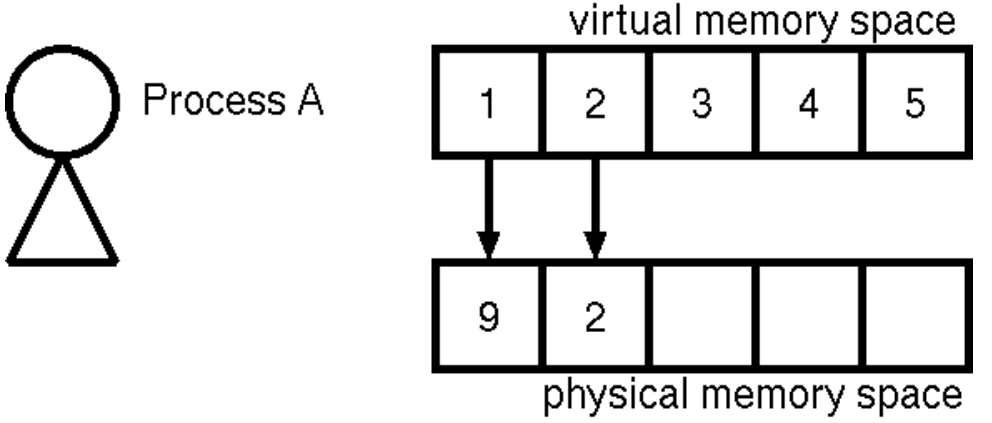
アドレス変換の様子

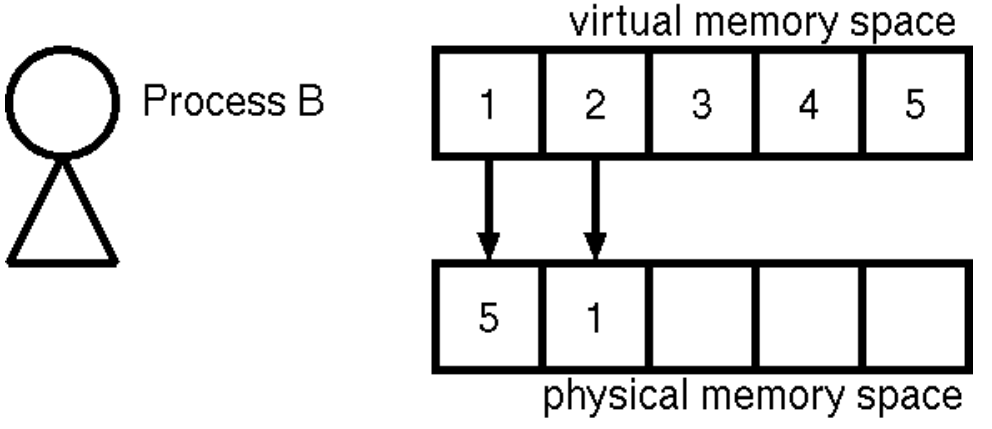
- 各プロセスは同じ(仮想)メモリ空間を使っていると**錯覚**しています
- 右図は、**プロセスAもBも仮想ページ１～５を使っています** が、**実際に読み書きする物理ページは異なります** 。例：
  - プロセスAの仮想ページ1 -> 物理ページの9
  - プロセスBの仮想ページ1 -> 物理ページの5
  - (以下同様なので省略)
- 物理メモリ上には複数のプロセスが共存します



ページング型仮想記憶システムのメリット、デメリット

- メリットは実装がシンプルなこと
  - 処理単位が固定長なため
  - メモリ空間を単純に固定長で分割し領域ごとの役割などは考えない
- デメリット
  - MMUが必要(ソフト処理は非現実的)
  - 小さなメモリ領域が必要な場合でも、 ページ単位より小さな処理はないため非効率なことがある





(脚注) 役割を考える方法も有り、 セキュリティ上よい実装が出来る可能性があります(詳細省略)

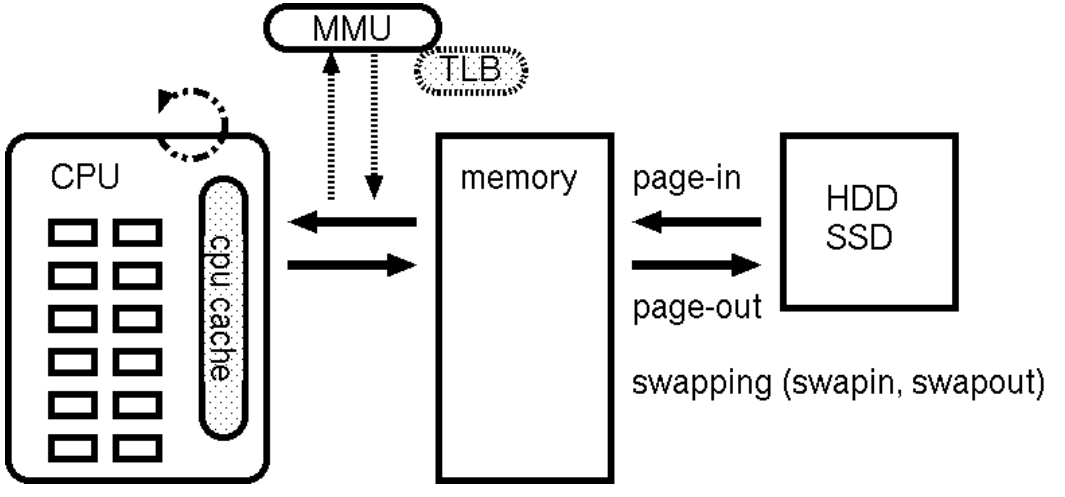
ページ置換アルゴリズム

- スワッピング（メモリとストレージ間でページを移動）しながら処理をしていますが、 メモリも大きくは無いため、 メモリに読みこもうとしても、すでにメモリに空きがないことがあります
- 空きページを作る際は、 **ページ置換アルゴリズム**に従い、ページを削除(or ストレージへ書き戻し)後、空いたページに必要なデータを読み込み(ページイン)します
- ページ置換アルゴリズムの代表例
  - **LRU (Least Recently Used)** ... もっとも長期間使われていないページを削除
  - **LFU (Least Frequently Used)** ... もっとも参照回数が少ないページを削除
  - **FIFO (First In First Out)** ... もっとも古いページを削除

(脚注1) これも基本情報処理試験の範囲なんですよね (脚注2) 余談ですが、ページ置換アルゴリズムを正しく実装するのは意外と大変で... なにしろPTEを参照する様子をだれかが見張っているの?(いえ見張って無いです:-)

参照の局所性とTLB(PTEのキャッシュ)(1)

- MMUはMMU自身が知らない(MMUのキャッシュにない)アドレス変換を要求された場合、 ページテーブルを検索し該当するPTEを読み込みます
  - カーネルがページテーブル(PTE)をメモリ上に準備しています
  - このテーブルサイズはかなり大きいので、メインメモリでないと置けません
  - 読み込んだPTEはMMUでキャッシュします
  - **同じPTEをしばらく使うことが多いのでキャッシュは有益**です
  - このキャッシュを**TLB**(Translation Lookaside Buffer)と呼びます

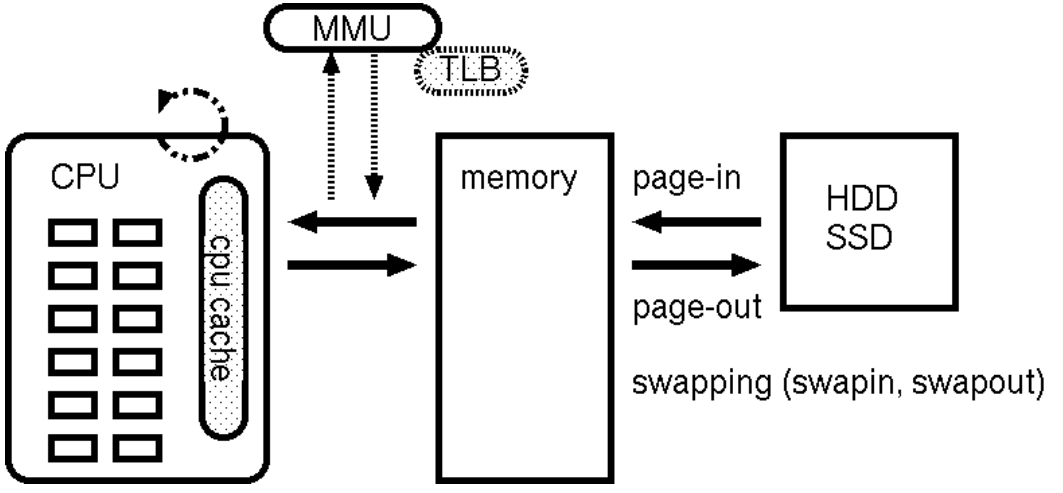


(脚注) TLB = PTEキャッシュ相当ですがTLBという特別な名称があるので覚えてください

参照の局所性とTLB(PTEのキャッシュ)(2)

- TLBのキャッシュヒットは仮想記憶の性能向上にとって重要ですが、 このキャッシュヒットの高さを期待できる理由が**参照の局所性(locality of reference)**です
- プログラムには**参照の局所性**(経験則)があります
- もっともわかりやすい例がループ(一部のメモリ領域だけを重点的に使う例)ですが、 プログラムには多かれ少なかれ参照の局所性があります

```
for(i=0;i<N;i++){ sum+=i;}
```



(脚注) TLB = PTEキャッシュ相当ですがTLBという特別な名称があるので覚えてください

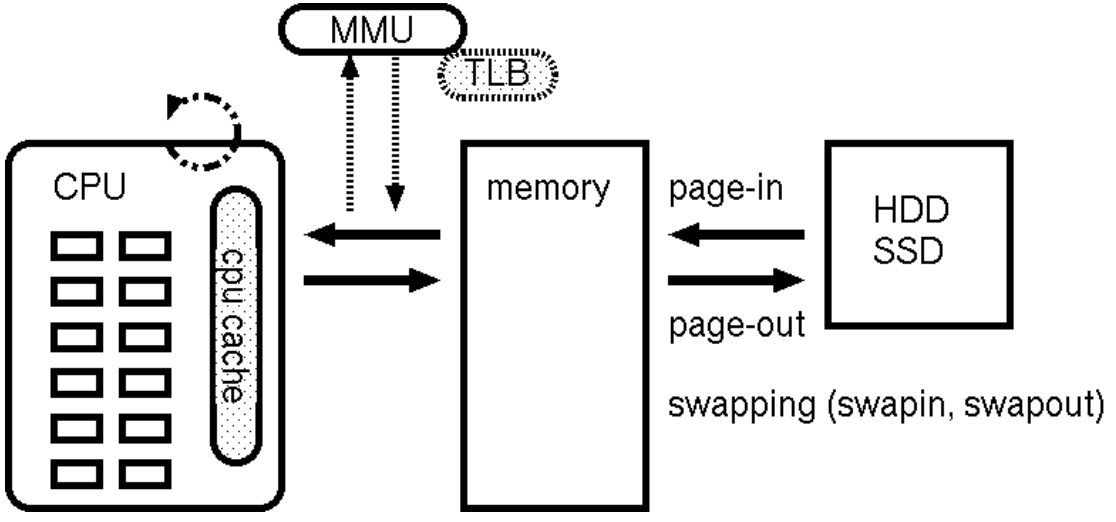
記憶装置(ハードウエア)の多段構造とキャッシュ

コンピュータにおけるキャッシュの重要性

- (初回にHDD,SSDから読み出す時は異なりますが)2回め以降、 読み書きは次のような多段構造

```
CPU <- レジスタ <- CPU cache <- memory <- HDD,SSD
CPU -> レジスタ -> CPU cache -> memory -> HDD,SSD
```

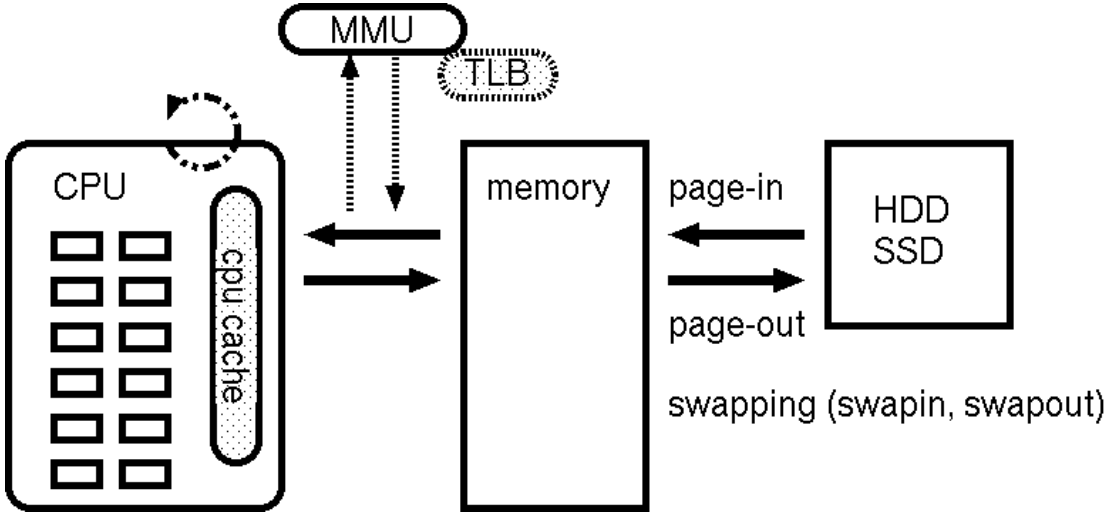
  - 左側へいくほど高速・低容量
  - CPU cacheも1次、2次、3次などと多段
- **より高速なキャッシュから必要なデータを読みこめれば、 それだけ処理速度が上がります(キャッシュヒット)**
- **デバイスごとの速度差が大きいため、キャッシュはコンピュータにとって重要**



(脚注) コンピュータネットワークでもキャッシュは重要です。 OSの速度にくらべてネットワークが遅すぎるため、 CDNなどのキャッシュサービスは通信の高速化に役立ちます

記憶装置(ハードウエア)の多段構造(1)

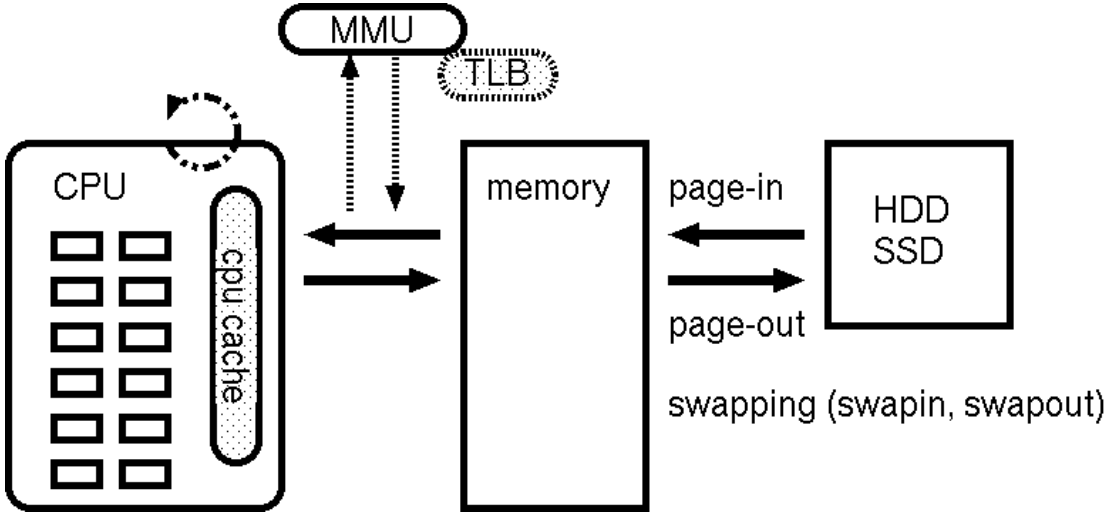
- プログラムやデータは、**安価で大容量ですが低速な補助記憶装置(HDD,SSD)**に格納していて、使うときに始めて補助記憶装置から**主記憶装置**(メインメモリ)に読み込みます
- CPUはメインメモリにあるデータを読みながら処理を進めますが、 メモリもCPUよりほぼ1桁低速なため、 一度読んだデータはCPU内の**キャッシュに格納し(遅い)メモリアクセスを減ら**します
- キャッシュは**SRAM**、メインメモリは**DRAM**というLSIです。 SRAMは高速・高価・低容量、 DRAMはSRAMにくらべ低速・安価・大容量。 使い分けが重要



(脚注) 書類のなかには、滅多にアクセスしないけれど捨ててはいけないデータ(Cold data)があります。 そういったデータをHDDに置いています。 クラウド(AWS)も考慮に入れれば、 処理結果 -> SSD -> HDD -> AWS S3 -> AWS S3 Deep Archiveへと移動させていきます

記憶装置(ハードウエア)の多段構造(2)

- プログラムやデータは、
  - 使うときに始めて補助記憶装置から**主記憶装置**(メインメモリ)に読み込みます
  - 補助記憶装置は通電していなくてもデータが消えない媒体です
- 製品
  - **安価で大容量ですが(CPUに比べれば)低速な補助記憶装置(HDD,SSD)**に格納しています
  - クラウドストレージも考慮に入れるべき時代
  - 最強のバックアップメディアは**磁気テープ**

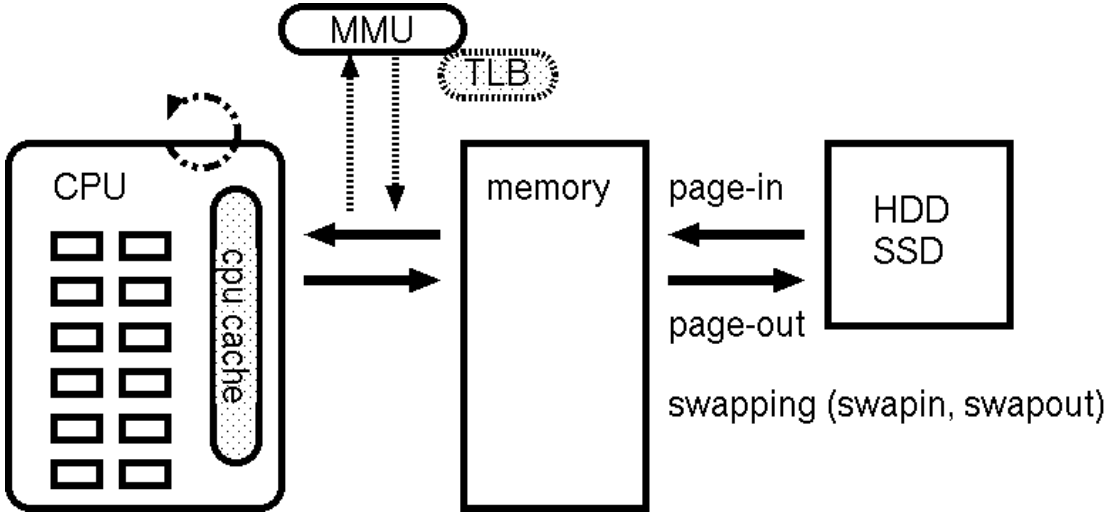


(脚注1) 書類のなかには、滅多にアクセスしないけれど捨ててはいけないデータ(Cold data)があります。 そういったデータをHDDに置いています。 クラウド(AWS)も考慮に入れれば、 処理結果 -> SSD -> HDD -> AWS S3 -> AWS S3 Deep Archiveへと移動させていきます

(脚注2) すでに磁気テープは日本（それも富士フィルムのみ?）でしか生産していません。 地味に世界中のクラウドを支えています

スワッピング(swapping)、スラッシング(slashing)

- メモリとHDD,SSD間の読み書きの単位は4KBなどの**固定長**、 この単位が**ページ**（復習）
- データをストレージからメモリへ読みこみ、(CPUで処理、メモリ上のデータを更新後)、 最終的に更新したデータをストレージへ書き戻すことも多いです。 このデータ交換が**スワッピング**です（復習）
- あまりにも**スワッピング**が多いと、 ストレージへの(遅い)入出力に引きずられてOS全体のパフォーマンスが低下します。 この状態が**スラッシング**で、 これを避ける設計や運用が重要です



(脚注1) そもそもスワップしたら負けなんですけどね;-) (脚注2) 本稿では4KBなどの固定長でswapする話をしますが、 16bit Unixではプロセス単位でswapするメモリ管理でした(swap-based system)。 どちらもswapという用語で紛らわしい (脚注3) HDDの処理単位(512B)がメモリの処理単位(4KB)と異なる場合もありえますが、 そういう差異はデバイス担当ソフトウエアなどで隠蔽します

(アーカイブ動画で見ている人も)一時停止して休憩

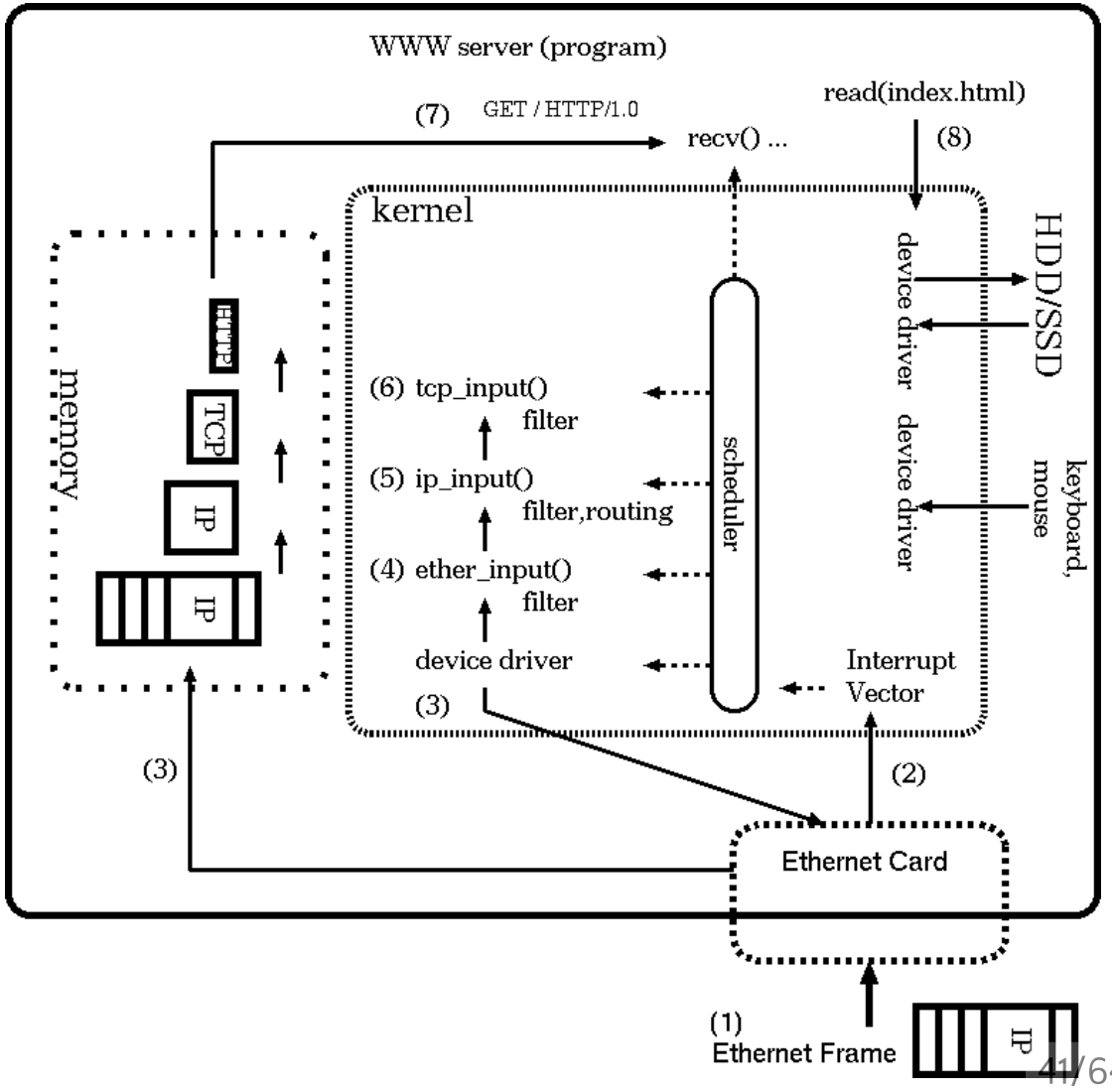
「30分ごとに雑談（休憩）をしろ」というのが指導教官の教え

正確には「アメリカでは30分ごとにジョークを言わないといけない」だけれど、そんな洒落乙なこと無理:-)

割り込み

シリーズ構成、サーバの全体像との関係

- シリーズ構成
  - 第05回
    - CPU
    - プロセス
    - メモリ（仮想記憶）
    - 割り込み
  - 第06回
    - 競合状態と排他制御
    - デバイスドライバ
    - ストレージ
- 今回（第05回）の最後の1/3
  - 図の(2)(7)(8)、 右側のデバイスドライバ（詳細は次回）のところもハードウエア割り込み



割り込み(interrupt)とは

カーネルに気づいてほしいイベントがある時に、それを知らせる処理が**割り込み**です

1. カーネルに気づいてほしい時とは？例：
    - エラーが発生しました
    - 下請け（のデバイス）さんから「依頼された仕事の終了」を連絡したい
    - （一般ユーザプロセスが）カーネルさんにしかできない仕事を依頼したい
2. **特権権限**が必要な仕事 … （一般プロセスに出来ない処理を）カーネルにしてもらいたい
3. 仕事が終わったか?を知る方法は2種類(**攻めvs受け**?)考えられます
    - 「処理が終わったら連絡して欲しい」という方法として**割り込み**を用います
      （次回の「同期と排他制御」でも少し話します）

(脚注) 攻め=こちらから定期的に終わりましたか？と尋ねまくる方法もありますが、 それは効率がよくない

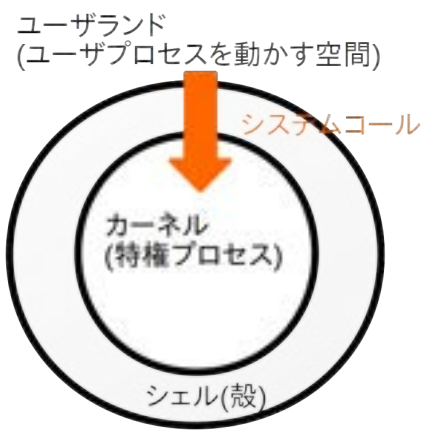
【復習】カーネルだけが特権処理を行います

- OSでは特権が必要な処理があります。典型例：**デバイスの操作**
- そもそも、ほぼどんな処理も一般プロセスはカーネルにお願いしている形です
  - 例：プログラムの実行、ファイルシステムの操作/設定、ファイルの入出力
- **カーネルだけが特権処理を実行できる**ように実装するには、 **CPUの支援**が必要です

(脚注1) 演習ではsuもしくはsudoコマンドで管理者になって行う操作をしています。 そういった**演習/実務**で行うシステム管理(例:ユーザ作成,パスワード変更)も**OS管理者だけが行う特権処理**ですが、 ほぼ**設定ファイルの編集権限の制御(Unixセキュリティ,ユーザ権限)**の話なので、 今回のテーマとは毛色が異なります (脚注2) だれでもストレージに書きこめたら重要なデータを好きに書き換えできてしまいます。 各種入出力も同様で、入出力に介入できたら嘘の情報を入力、出力できます (脚注3) かつてのMS-DOS時代の(昔の)パソコンのように、 ユーザの区別が概念がない(個人向け)OSもありましたが、 現代では、個人向け製品でも、きちんと権限の概念を持つべきでしょう (もちろん家電製品なら不要ということはありえます。 炊飯器や冷蔵庫には要らなそうです:-) (脚注4) カーネルが関係ないのは、実行中の数値計算プログラムくらいか？（もちろん結果の入出力はカーネル）

【復習】OSの階層セキュリティモデル

- 階層構造の**卵**もしくは**玉ねぎ**型モデル
  - 真ん中(卵の黄身)は特権が必要とされるゾーン
    (**カーネル**に相当する部分)
  - 境界線を通過できるのは特別な命令のみ
    - 卵の殻=**シェル**
    - 一般ユーザプロセスからカーネルへ依頼する仕組みが**システムコール**です
  - 殻の外はユーザアプリを動作させる世界
    (**ユーザランド**)、**特権処理は不可**です



(脚注) カーネルはスーパバイザ、システムコールはスーパバイザコールとも呼ばれます。 本稿ではUnixに合わせてカーネルとシステムコールという用語を使います。 仮想化で**ハイパーバイザ**という用語が出てきますが、 これはsuperより上位の動作なのでhyper:-)

実行を一時中断するイベントは3種類

- CPUは機械語を読みながら処理を続けます。 特に何もなければ機械語を読みつづけ、実行しつづけます。 これが通常状態です

- この通常処理を一時**中断**して**特別な処理に移るイベント**が**3種類**あります。

種類	別名	備考
システムコール	ソフトウエア割り込み	ユーザプロセスからカーネルへ依頼する方法
例外		0で割り算やハードウエアエラーなど
デバイスからの割り込み	ハードウエア割り込み	ハードウエアの制御

実行を一時中断するイベントの別分類(基本情報処理試験)

- 割り込みは原因により二種類に大別できます。
  - 実行中のプログラムが原因でおこるものが**内部割込み**
  - プログラムに関係なくおこるものが**外部割込み**

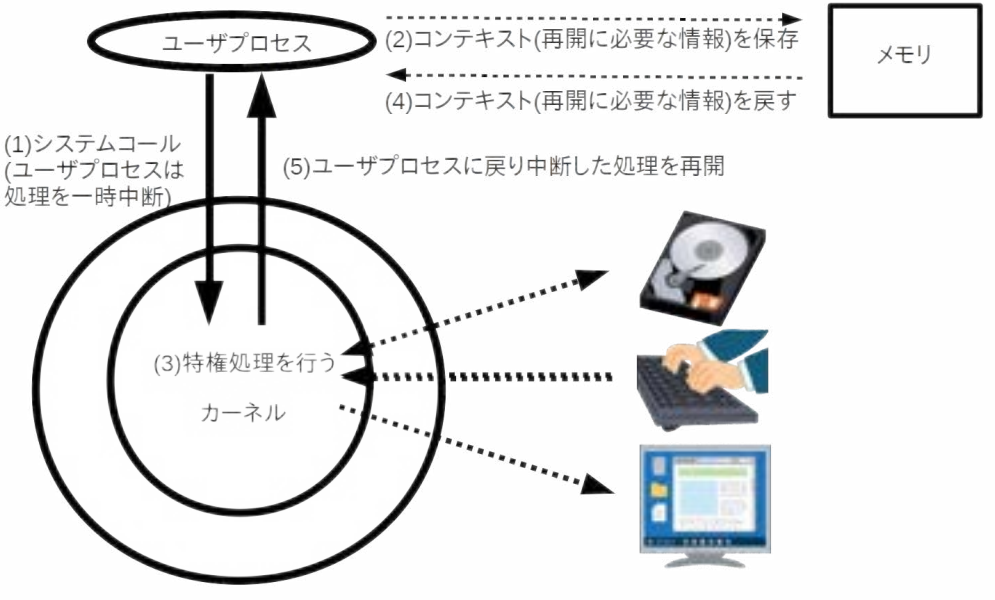
大分類	中分類	概要
内部割り込み	エラー	ゼロで割ったなど
	システムコール	ユーザプロセスからカーネルへ依頼する方法
外部割り込み	エラー	ハードウエア障害
	ハードウエア割り込み	ハードウエアの制御

(脚注1) 基本情報処理試験の分類では、こうなっています。 仕方ないので、この用語も覚えてください。 Unixカーネルのソースコードでは、こういう区別をした書き方をしていないので、 本稿では無視して前頁の用語のまますすめます:-)

(脚注2) 中分類は前頁とあわせていますが、 基本情報処理試験では、もっと細かい分類をした用語を覚える必要がある?

システムコール: 特権処理をカーネルにおねがいする

1. ユーザプロセスが**システムコール**を呼び出すと、CPUにより処理が一時中断されます
2. ユーザプロセスの情報(コンテキスト) をメモリに保存します
3. カーネルに制御が移り、カーネルが依頼された処理を行います
4. 保存していたユーザプロセスの情報を復元し、中断されたところから再開できるように準備
5. ユーザプロセスに戻り、中断されたところから処理を再開します



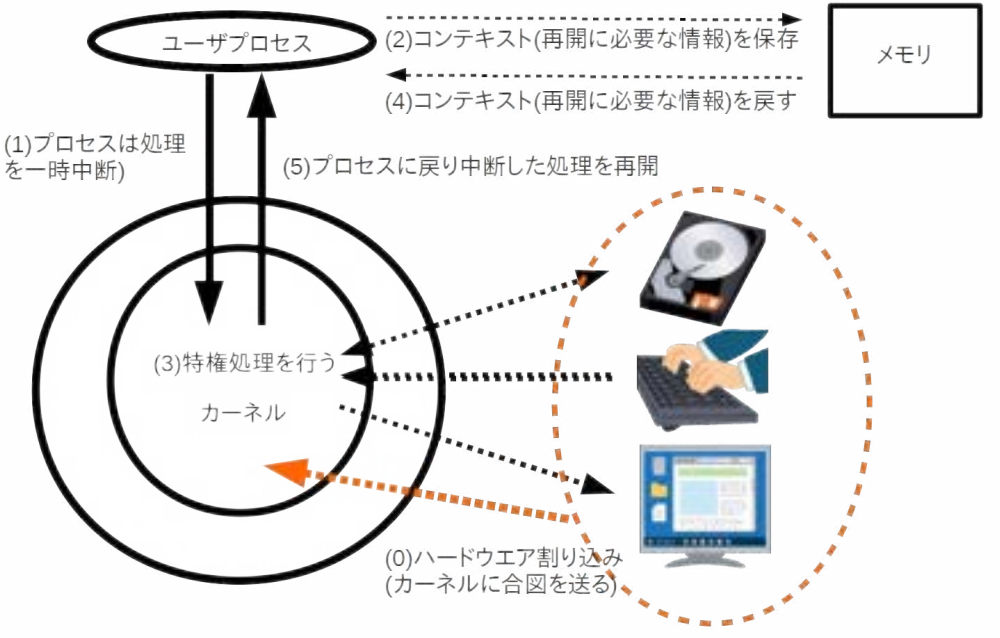
(脚注1) **コンテキスト切り替え(context switch)**は、 ユーザプロセスからカーネルプロセスへ切り替わる動作および逆の動作部分

(脚注2) 3の処理が長い場合(例:HDD/SSDへの読み書き)、 途中で4へ進み、他のプロセスへswitchします。 3の処理終了(その合図が次頁)後のいつか、依頼主へ戻る4-5が行われますが、 その時期は3の処理とスケジューラ次第 (解説を適度な長さにするため不正確)。

ハードウエア割り込み: デバイスからの合図で処理を中断

ハードウエアから割り込み=終了の合図が送られてきます (右図の0;オレンジ色の部分)

1. CPUにより処理が一時中断されます
2. ユーザプロセスの情報(コンテキスト) をメモリに保存します
3. カーネルに制御が移り処理を行います
   (例:HDD/SSDからの読み込み結果を返す準備)
4. 保存していた情報を復元して中断されたところから再開できるように準備
5. ユーザプロセスに戻り、中断されたところから処理を再開します



(脚注1) **処理のトリガーが異なるだけで、動作は前頁と同様です** (前頁はプロセスから自発的に依頼、本頁はハードウエアから合図)

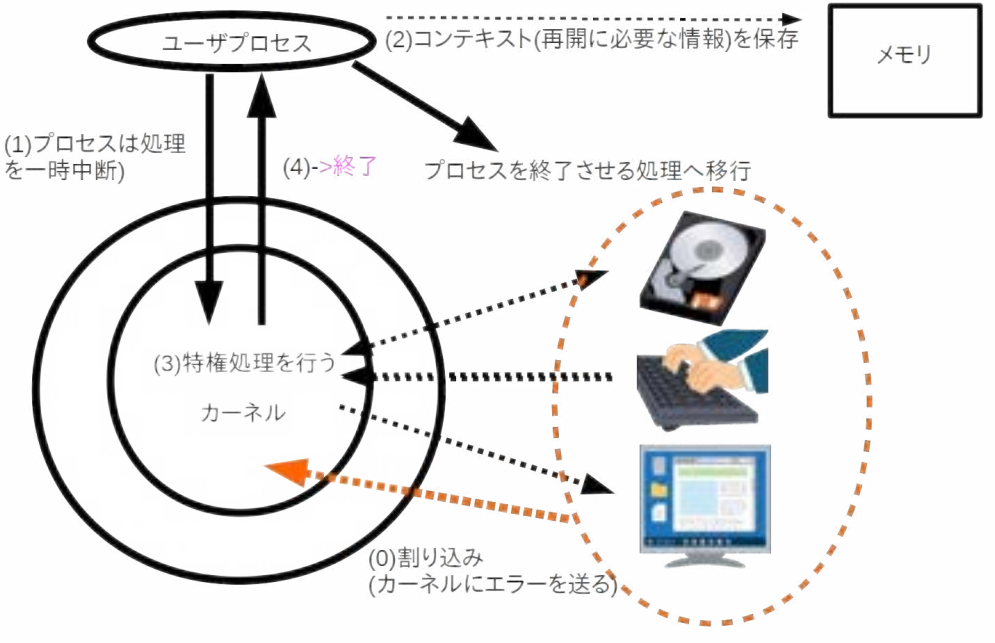
(脚注2) 図ではユーザプロセスですが、 割り込まれるプロセスはカーネル(が実行中の処理)のこともあります

(脚注3) 前頁の脚注にもあるように、 適度な長さに抑えるために正確さは犠牲 (モヤモヤする人は是非ソースコードを読もう！)

エラー: デバイスからの合図で処理を中断

エラーを知らせるために割込みをかけると

1. CPUにより現在実行中の処理が一時中断され
2. エラー情報とユーザプロセスの情報(コンテキスト) をメモリに保存します
3. カーネルに制御が移り、エラーに対応します。たいていは致命的エラーなので該当するプロセスは終了です (死亡印をつけておきます)
4. プロセスに戻る途中でプロセスは終了



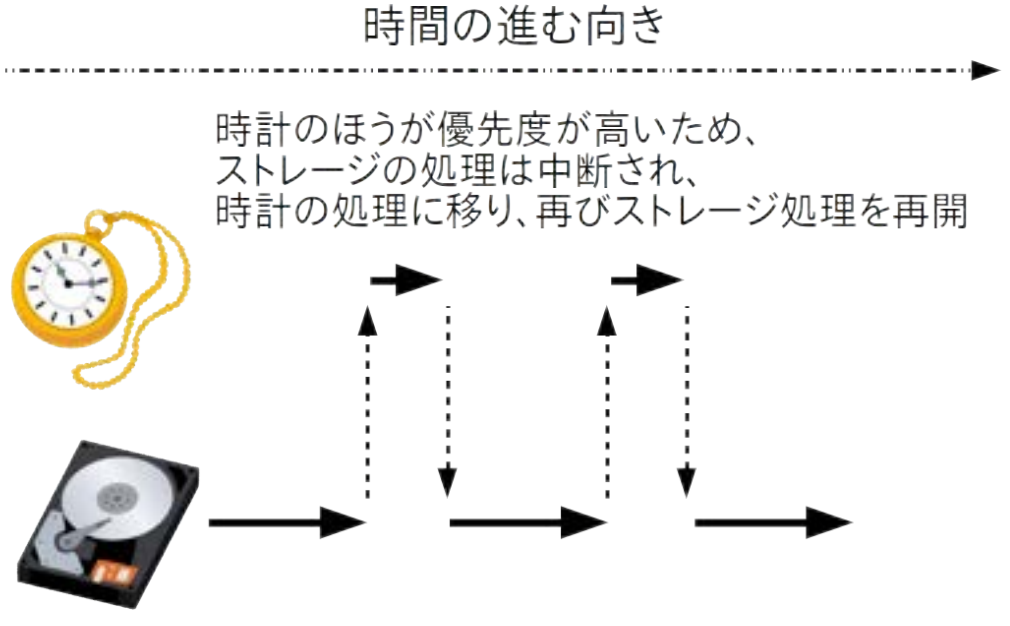
(脚注1) 前半は前頁と同様ですが、後半は異なり、 プロセスを終了させることが多いです (ハードウエアからのエラーは致命的なレベルが普通なので処理を再開することは少ない)。 プロセスはzombieになり、後で死神=reaperが回収します:-)。 Unix Kernelではreaperという用語を使ってないですが、 他のソフトウエアではreaperという表現を使うことがよくあります (19世紀なかば以降の用例のようです)

(脚注2) 他の用例: CHAPTER 37: THE REAPER WHOSE NAME IS DEATH - L.M.Montgomery, Anne of Green Gables (1908)

(脚注3) この図は、実行中に0の割算などの致命的エラーがおきたケースです。 ハード障害などでは、そのハードを使おうとしたプロセスが終了するわけですが、 それは図で中断されたプロセスとは限りません（前頁までと同様、不正確です）

割り込み優先度と多重割り込み

- 割り込み処理の途中で、別の割り込み処理をすることもよくあります
- 各割り込みに優先度を設定しています
  - 致命的なエラーは最高優先度で割り込むべき
  - 時計も優先度を高くしないと時間が狂います
  - 一方、ストレージやネットワークの読み書きはエラーや時計ほど優先されなくても大丈夫
- 右図は、 ストレージ(HDD)の処理をしている途中で、時計のハードウエア割り込みがかかり、優先度の高い時計の処理を一時おこなったあと再びストレージの処理にもどっている様子です



(脚注1) 多重割り込みくらいまでは基本情報処理試験に出るから、やっておきますね... (脚注2) 突然プロセスAが中断し、 カーネルが処理Bを始めたとおもえば、割り込み処理がかかって処理Cに移り、 処理Cを終えると、プロセスEにコンテキストスイッチし ... （いつAに戻るんだ？） と言う具合にTSSは動いています。 これがソースコードを読みながらイメージできないとOSは理解できません

【参考】レジスタとアセンブリ言語の具体例

(脚注) 【参考】部分は試験範囲外です

【参考】レジスタ: PDP11 (16bit CPU)

	レジスタ	主用途	備考
汎用レジスタ			
	r0	汎用	
	r1	汎用	
	r2	汎用	
	r3	汎用	
	r4	汎用	
	r5	環境ポインタ	Unixの慣例では環境ポインタ
	r6	スタックポインタ	spとも表記される
	r7	インストラクションポインタ	ipとも表記される
メモリ管理レジスタ	SR0,SR1,SR2,SR3		SR1,SR3は機種依存
管理レジスタ	PSW	ステータスなど	PSW = Processor Status Word

【参考】レジスタ: Intel 8086 (16bit CPU)その1

	レジスタ	主用途	備考
汎用レジスタ			
	AX	計算用	
	BX	ポインタ	
	CX	カウント	
	DX	一時記憶	
	SI	転送元	
	DI	転送先	
	BP	ベースポインタ	
	SP	スタックポインタ	
	IP	インストラクションポインタ	
flags	ステータスなど		

【参考】レジスタ: Intel 8086 (16bit CPU)その2

	レジスタ	主用途	備考
セグメンテーションレジスタ			
	CS	コード	
	DS	データ	
	ES	エキストラ	
	SS	スタック	

- Intel CPUは、もともとセグメンテーションという方式を使っていたため、 アドレス(ポインタ)はセグメントとセグメント内の位置(オフセット)をセットで指定します

```
セグメントアドレス:オフセットアドレス
30A4:0100 -> 30A40 + 0100 = 30B40 が実際のアドレス
```

- この方式は、 低機能なCPU上で低コストに大きなメモリを扱うには便利だったようです。 低コストとは「一度セグメントを決めればオフセットだけで処理可能=アドレス用のメモリも小さくてOK」だから

(脚注) Intelって元々単なるメモリ屋さんだったからPCという思想に興味がなかったんでは?

【参考】Intel CPUでループ(C言語)

```c
#include <stdio.h>

int main (int argc, char **argv) {
  int i, sum;

  sum = 0;
  for (i = 0; i < 10; i++) {
    sum += i;
  }

  printf("sum = %d\n", sum);
}
```

- 単純な足し算をするループです。コンパイルするとCPUが理解できる**機械語**になります
- 次頁では、 このループ本体部分の機械語を**アセンブリ言語に逆変換**(disassemble)したものと、 その各行の解説をしています(CPUはIntelです。 本当は64bit環境なので少し違うのですが、 前頁にあわせて、アセンブリ言語と解説は16bit風になっています)

【参考】Intel CPUでループ(main部分のアセンブリ言語)

objdumpしてmain部分を抜粋(+最初の数行と最後の2行を省略して一頁に)したもの
注: -0x8(%bp)は省略記法,C言語風に書けば「bpポインタに対して *(bp-8) 」相当;mov命令はmoveですが実質コピーです

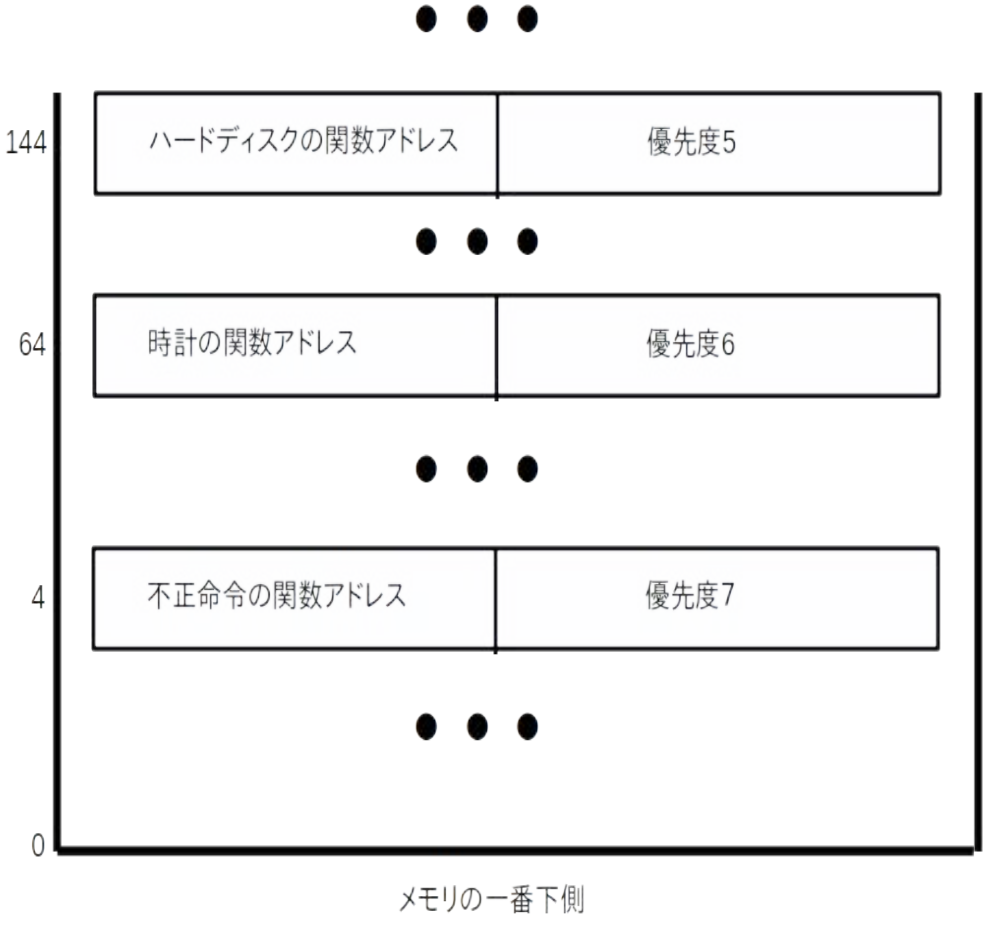
機械語	アセンブリ言語		解説
c7 45 f8 00 00 00 00	mov	$0x0,-0x8(%bp)	-0x8(%bp)を0に初期化(変数sumとして利用)
c7 45 fc 00 00 00 00	mov	$0x0,-0x4(%bp)	-0x4(%bp)を0に初期化(変数iとして利用)
eb 0a	jmp	4行後のアドレス	forループの条件部分へジャンプ(4行先のcmp行)
8b 45 fc	mov	-0x4(%bp),%ax	axレジスタに変数iを代入
01 45 f8	add	%ax,-0x8(%bp)	axレジスタの値を変数sumに加算(sum+=i相当)
83 45 fc 01	add	$0x1,-0x4(%bp)	変数iに1を加算(i++相当)
83 7d fc 09	cmp	$0x9,-0x4(%bp)	比較 i<=9 をし、結果をフラグレジスタに保存
7e f0	jle	4行前のアドレス	フラグレジスタを見て結果がtrueなら4行前へ戻る(ループ)
8b 45 f8	mov	-0x8(%bp),%ax	変数sumの値をaxレジスタへコピー
89 c6	mov	%ax,%si	siレジスタにaxレジスタの値をコピー(変数sum)
bf ce 09 40 00	mov	$0x4009ce,%di	diレジスタにformat文の文字列アドレスをコピー
b8 00 00 00 00	mov	$0x0,%ax	axレジスタを0に設定
e8 a3 fb ff ff	call	printfのアドレス	printf関数(引数はdi,siに設定済)を呼び出す
b8 00 00 00 00	mov	$0x0,%ax	axレジスタを0に設定(関数の返り値の設定)

【参考】割り込みとメモリレイアウト

メモリレイアウト例と割り込みの関係を解説

【参考】メモリ最下部にある割り込み設定の例

- DEC PDP11上のUnix第6版(16ビットCPU)における割り込み設定の例です。 縦はアドレスの番地(8進数)です。 横幅は32ビットで、16ビットの設定を2つ並べて書いています。
- メモリの一番下に、 アドレス0番地から上に向かい、 割り込みベクタの定義が並んでいます。
- たとえば時計(水晶振動子)からのハードウエア割り込みの場合、 アドレスの64番地から設定を読み込むようにハードウエアを作りこみます。 読み込むのは「時計の関数アドレス」と「優先度6」という設定で、 これらをCPUに設定します(そのような動作を自動的にするようCPUが作られています)



【参考】割り込みベクタの例

【参考】PDP11 割り込みベクタ

ベクタ位置	周辺デバイス	割り込み優先度	プロセス優先度
060	テレタイプ入力	4	4
064	テレタイプ出力	4	4
070	紙テープ出力	4	4
074	紙テープ出力	4	4
100	クロック(電源周波数)	6	6
104	クロック(プログラマブル)	6	6
200	ラインプリンタ	4	4
220	RKディクドライブ	5	5

【参考】PDP11 トラップベクタの例

ベクタ位置	トラップ種別	プロセス優先度
004	バスタイムアウト	7
010	不正命令	7
014	bpt トレース	7
020	iot	7
024	電源異常	7
030	エミュレータトラップ命令	7
034	トラップ命令	7
…	…	…
114	11/70 パリティ	7
240	プログラムされた割込み	7
244	浮動小数点エラー	7
250	セグメンテーション違反	7

【参考】Intel 8086 割り込みベクタ

周辺デバイス	割り込み番号
タイマー	0x08
キーボード	0x09
RS-232C	0x0B
RS-232C	0x0C
フロッピーディスク	0x0E

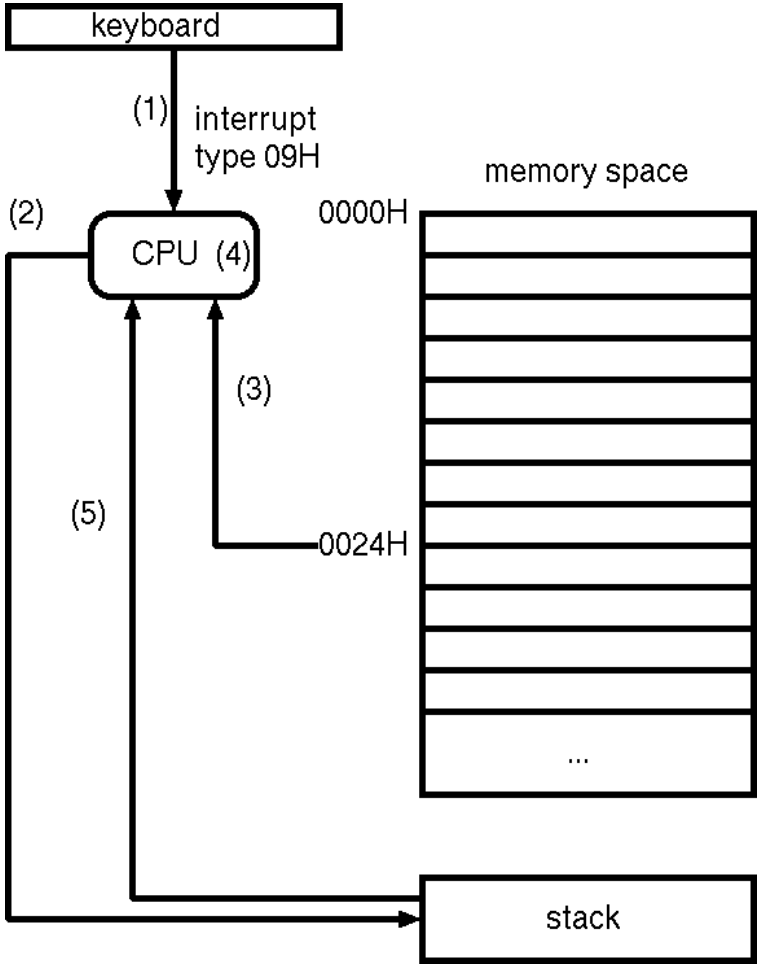
【参考】NEC PC-9800 割り込み番号の例

周辺デバイス	割り込み番号
タイマー	0x08
キーボード	0x09
CRTC	0x0A
RS-232C	0x0C
フロッピーディスク	0x13

【参考】8086 リアルモードにおける割り込み(キーボードの例)

- 割り込みベクタはメモリ空間の先頭1024バイトに書かれており、 タイプひとつごとに4 バイト(2 バイトの offset と 2 バイトの CS)を使用。 よってロードするベクタは「4*(タイプ-1)」の位置

1. 割り込みタイプ09Hがかかる
2. CS,IP,flags をスタックへ退避
3. 0024Hからタイプ09Hの割り込みベクタをロードする
4. ロードしたCS IPを実行する
   (キーボードの関数を呼び出して処理)
5. スタックへ退避しておいたCS,IP,flagsを元に戻す



情報技術応用特論 第06回
OS(3): ハードウエアの制御

競合状態と排他制御

競合状態とは?

- 複数のプロセスがリソース(資源)を取り合う状態
- スーパーの入場人数制限(競合状態の好例,右図)
  - **最大N人**しか入れません。 入口に**N個のカゴ**があり、 カゴがないときは入れません
  - N+1人め以降の客は入り口で待ちます
  - 客が一人でれば一人はいれるようになるので、 待っている客に「入れるようになった」と知らせます
- これをモデル化したものが**セマフォ(semaphore)**(ダイクストラ,1965)

(脚注1) N+1番目に入れる客は? -> （スーパーならFIFOでいいのだろうけれど） OSの場合一般には並んだ順とは限りません。 一時休止しているプロセスを起こすところが複雑 (脚注2) セマフォの詳細は省略;基本情報より難しい試験には出ます e.g. P操作,V操作

一つのリソースを排他制御したい(例: 在庫処理)

- 実際にプログラムを書こうとすると**排他制御**が必要になります
  - 「一つのリソースを取り合う」状態をなんとかするということです
- ショッピングサイトの在庫管理が良い例です
  - 多数のユーザが同時にカートを操作し、 ほぼ同時に購入ボタンをクリックする可能性があります
  - カートにいれた注文の「確定」とは、 以下の擬似コード(関数「確定」)のように在庫(変数)の値を減らす処理です。 この関数よびだしが**同時に複数おこわれた**時も正しく動くか？が論点です

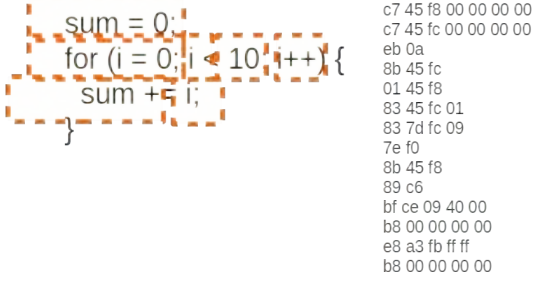
```
関数 確定（商品）{
    商品の在庫を -1 する
    // 在庫 -1 に成功できれば、このお客さんの商品を確保できている
    // このお客さんの購入品としてデータベースをアップデートなりする...
}
```

間違ったコード: 在庫を -1 するとは？

- 素直に考えると次のような擬似コードを書きそうですが、これは間違いです

```
if (操作してますフラグ == 0) {      // 誰も操作してないなら
    操作してますフラグ = 1          // 自分が操作します！
    商品の在庫数を取得
    商品の在庫を -1 する
    操作してますフラグ = 0          // 操作終わりました！
}
```

- なぜなら、どこでコンテキスト切り替えするか分からない。 プログラムの途中で中断/再開しうるから



(脚注) この例の「操作してますフラグ」は（複数のプロセスにまたがる）グローバル変数のイメージで読んでください

間違ったコード: こう動く可能性があります

```
A  if (操作してますフラグ == 0) {   // プロセスAは操作してOKと思っている
B  if (操作してますフラグ == 0) {   // プロセスBも操作してOKと思っている
B      操作してますフラグ = 1       // 自分Bが操作します！
B      Bが在庫数を取得              // たとえば在庫数 = 100
A      操作してますフラグ = 1       // 自分Aが操作します！
A      Aが在庫数を取得              // Aにも在庫数 = 100 と見えてしまう
A      商品の在庫を -1 する         // Aが在庫数を 99 に設定
B      商品の在庫を -1 する         // Bも在庫数を 99 に設定
A      操作してますフラグ = 0       // 操作終わりました！
B      操作してますフラグ = 0       // 操作終わりました！
```

- プロセスAとBが同時に注文しているとすると、 以下のように、 最終的に在庫数が-2(して在庫数98)にならないといけないのに-1(在庫数99)にしかならないことがありえます
- どこでプロセスが切り替わるかわからない(前頁を参照)ため、
  フラグのチェックと更新タイミングが絶妙に入り乱れた結果、 間違いになっています

(脚注) 何が悪いのでしょうか？各自、少し考えてみてください。 そして次頁の正解と比べてみてください

正しい動作: これを防ぐには?

```
A*  if (操作してますフラグ == 0) {  // プロセスAは操作してOKと思っている
A*    操作してますフラグ = 1        // 自分Aが操作します！
B   if (操作してますフラグ == 0) {  // Aが操作中Bは操作できない、のちほど再挑戦
A     Aが在庫数を取得               // 在庫数 = 100 を取得
A     商品の在庫を -1 する          // 在庫数を 99 に設定
A     操作してますフラグ = 0        // Aは操作終わりました！
B* if (操作してますフラグ == 0) {   // プロセスBが再挑戦、操作できるようになった
B*    操作してますフラグ = 1        // 自分Bが操作します！
B     Bが在庫数を取得               // 在庫数 = 99
B     商品の在庫を -1 する          // 在庫数を 98 に設定
B     操作してますフラグ = 0        // Bは操作終わりました！
```

- (a)「条件:だれも操作してない?」の確認と (b)「自分が操作します(フラグを1に設定)」
  (擬似コードで * がついている2行)を中断されることなく実行できる保証があればOK
- CPUは「同時に二つの動作(条件文と何らかの値の操作)をする」機械語を提供します。

  これを使った**排他制御**の機能をカーネルが提供しているので「これを利用する」が正解です

排他制御のコード例

(脚注) 本節はカーネル内の実装の話です、ユーザランドのロックの使い方が別途ありますが省略

ロック

- 大昔はgiant lock (ジャイアントロック)、現代ではmutex(ミューテックス)を使います
- ジャイアントロックの典型は「割り込みを禁止」する実装です
  - 多重割り込み(割り込み優先度)の応用です
  - 割り込み優先度を変更し「他の割りこみが出来ない」ようにするのが基本原理です
  - メリット：**シンプルな実装で分かりやすい**こと
  - デメリット： 各種**割り込みができない=待ち時間などを有効に使えない**ため、 OS全体の性能に影響をおよぼします。 とくにハードウエアの性能があがるとCPUが遊んでいる時間が増えてしまいます
- mutexを使い粒度をあげます
  - ロックしたい対象ごとのロック変数に対して操作します -> 粒度があがります
  - できるだけ細かい処理単位で書くことが大事で、「ある処理を使うフラグをoff/onする」だけのコードを書くことが原則です (実例は後述の条件変数のところで)

擬似コード「ジャイアントロック」（例:HDDへの読み書き）

```
関数 HDDへの読み書き（HDD）{ // もっとも大味なジャイアントロック版のイメージ
    割り込み優先度をあげる      // 他の割り込みをうけつけない
    HDDを読み書きする           // HDDの読み書きだけが（排他的に）実行され続ける
    割り込み優先度を下げる      // 他の割り込みの受け付けを再開
}
```

- ジャイアントロックでは**割り込まれません**
  - この例ではHDDの読み書きだけを実行しつづけます
  - 下手をすると数十秒この処理だけを実行しているでしょう
  - その間、短時間の処理（例：時計をすすめる、キーボードの入力）も受け付けません
    - 非効率だし、レスポンスが非常に悪い挙動になります

(脚注1) 非効率? = HDDの読み書きは待ち時間も長いので、その間は他の仕事もできるのに... (脚注2) 長時間処理の例として分かりやすそうなのでHDDを例にしていますが、 実際のTSSならHDDの割り込み優先度を最高に設定などしません。 時計は最高優先度にするでしょうし、 HDDよりもキーボードなど「人間が操作するもので短時間処理のもの」の優先度を高くします

擬似コード「ミューテックス」（例:HDDへの読み書き）

```
関数 HDDへの読み書き（HDD）{ // mutex(を使うけれど単純化された大味)版のイメージ
   mutexの処理(「HDDを使ってます変数」だけをロック,他の仕事も平行処理が可能)
   HDDを読み書き
   mutexのロックを開放
}
```

- 「HDDを使ってます」変数をロックして、「HDDを読み書き」を排他的に処理します
- HDDの読み書きと(擬似)並行で他の処理が実行できます
  - 何が平行実行されるか？はスケジューラ次第ですけど

(脚注) イメージ優先で、このあたりは端折っています。 カーネル内の話なので、 この文脈で競合しているコードは、 各プロセスのカーネルモードで動作している部分とか、 LWP（LightWeight Process / カーネルスレッド）。 コードを読まないと分からないのでは...

もう少しモダンな書き方（テクニック）

- OS全体の利用効率や反応速度を上げるには、競合状態をさばく**粒度**を上げる必要があります
  - いろいろ細かいことを考えていけば、どんどん粒度があがりますが、
  - 緻密なコードの書き方が要求されるようになり、 ミスをさそいがちです
- 自力で書くのではなくOSの提供する機能を使うべきです
  - 次頁は代表例の**条件変数(CV: condition variable)**です
  - **mutexと条件変数をコンビで使います**

(脚注1) 粒度 = granuality (脚注2) giant lockのような「他の処理をうけつけない」部分を減らさないとイケません。 そうしないと、 プリエンプションされてコンテキスト切り替えしても意味が無い。 たとえば、切り替えてもデバイスの処理が終わるまで待つ（=休眠するしかない）状態になります (脚注3) 解決策にはコンパイラがサポートする案もあります (脚注4) CV + mutexは、いわゆる鉄板パターンで、開発者がミスをしないようにしてくれます。 定番の細かい手順のいくつかをまとめて書きやすくしたものと考えてよいでしょう

もう少しモダンなコード例：Mutex + CV（条件変数）

```
1   mutexでロックをかける
2   while (リソースを使っていますフラグ == 1) { // (mutexロック下で)使用中か?を確認
3       cv_wait(条件変数cv, mutex変数)          // 条件変数cvで待つ(待つ間は休眠)
4   } // 注: whileを抜ける時はリソースが使えるようになった時, そしてmutexを再ロック済
5   リソースを使っていますフラグ = 1            // (mutexロック下で)フラグ=1
6   mutexのロックを外す
7   リソースを使う                              // デバイスドライバの処理本体を書くところ
8   mutexでロックをかける
9   リソースを使っていますフラグ = 0            // (mutexロック下で)フラグ=0
10  条件変数cvで待っているプロセスを起こす      // 次に処理できるプロセスはスケジューラしだい
11  mutexのロックを外す
```

- デバイスドライバ（後述）では上記のように書きます。「リソース」はHDDやキーボードなど各デバイスのこと

(脚注1) 詳細は分からなくてもOK (脚注2) **3行目で休眠**するところがポイント！ **cv_wait()はmutexのロックをはずしてから休眠し、復帰する際にmutexを再度ロック**します。 休眠中おなじmutexを使う他の処理は実行可 (脚注3) フラグの操作だけをmutexでロック

(アーカイブ動画で見ている人も)一時停止して休憩

「30分ごとに雑談（休憩）をしろ」というのが指導教官の教え

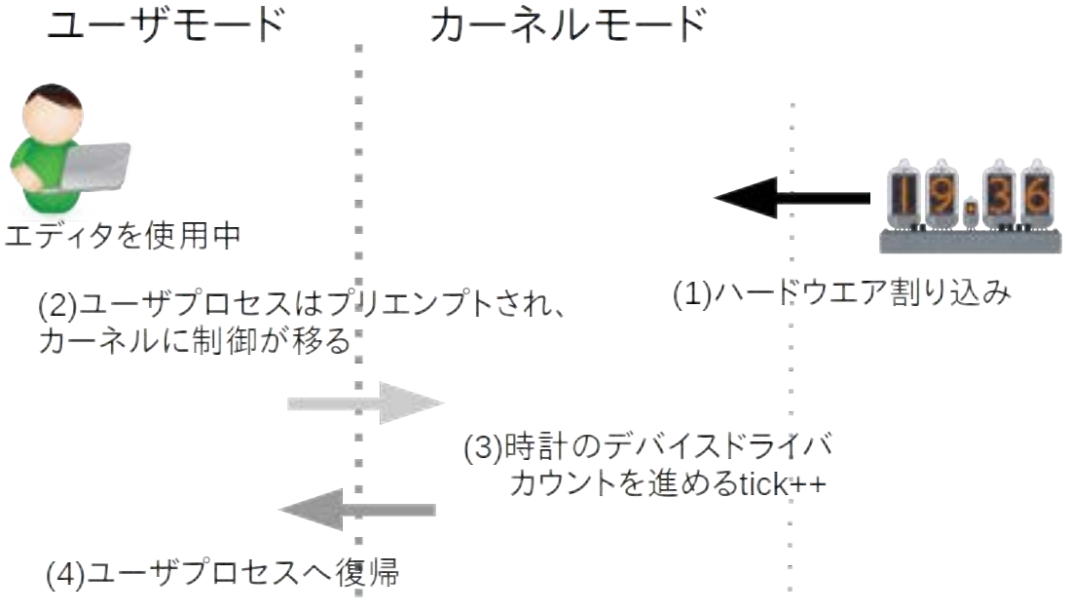
正確には「アメリカでは30分ごとにジョークを言わないといけない」だけれど、そんな洒落乙なこと無理:-)

デバイスドライバ

(脚注) 前節「競合状態」の続き(「競合状態」で使うテクニックの説明後、やっとデバドラの話ができます)

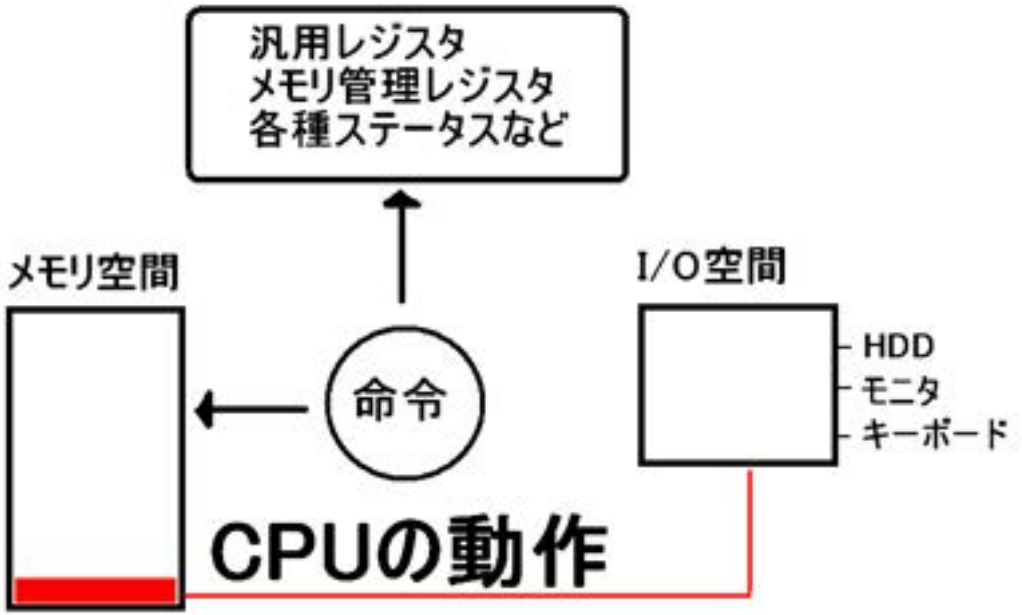
例: 時計の動作

1. クロックを刻んでいる回路からハードウエア割り込み (例: 1/100秒に一回)
2. 実行中のユーザプロセスはプリエンプトされ一時中断、カーネルへ移行
3. 時計のデバドラを実行
    - カウント(tick変数)を +1 する
4. ユーザプロセスへ復帰

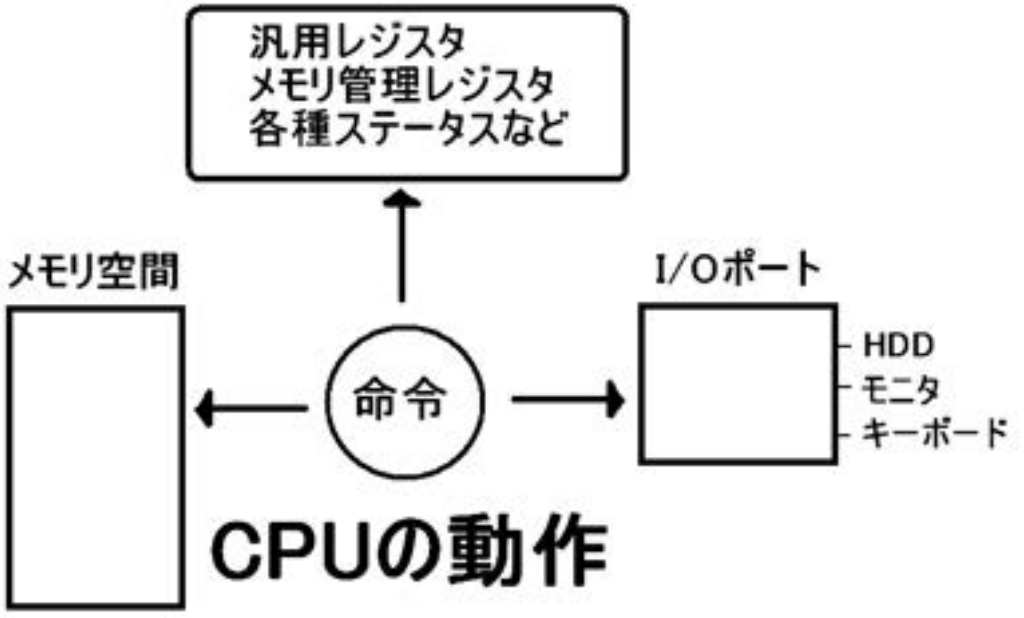


(脚注1) いろいろはしょってます (脚注2) コンピュータの時計は内部で「起動時の時刻 + 1/100 * tick」で推定しています。 当然すこしずつ狂うので、 Network Time Protocolで微修正をかけつつtickカウントをとっています (脚注3) 3.ではその他の処理をしてもok。 UNIX第6版では、 プロセス優先度の再計算や、 Nミリ秒後に実行予約された処理がないか?の確認などをしています (脚注4) カーネルが他の処理を実行中にクロックからの割り込みをうけることもありえます。 その場合カーネルが一時中断->時計の処理->元の処理へ復帰

CPUとデバイスの接続形態の代表例（0）



memory mapped

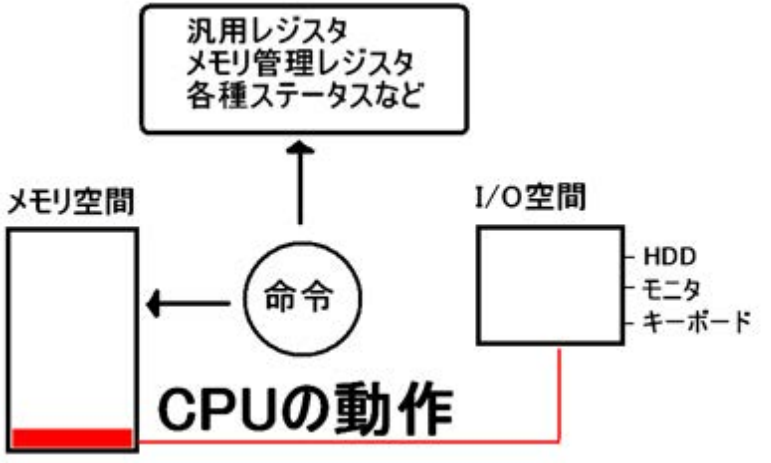


IO mapped

(脚注) 詳細は次頁以降

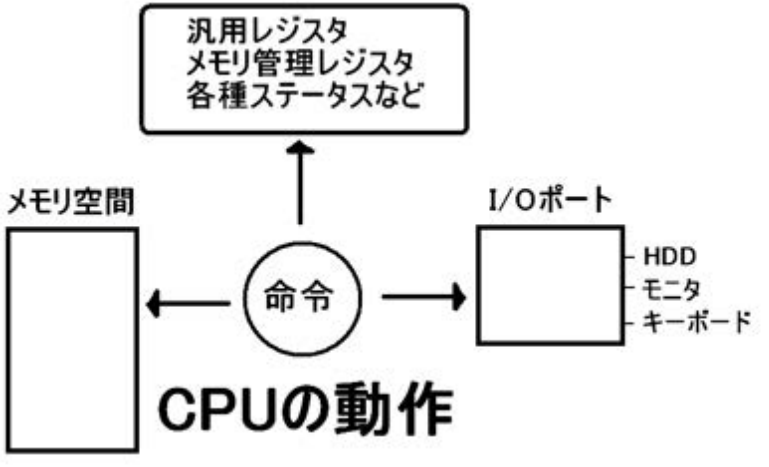
CPUとデバイスの接続形態の代表例（1）: memory mapped

- メモリ上にデバイスアクセス用の場所
  - 例:キーボードは0x220
- ポインタ操作でデバイスが制御できるため、デバイスだけ特別あつかいをしない統一感のあるコードの書き方ができて綺麗
- 機種例
  - DEC PDP-11（UNIXのターゲットマシン）
  - 最近のCPUほぼ全部



CPUとデバイスの接続形態の代表例（2）: IO mapped

- IO mappedもしくは port mapped
- メモリは全てプログラム用、デバイスは別枠
- 実装例：Intel CPU
  - IntelならIOポートというメモリ空間とは別にある回路を経由してデバイスへアクセス

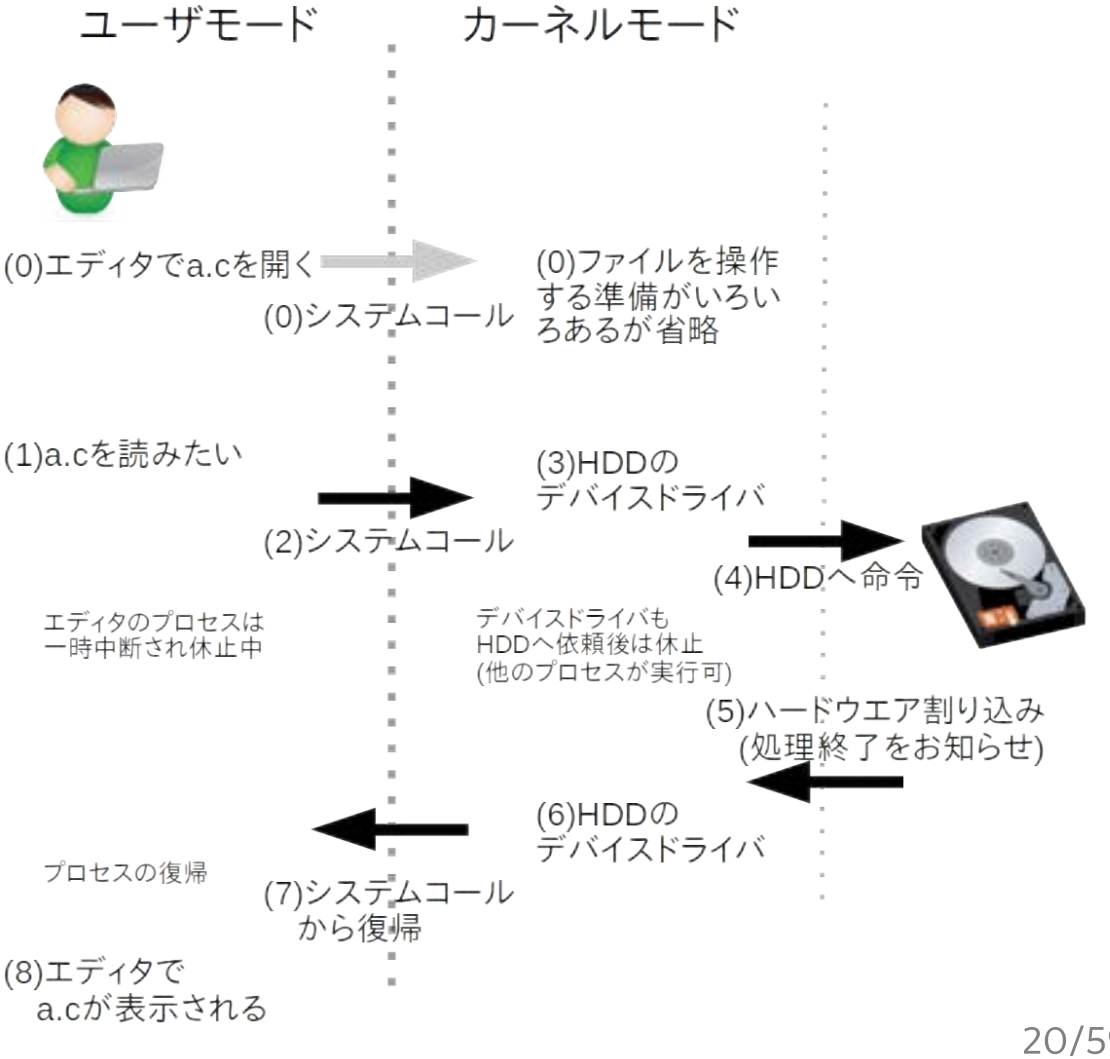


(脚注1) 当時のIntelの設計思想としては「PCはメモリが小さいためmemory-mappedでは**もったいない**ためらしい」という噂あり

(脚注2) 互換性のため、 現代のIntelは多分memory mappedとIO mapped両方とも使っていると思う。 もう中身グチャグチャ

HDDへの読み書きの全体像(半分は復習)

1. エディタでa.cを読みたい(と命令)
2. **システムコール**をかける(カーネルモードへ移行、エディタは一時中断)
3. HDDの**デバイスドライバ**(以下デバドラ)
4. デバドラからHDDへ命令を送り、処理(READ)が終わるまでデバドラは休止
5. HDDが依頼された処理を終えると**ハードウエア割り込み**をかけてカーネルに合図
6. デバドラに戻り、 kernelは読みこんだa.cの中身をエディタに返す準備をする
7. システムコールから復帰(ユーザモードへ戻り、エディタプロセス再開)
8. エディタでa.cが見えるようになる



(脚注) いろいろはしょってます

HDDデバイスドライバの擬似コード例(前頁の4.部分)

- デバドラの詳細はハードウエアごとに大きく異なります。 これはDEC PDP-11のRK05というHDDの例でmemory mappedです。コマンドを書き込んだ瞬間に動作が始まります

```
ポインタ *p; // 2バイト幅の変数へのポインタなので、p--すると2バイト減少することに注意

ポインタpをHDDを制御できるアドレス(65280〜65291番地間)の一番上(65290)に合わせる
*p-- = HDD上に書き込む場所の指定(セクタの番号)
*p-- = メモリ上のバッファのアドレス
*p-- = 読み書きするバイト数
*p   = コマンド

// memory mappedの例: メモリ上の配置は以下のとおりで、アドレスは十進数表記に変更済
65290   HDD上の場所の指定(セクタの番号)
65288   メモリ上のバッファのアドレス(読み書きする場所、読み書きの方向はコマンド依存)
65286   読み書きするバイト数
65284   コマンド（読み/書きの指定)
65282   デバイス側でエラーの理由を書き込む場所, READ ONLY
65280   HDDのステータス(Drive Status Register, READ ONLY)
```

デバイス
～HDD,SSD,磁気テープ～

(脚注) せっかく物理な話に近づいたので、このまま物理デバイスを概観してみましょう

デバイス: HDD, SSD

2.5インチのSATA HDD(左)と、 M.2 SSD(右側の銀色ヒートシンク下の板,よく見えないけど;-)

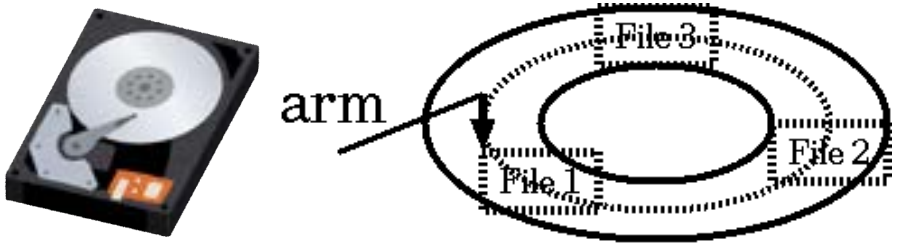
2.5インチのSSD(端子はSATA)、 見栄えは写真(左)のHDDと変わらない (–> ノートPCのHDDをSSDに入れ替えて高速化できる)

(脚注) いまどきの（身の回りにある）PCが使うストレージ端子の規格はSATAかM.2くらいでしょう

デバイスの物理的な話: HDD

- 小さな磁石(N極とS極)で0と1を表現する媒体
  - 磁石の層が金属盤の上に塗られています
  - 金属板は常に回転しています
    - 回転速度(分速)は4200〜15,000rpm
  - 読み書きしたい場所にアームを近づけ、アームの先で磁場を使って読み書きします
    (アームを直径方向に移動させ、読み書きしたいセクタが回転してくるのを待ちます)

- YoutubeなどにHDDを分解して動作させている動画があるので、 それらを見た方がわかりやすいでしょう(-> 右の動画)



(脚注) レコードと一緒ですよ！って説明が通じない世代ですね... orz

デバイスの物理的な話: HDDのメリット、デメリット

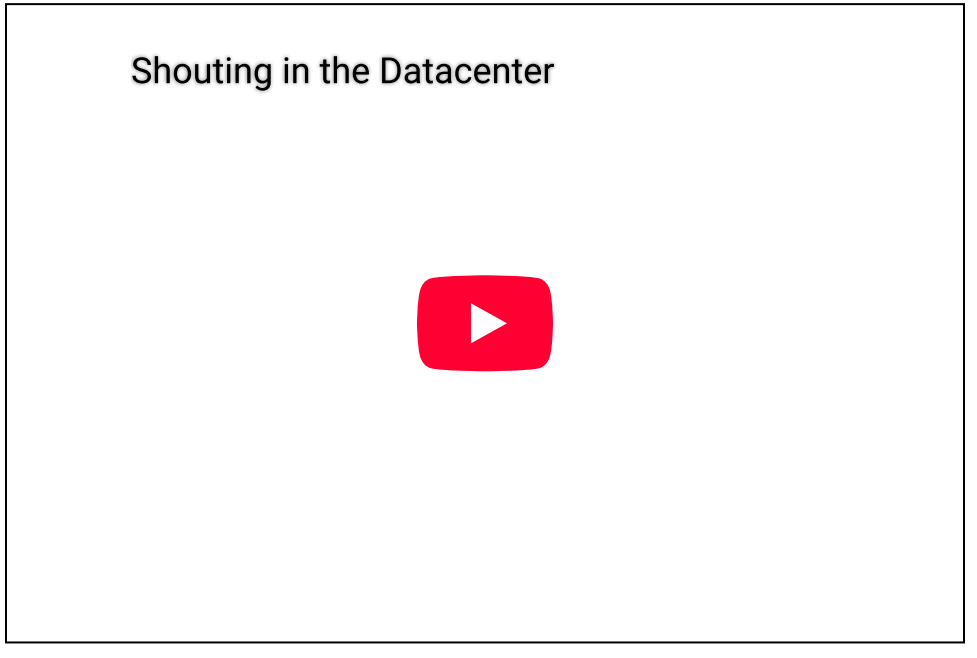
- 枯れた技術
- コストパフォーマンスがよい
- 保存性能も予想以上に高い

(脚注1) 最近はセンサーで墜落を関知した場合アームを固定して衝撃に備えてます (脚注2) 2021年末、Seagateが20TBのHDDを発売

(脚注3) アームは円盤上20nm ... もしHDDが飛行機サイズとしたら翼は地上数mmでしかない。 この精度で15000回転/分です...すごい

デバイスの物理的な話: HDDのデメリット

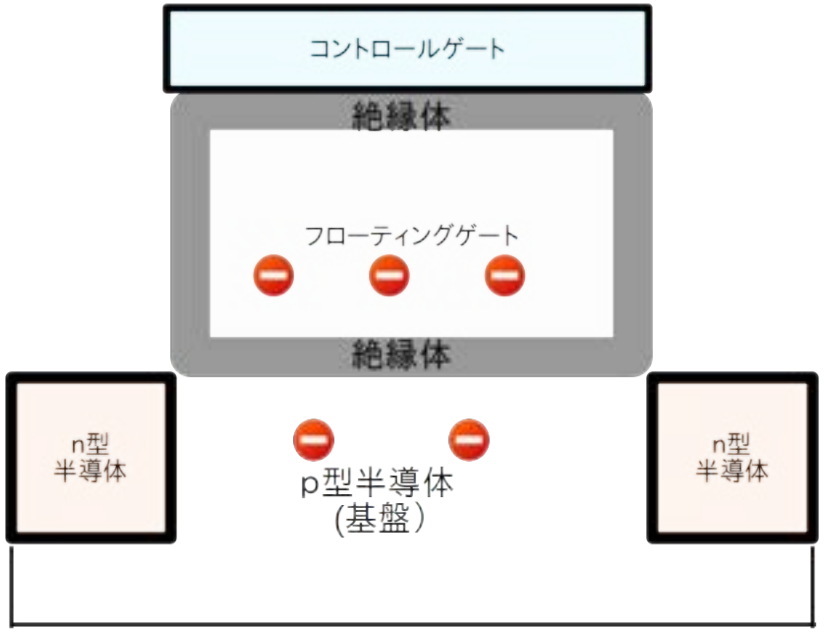
- ある程度の円盤の大きさは必要です(脚注1)
- 円盤を回しアームを動かすため
  - 回転させる電力が必要
  - 速度にも限界があります
  - 衝撃に弱い
  - ランダムアクセスは不得意
- 開放系
  - HDDは密閉されておらず、円盤は空気の粘性からくる圧力を利用して浮いています
  - 周囲の騒音(->動画)や空気の汚れの影響を受けます



(脚注1) 磁気媒体も原子1個サイズにはなりません。 かなりの数（数十万個？）の原子数で一塊の磁気的記録装置となっています。 原理的には12個で1ビットの実験に成功しているらしい (脚注2) 密閉型で、ヘリウムの中で回るHDDもありますね

デバイスの物理的な話: SSD(1)

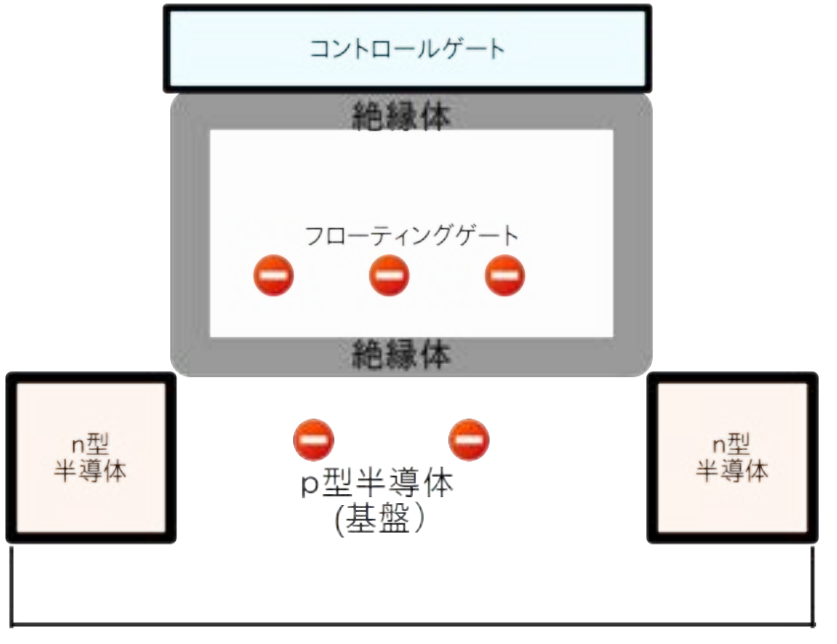
- LSI
- 絶縁体の箱の中に電荷がある/なしで1/0を表現
  - **SLC**は1/0(1bit)
  - 多値化
    - **MLC**(2bits)
    - **TLC**(3bits)
    - 箱内の電子量を細かく判定
- 書き込みや消去は**電圧をかけて強引に絶縁体を越えて電荷を移動**させます
  （これがSSDを痛める原因の一つです）



(参考) [いまさら聞けないSSDの基本](#) by (今はInsight Technology)の浅野さん (脚注) 量子力学効果のため、そろそろ微細化も限界で…

デバイスの物理的な話: SSD(2)

- 大きく分けて二つの型がります
- 右図は**フローティングゲート**型。 箱内の電子量により基盤側に流れる電流が変化するので、 そこから値(1-3bits)を判断します
- もうひとつ**チャージトラップ**型というのがあり、TLCは両タイプの製品があります
- **微細化は限界**
  - **3D**(縦方向の積層数)で勝負中
  - ようは高層マンション化ですね



(参考) [いまさら聞けないSSDの基本](#) by (今はInsight Technology)の浅野さん (脚注) 量子力学効果のため、そろそろ微細化も限界で...

デバイスの物理的な話: SSDのメリット

1. 軽い、小さい
2. 回転部品が無いので(HDDよりは)衝撃に強い
3. ランダムアクセスが得意(脚注1)
4. 最近だいぶ安くなり実用的になりました(脚注2)

(脚注1) ランダムアクセス ... LSIのセクタ番号を指定するだけなので、どの場所でも同一速度ですが、 HDDは円盤の回転とアームの移動が必要なので、 アクセス速度のバラつきが大きいです (脚注2) 価格 ... 最近は（最安値の単価比較で）HDDとの差は数倍ていどです。 数百GBならSSD一択でしょうが （そもそも今そんな小さなHDDって売ってるっけ？）、 十数TBとなるとHDD

(脚注3) SSDも大きくなってますから業務用ストレージもSSDベースが主流の雰囲気？ 専業メーカーとしてはPure Storageが老舗かな

デバイスの物理的な話: SSDのデメリット

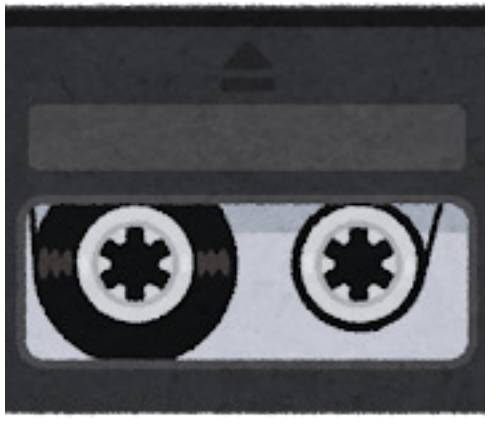
- 耐久性は（HDDほど）ない(脚注1)、**wear leveling**の性能が重要(脚注2)
  - 何もしなくても少しずつ崩壊しています。読み書き自体も半導体へ負荷をかけます
  - 半導体への負荷を均等化することが大事、つまり 特定の場所に読み書きが集中するのは避けないといけません
  - 均等に摩耗するようにコントローラが調整しています (**wear leveling**; wear=摩耗)
- 動作原理の違いからくるデメリット。例：処理単位の大きさ、消去
  - HDDのセクタより大きな単位で処理しています
  - とくにErase(消去)は大きな単位でしか処理できません。 (HDDのような単純な)更新処理は出来ません
- **コントローラ次第**(脚注2)
  - さまざまな合わせ技でSSDを動かすため、 **性能や耐久性はコントローラ**で大きく変わります

(脚注1) SSDによる長期保存は危険です。 物理的に部品へ無理がかかるため、 つねに少しずつ摩耗(崩壊)している媒体です。 有名ブランドのSSDをPCの耐用年数程度(数年)使うくらいなら問題ないでしょう。 つまり普段使いなら大丈夫です。 ただしHDDのように20年ものでも使えますか?というと、おそらく無理でしょう

(脚注2) Q: 安いSSDとは何か? A: 怪しげなコントローラとか、キャッシュを小さくするとか... (蓋を開けたらUSBメモリだったとかな:-)

デバイスの物理的な話: 磁気テープ(1)

- 容量単価、大容量性、長期保存性能という点で、 **磁気テープ**が最強の保存媒体です
  - 現在は手のひらサイズの1巻10TB台ですが、 1巻580TBの目処はあるそうです。まだまだ大容量化が可能
- 近年はLTFSという磁気テープ向けのファイルシステムも策定され使いやすくなりました！
  - LTO-5規格以降なら、 (HDD/SSDをみるように)磁気テープをフォルダとして開いて操作ができます



(脚注1) もはや磁気テープは日本(それも富士フィルムだけ?)しか製造しておらず、 全世界のクラウドの裏側は日本だより！ **需要拡大中**

(脚注2) 磁気テープの価格は手頃ですが磁気テープドライブがサーバ用しかなく高価(数十万〜)なので、 個人はテープよりHDDが手頃

(脚注3) ちなみにLTFSの仕様は Unix初期のファイルシステムそっくりでINDEXとデータを分けて管理

デバイスの物理的な話: 磁気テープ(2)

- ユーザとしては磁気テープよりAWS S3などにバックアップする時代でしょう
- クラウド事業者などの裏側の話をすると
  - 巨大ストレージのバックアップは磁気テープ一択
    - オンラインストレージは容量無制限なので普通のデバイスにバックアップなど出来ません
- ランサムウエア対策にもなるので需要急増中（ここは個人/法人とわず）
  - いったんテープに書きこんだらオフラインにする前提ですけどね

(脚注1) AWSは裏側を教えてくれないのですが、 AWS S3 Deep Archiveは磁気テープに書きこむから激安サービスなのではないかと疑っています。「読み出すのに半日お待ちください」みたいな仕様なんですもの。 テープを探しに行ってるんでしょ？(w)

(脚注2) たしかGoogleも磁気テープでバックアップをとっています

銀の弾丸などない

- SSDは万能ではありません
  「新技術も特定の分野に秀でているだけで前時代の問題の全てを解決するわけではありません」

- 身の回りからHDDや磁気テープがなくなることはあるかもしれませんが、
  クラウドの裏や業務用サーバでHDDや磁気テープがなくなることは(当面)ありません

(脚注1) HDDや磁気テープは永遠に不滅です！(は言い過ぎか?:-)

(脚注2) F.J.Brooks, "No Silver Bullet"、 Brooksの有名な「人月の神話」新装版の第16章に日本語訳が収録されている

【参考】データセンターが次世代のHDDに求めるもの

- Disks for Data Centers (Google, 2016)
  - https://research.google/pubs/pub44830/
  - ランダムアクセス性能
  - 容量、代替領域は不要なので、その分も容量へ回してほしい
  - 継続利用性
    - アーム先端のヘッドが壊れたらHDDは使えない?-> アームの先端部を冗長化してほしい
- Hyperscaler Storage (SNIA, 2016)
  - https://www.snia.org/educational-library/hyperscaler-storage-2016
  - 既存の標準規格や標準規格の改訂案がどのようにこれらの要件に対応できるかを検討
- Y. Li et.al., “Facilitating Magnetic Recording Technology Scaling for Data Center Hard Disk Drives through Filesystem-Level Transparent Local Erasure Coding” (USENIX, 2017)
  - https://www.usenix.org/conference/fast17/technical-sessions/presentation/li
  - 読み取りリトライの発生を減らすため、各HDDローカルに消失訂正符号化
  - OSのファイルシステム側にも改造を…

(アーカイブ動画で見ている人も)一時停止して休憩

「30分ごとに雑談（休憩）をしろ」というのが指導教官の教え

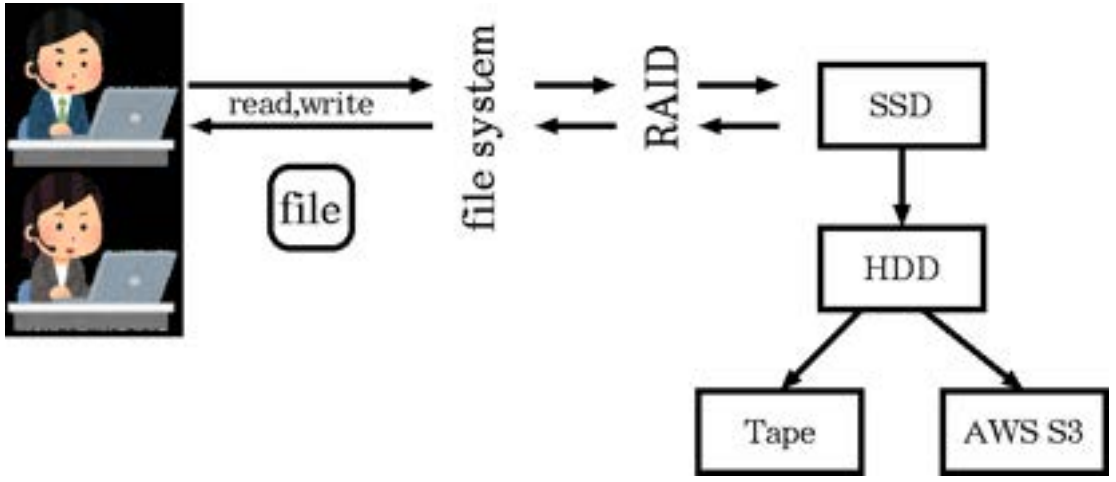
正確には「アメリカでは30分ごとにジョークを言わないといけない」だけれど、そんな洒落乙なこと無理:-)

ファイル管理技術

～ファイル,ファイルシステム,RAID～

【概要】データの読み書き

- オンプレミスの業務用サーバの場合、 読み書きは図のように様々なシステムを経由して、 最終的にSSDやHDDに書きこまれます
- 図のような多段構造があります
  - これをうまく運用できると良い
  - OSレベルの「ファイルシステム」
  - ファイルシステムが操作するハードウエア
    - HDDやSSDを（直接）操作
    - RAIDというHDD,SSDの塊を操作



(脚注1) 分散システムは（時間があれば）次回すこし触れます (脚注2) 図の右下は少し適当で、 HDDの生イメージをテープやS3にコピーしているようにも読めます。 テープなら生コピーがありうるかもしれませんがS3はセクタではなくファイルでしか操作できません

ファイルとファイルシステム

【用語】

- **主記憶装置** … メインメモリ
- **補助記憶装置**
  - **HDD** (ハードディスク)
  - **SSD** (**Solid State Drive**)
  - FD (フロッピーディスク)
  - 光ディスク(CD-ROM,DVD,BD)
  - **磁気テープ**(DAT)
  - USBメモリ,SDカードのたぐい

- Unix上でのデバイスの分類
  - **ブロックデバイス** … ブロック単位での読み書きをするデバイス
    - 代表的な単位はHDDのセクタ=512バイト
    - 例：HDD、SSD
  - **キャラクタデバイス** … 低速のデータ転送(1文字ずつのスピード感)
    - ブロックデバイスの対極、入力デバイスが典型。例： キーボード,マウス,ジョイスティック,スキャナー,バーコード

(脚注)【基本情報試験の出題範囲】(a)HDDは動作原理も含む (b)入出力装置や補助記憶装置の各種デバイスについての名称くらい

ファイルのデータ表現（半分は復習）

- ファイル ... 数字の**バイト列**を格納するもの
  - **データストリーム**
- プロトコル
  - 改行や空白(SPACEやTAB)を意味するコードを入れておき、 テキストを表示するプログラム(エディタやブラウザなど)が適切に表示
  - Q: 日本語は? A: UTF-8 (脚注)

```
HAMLET

To be, or not to be, that is the question,
Whether 'tis nobler in the mind to suffer
... 以下略 ...
```

(脚注) 21世紀のデファクトスタンダードは、 **Unicode**で定義された数字(code point)を**UTF-8**方式でエンコードした数字列。

表示の際は、 その数値を解釈し、該当する日本語フォントで表示します。 例：あ = 3042 (code point)、343 201 202 (UTF-8)

Unix上でのファイルという論理構造(or 概念 or プロトコル)

- ファイル=**メタデータ+データ(本体)**
  - Protocol Header = メタデータ
  - Protocol Payload = データ (例: Cのソースコードそのもの、テキストつまりUTF-8の数字の列)
- ファイル操作の例
  1. ファイル名の変更(a.c -> b.c)
     メタデータのみを書き換え
     (ファイル名と最終参照時刻を変更)
  2. ソースコードを書き換える場合
     データそのものの書き換え + メタデータを更新
     (サイズや最終参照時刻、最終更新時刻などを変更)

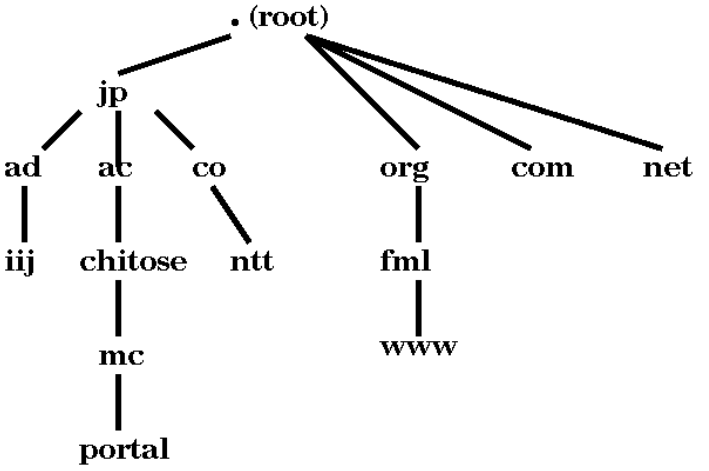
属性	備考
ファイル名	文字列
ファイルの長さ	バイト数
所有者のID	uid(数字)
所有者のグループ	gid(数字)
ファイルモード	いわゆるpermission
最終参照時刻	unixtime
最終更新時刻	unixtime
データ格納位置	セクタ番号（群）

表: メタデータの例

(脚注) 実際にファイルを操作する手段はファイルシステムが提供します。 システムコールでカーネルにファイルの操作を依頼します

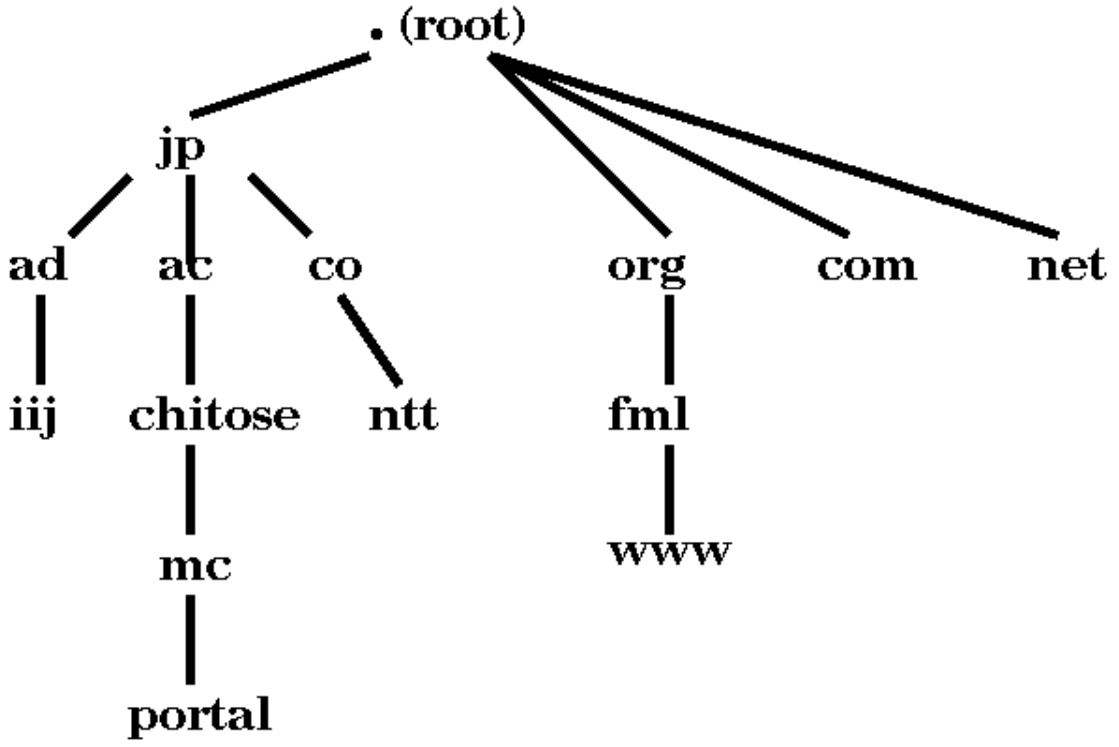
ファイルを管理する（無限階層）本棚

- ファイルは本で、ディレクトリ(フォルダ)は本棚
  - 本の奥付 = メタデータ
    - 例：タイトルや著者、出版日など
  - 本の中身 = データ(ファイルの本体)
- 本棚の中に本棚を作れます
  - 無限本棚
  - 無限マトリョーシカ
  - つまり階層化されています
    - 棚のなかを区切り、本棚の階層(入れ子構造)が作れます。 深い階層化も可能です



階層型ファイルシステム

- Unixのファイルシステムの特徴
  - 木構造
    - DNS(右図)と同様です
  - 階層型ファイルシステム
- 階層型ファイルシステム以外
  - 大昔の大型計算機
  - スマートフォンの見栄えは非階層（裏側は階層型ファイルシステム）

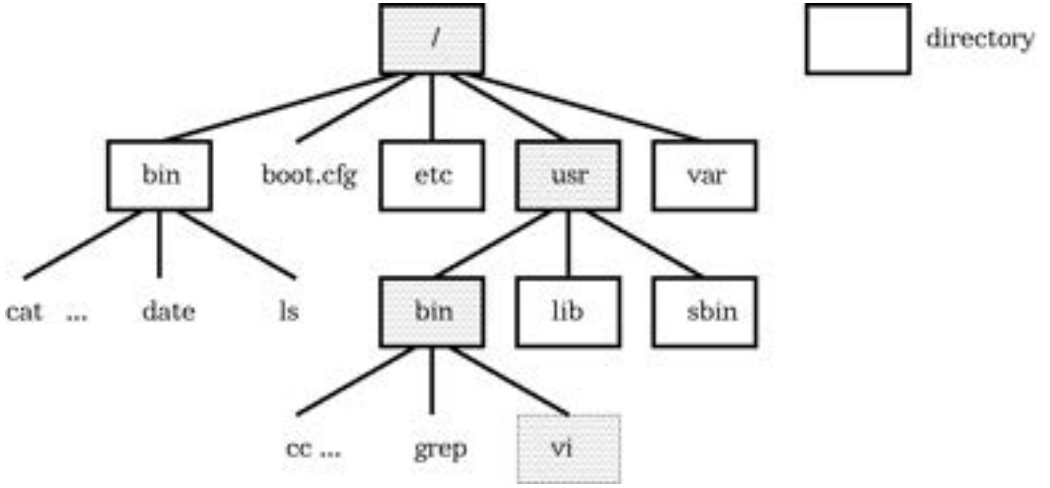


(脚注1) DNSのほうが後なので、正確にはDNSがUnixファイルシステムをリスペクトしている？

(脚注2) 大型計算機のファイルシステムは一階層なので本当に本棚みたいに見えます

Unixファイルシステム: ディレクトリ(フォルダ)と木構造

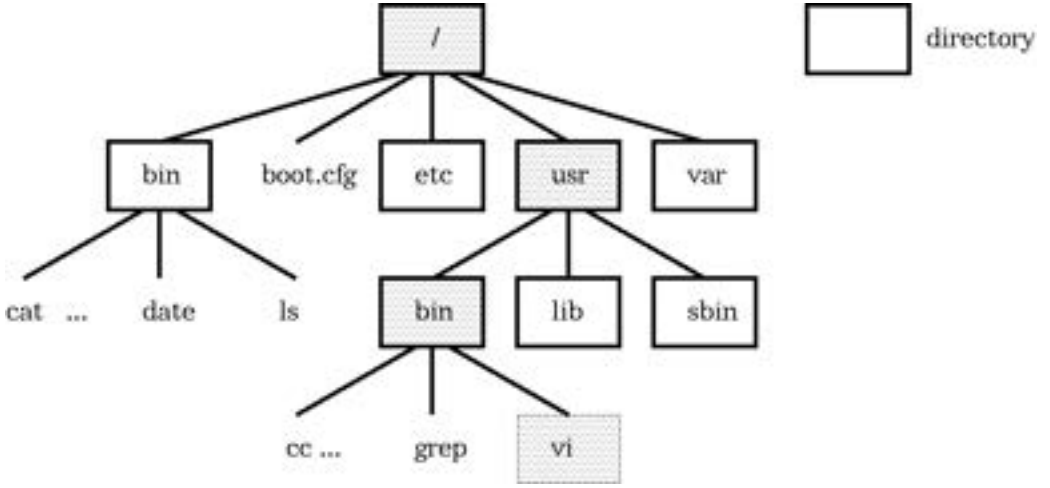
- ファイルをおさめる入れ物をUnix系では**ディレクトリ**、 Windowsなどでは**フォルダ**と呼びます (以下Unixの話をします)
- ディレクトリは**入れ子**にでき、**木構造**を構成
  - ディレクトリの中に、ディレクトリもファイルも入れられます
- Unixでは/で階層の区切りを表現します。 階層の一番上は**/**でrootと呼びます
  - たとえば/usr/bin/viは図(の網線部の)ような階層をあらわしています



(脚注1) rootからの木構造は(/を.にすれば)DNSと同じなので分かりますよね? (脚注2) Windowsの遠いご先祖MS-DOS 2.0でUnixの設計思想を大々的に取り入れたのですが、 MS-DOS 1.0のオプション記号/を維持するために、MS-DOSは区切りに/ではなく\を使い大迷惑(\はC言語で特別なのに...) (脚注3) T.Berners-LeeがURLを/区切りにしたのはUnix互換OS上での開発だからだと思う(推測)

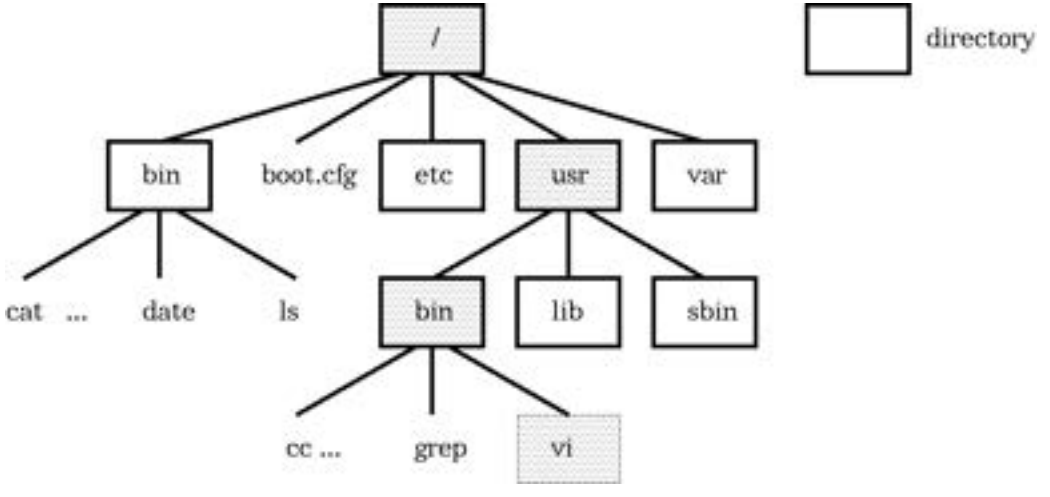
Unixファイルシステム: 表現形式、パス

- Unixでは/で階層の区切りを表現します
- 階層の一番上は/でrootと呼びます
    - **区切りも/、一番上も/**
      これが初心者を困らせる何か？
- たとえば/usr/bin/viは図(の網線部の)ような階層をあらわしています
- ファイルの位置を表現する/usr/bin/viのような表記を**パス(path)**と呼びます



絶対パス、相対パス、cdコマンド

- **絶対パス** ... /(root)からの階層をすべて表現
- **相対パス** ... 作業中の場所からの相対表記
- ディレクトリの移動はcdコマンドの引数にパスを書きます。 .(自分自身)と..(一つ上,親ディレクトリ)は特別なパスです




```
/usr/bin/vi  # #の右側はコメントです
cd /usr/bin  # /usr/binに移動して
./vi         # viを起動(/usr/bin/viの起動と同じ意味)
cd ..        # 一つ上に移動したので今いる場所は/usr
./bin/vi     # viを起動(/usr/bin/viの起動と同じ意味)
```

(脚注1) パスの問題は意外と基本情報処理に出ます

(脚注2) ./viは、そのディレクトリにあるviコマンドを実行という意味です。 コマンド検索時に.を探さないのがdefaultなので./が必要

ディレクトリ: lsコマンドでの表示(前頁の図と比較してください)

```
prompt> ls /
altroot/   boot.cfg   home/      libdata/   netbsd     root/      tmp/
bin/       dev/       kern/      libexec/   proc/      sbin/      usr/
boot       etc/       lib/       mnt/       rescue/    stand/     var/

prompt> ls -l /
  ... 省略 ...
drwxr-xr-x   2 root  wheel      1024 May 12 22:15 bin/
-rw-r--r--   1 root  wheel       172 May 12 22:15 boot.cfg
drwxr-xr-x  30 root  wheel      2560 Oct 23 10:04 etc/
  ... 省略 ...
drwxr-xr-x  15 root  wheel       512 Oct 20 09:59 usr/
drwxr-xr-x  25 root  wheel       512 Oct  8 21:12 var/

permission     ユーザ グループ サイズ 最終更新時刻 ファイル名もしくはディレクトリ名
```

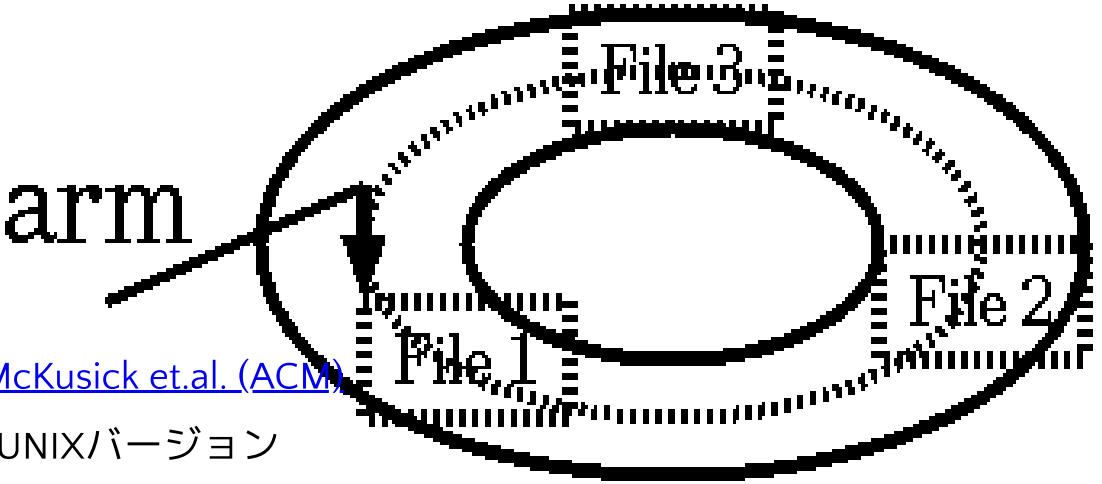
(脚注1) この例で使っているlsコマンドは、 対象がディレクトリの場合、 右端に/を追加して表示しています

(脚注2) オプションなしlsコマンドは名前一覧のみを表示、 -l オプションをつけると最重要なメタデータ情報とともに表示します

OSや運用レベルでの工夫

事例：4.2BSD FFS (Fast File SYstem)

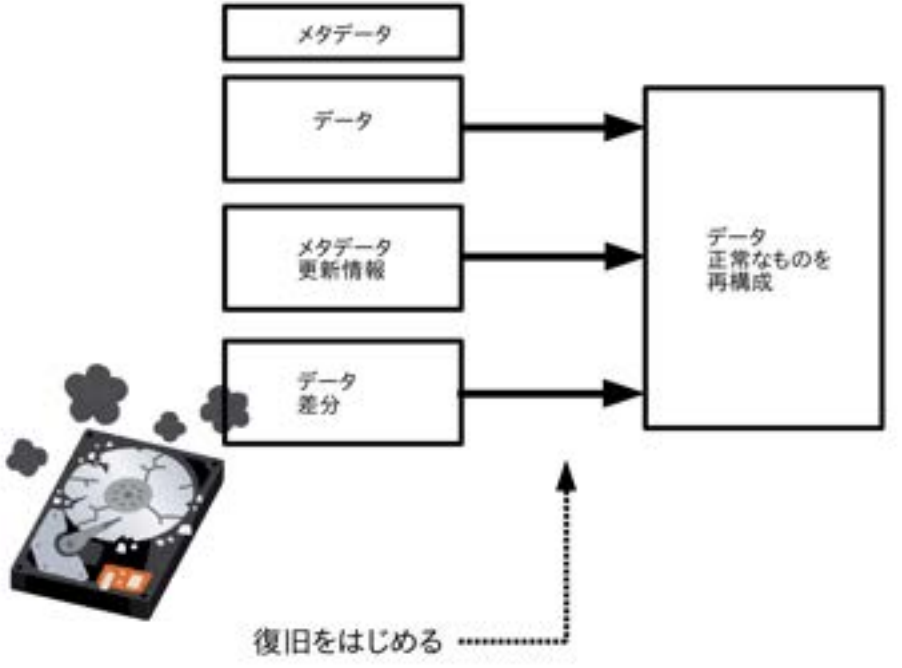
- HDDは回転する金属の円盤なので、その構造を意識した読み書きをすれば速くなるはず
- たとえば、書きこむ際、 アームを極力動かさずにすむ配置を考え、 なるたけデータを同心円上に配置していきます
- アームを直径方向には動かさず、円周に沿って読み書きするのが効率的です。 アームを上げて下げる必要がある場合は、上下の時間差と回転する分を考えてデータを配置しておきます



(脚注1) FFSの原論文(PDF,1984): [1] M.K.McKusick and W.N.Joy [2] M.K.McKusick et.al. (ACM)
FFSで劇的に高速化。 4.2BSDはTCP/IPとFFSという売りが多い有名なUNIXバージョン
(脚注2) この話は、もちろんSSDには関係ありません;-) (脚注3) この発想の元ネタはPWB UNIXにあるらしいです

事例：ファイルシステムレベルでの障害対策

- 電源断などの非常事態からの復旧を容易にする工夫もあります
  - メタデータの書き込む順序の最適化
  - ジャーナリング … メタデータやデータ変更履歴を記録（右図）
- 【注意】 これらは**中途半端な状態にならない(一貫性を維持する)努力であって、 ハードウエア障害でデータがなくなることは避けられません**
  - **データの消失をふせぐためには冗長化が必要で、 そのためにRAIDや分散システムを利用**



(脚注1) （ファイルシステム版）RDBMSのreflog (脚注2) FFS以前のUnix初期のファイルシステムは障害に弱いです

(脚注3) FFSの作者M.K.McKusickによる上述のような改善案としてSoft Updatesがあり、現在のBSD系ではデフォルトです

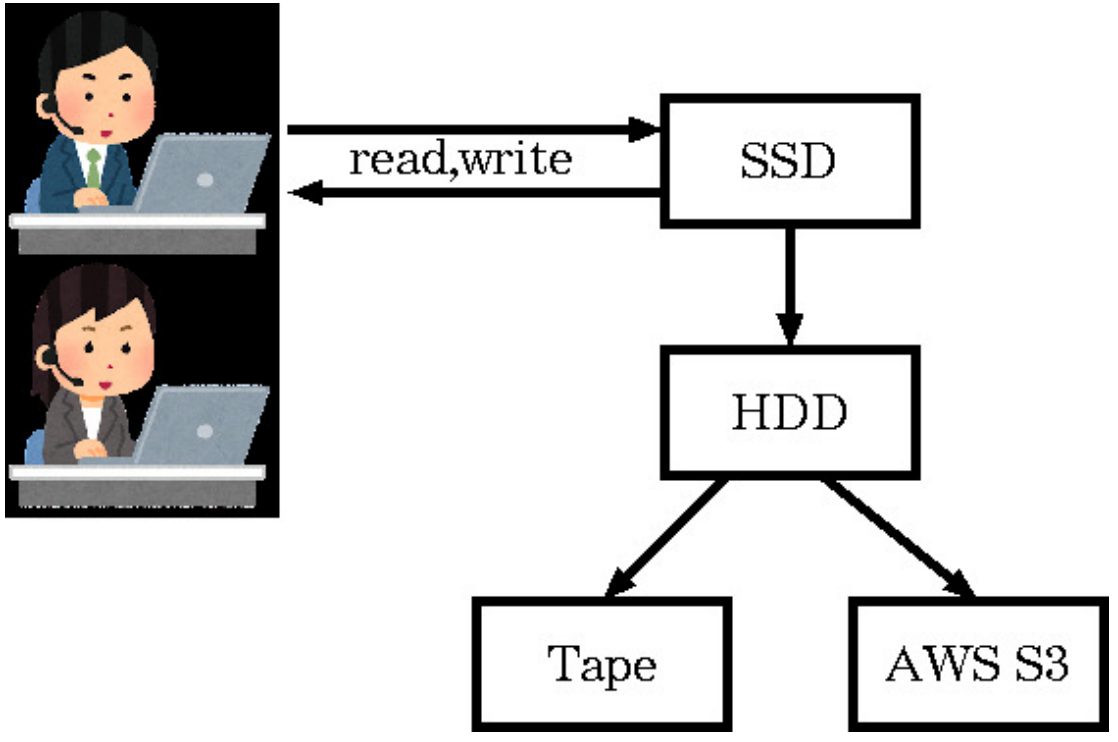
事例：【運用】データを確実に長期保存する

- データの保存（確実な読み書き）はOSの最も重要な機能の一つです
- 障害対策の例
  - ハードウエアでの耐障害性向上 -> RAID
    - サーバ機材では専用ハードウエア（RAIDカード）の利用が普通です
  - ソフトウエア(ミドルウエア)による冗長化
    - Googleに代表される分散システム（ネットワーク接続された安価なコンピュータ群で構成）
- 【運用】定期**バックアップ**は基本
  - 手動より自動がよい。 現代では保存先としてAWS S3やGoogle Driveなどの**オンラインストレージ**も選択できますが、利用には十分なネットワーク帯域が必要だし、 外部におく以上セキュリティには一層きをつけないといけません

(脚注1) データ作成に費やされた人間の時間は貴重なので確実にデータを保存したい！ (脚注2) 分散システムは次回すこし取り上げる予定です (脚注3) ランサムウエア対策としてもマメなバックアップは重要で、 バックアップしたものはREAD ONLYにできるとなおよし

事例：【運用】ストレージの階層化

- データにも**参照の局所性がある**ので、よく使うファイルはストレージのごく一部です
- SSD -> HDD -> 磁気テープの順に**大容量**ですが、その順に**低速**になります。 そこで、 まずは**SSDに読み書き**し、一ヶ月アクセスがないファイルはSSDからHDDへ移動し、 HDD上で3ヶ月アクセスがないファイルはHDDから磁気テープやAWS S3へ移動するといった運用が理想です(脚注2)
- これも一種の多段キャッシュ構造です



(脚注1) ホットデータ、コールドデータという表現があります (脚注2) SSDとHDD間で自動的に移動する製品(NASとか)が売られてますね。 自動化されていたら便利ですけど、 カーネルに実装しなくてもメタデータを見て適切に移動させるデーモンを書けばok

(脚注3) 例：AWS S3 -> S3 Deep Archive -> S3 Glacier(=氷河)。 アクセスを見て自動的に移動させる設定も出来ます

RAID

- Redundant Arrays of Inexpensive Disks -

(脚注) 本来はInexpensive Disksですが最近はIndependentの略と称することも

RAID (Redundant Arrays of Inexpensive Disks)

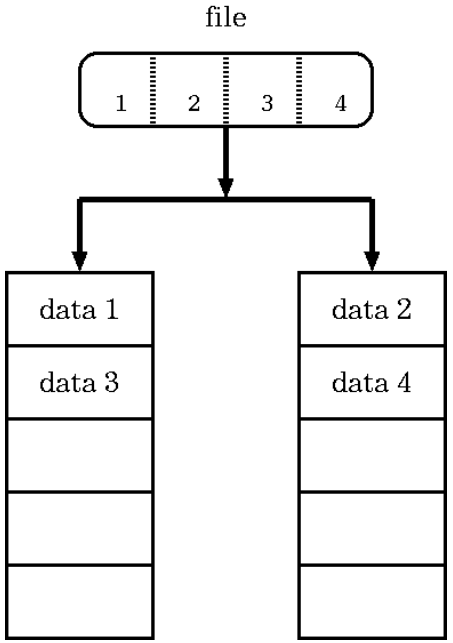
- 本来は**安いHDDをたくさんつなげて大きなHDDを作りたい**という動機
  - 1980年代後半の話でRAID 0などは、まさにそれ
- データの**冗長化**->耐故障性向上;サーバ機ではHDD障害にそなえてRAIDを組むのが基本
  - 実務では専用ハードウエア(RAIDカード)の利用が普通ですが
  - 個人用であればソフトウエア(カーネルの機能)でも十分な速度です
- RAIDの名称はレベル,よく見るレベルは0 1 5 6および10(いちぜろ)

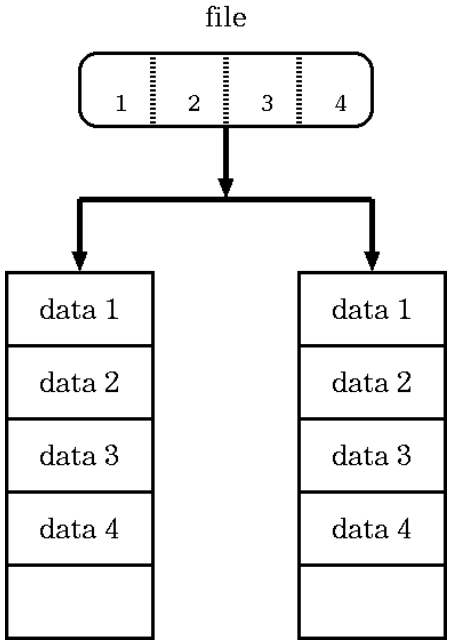
レベル	概要	冗長性	利用可能な最大容量	コスト
0	ストライピング:分散して書き込む	なし	1	安価
1	ミラーリング:2個のDISKに同じデータを書きこむ	あり	1/2	高価
5	データとパリティを書き込む	あり	(HDD数-1)/HDD数e.g.2/3	やや高価
10	RAID1群を作成後それらをRAID0で束ねる	あり	1/2	高価

(脚注1) Redundant Arrays of Inexpensive Disks つまり Inexpensive (安い)という用語のとおりです

(脚注2) RAIDレベル 2 3 4 は定義されていますが製品を見たことがありません

RAID 0および1

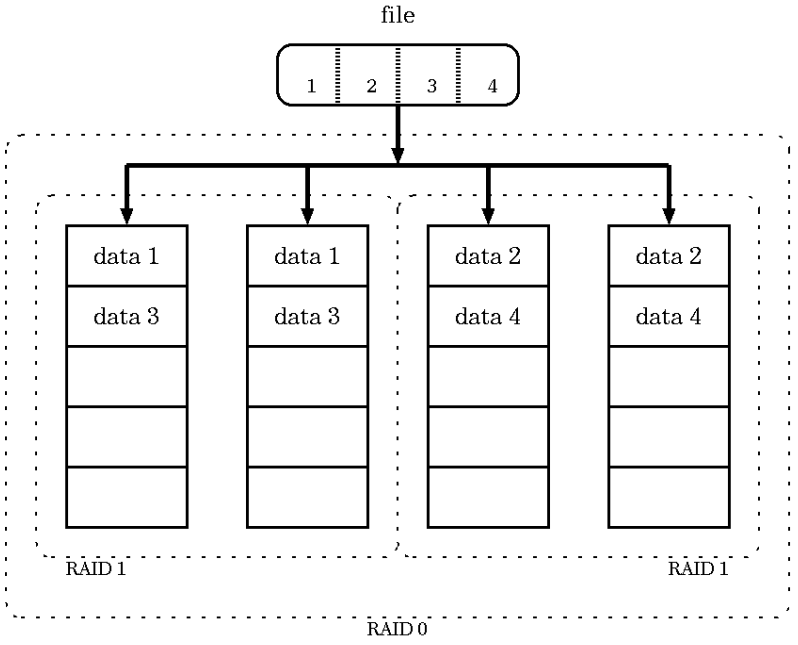




RAID 0 = ストライピング: 大容量かつ(HDDの並列動作で)高速化が可能ですが、HDDが1つ壊れるだけで全滅。 キャッシュなどの一時領域としては便利ですが、保存領域に使ってはいけません

RAID 1 = ミラーリング: 二つの HDD に同時に同じデータを書きこむ。 簡潔な理屈のため回路も簡単で信頼性が高いが、 HDDの半分しか使えない高価な手法。 読むときは2つのHDDから読みこむことで倍速が可

RAID 10



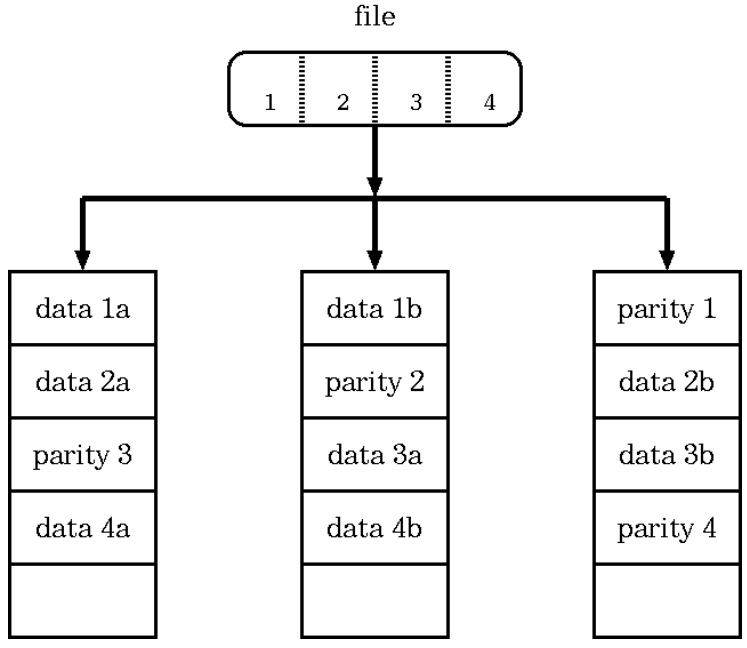
RAID 1のセットを作り、それらをRAID 0で束ねることで高速化と大容量化が可能
メリット: RAID 0と1のメリットのいいとこ取り
デメリット: RAID 1と同じでHDD数の1/2しか利用できない高価なところ

(脚注) RAID 10は数字の10(ten)ではなく1と0です

RAID 5および6

RAID 5 = RAID 1と0の妥協点とも言え、復旧するためのパリティデータを分散配置 (HDDが1台なら壊れても大丈夫、2台こわれたら全滅)。 パリティにHDD 1台分が使われますがRAID 1ほど高価な方法ではないため、よく見かけます。 例: 合計3台のHDDなら2個分、合計4なら3個分の容量が利用可能

RAID 6 = RAID 5 の拡張でパリティを二重にもつ方法です。 HDDの総容量はRAID 5より少なくなりますが故障には強くなります。 最近は 6 を見ることが多い気がします



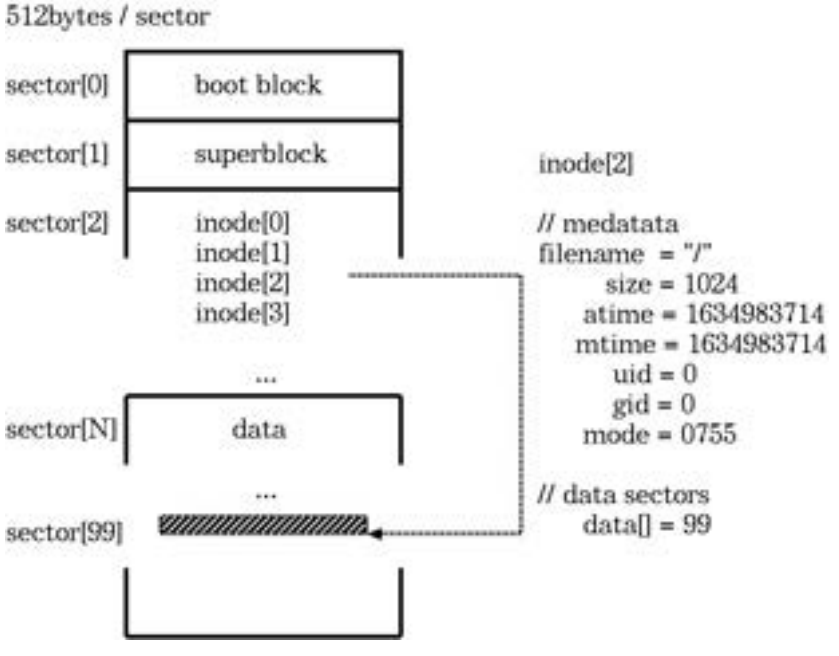
(脚注) RAID5の障害時からの復旧は、 壊れたHDDを新しいHDDに交換し、パリティを利用したHDDの再構成を行いますが、 この処理が非常に高負荷のため、 この負荷により別のDISKも壊れて結局HDDが全滅することがあります(実話) <- parity系RAIDがきらいな由縁

付録

(脚注) これ以降は中間試験に出ません

参考: ファイルシステムの例(Unixの初期, 1970年代)

- HDDは512バイト単位のsectorという配列と考えてください
- セクター0(sector[0], 先頭の512バイト)にはOS起動時に使う特別なデータを格納
- セクター1はHDDのlayoutなど設定情報が入っています(superblock, 詳細は略)
- セクター2から**inode構造体**配列が並び、 その後にデータが並んでいます(図ではセクターN以降がデータ): 各inodeが持つデータは (a)メタデータ (b)該当するファイルやディレクトリのデータを格納しているセクタ番号群(必要なだけ複数個)



(脚注1) 中間試験には出ません (脚注2) inodeはサーバのデバッグに必要な知識ですが、 OS各論すぎるので情報処理試験には出ません。 LPICでは出題されます。 さすがにレイアウトの詳細までは出題されない模様