第1章 序論 演習問題の解答と解説

1. オペレーティングシステムの主な機能を2つ挙げなさい。

- **[解答訳]** オペレーティングシステムは、ユーザーに拡張されたマシンを提供しなければならず、また、I/Oデバイスやその他のシステムリソースを管理しなければなりません。これらはある程度、異なる機能です。

- **[より詳しい解説]** この問題は、OSの最も基本的な2つの役割を問うものです。教科書では、OSの役割をトップダウンとボトムアップの2つの視点から説明しています。

    1. **拡張マシンとしてのOS (The Operating System as an Extended Machine)**: これはトップダウンの視点です。ハードウェアは本来、非常に複雑で扱いにくいものです。OSはこうした複雑なハードウェアを隠蔽し、プログラマに対して「ファイル」のような、よりシンプルで扱いやすい抽象化されたリソースを提供します。この抽象化により、プログラマはハードウェアの詳細を意識することなくアプリケーション開発に集中できます。
    2. **リソースマネージャとしてのOS (The Operating System as a Resource Manager)**: こちらはボトムアップの視点です。コンピュータにはCPU、メモリ、ディスクといった多くのリソースがあり、複数のプログラムがこれらを奪い合います。OSは、これらのリソースを秩序立てて管理し、どのプログラムにどのリソースを、いつ、どれくらい割り当てるかを決定します。リソースの共有には、時間を区切って交代で使わせる**時間的多重化**（例：CPU）と、空間を分割してそれぞれに割り当てる**空間的多重化**（例：メモリ）があります。

2. 1.4節では、9種類のオペレーティングシステムが説明されています。これらのシステムそれぞれについて、アプリケーションの例を挙げなさい（OSの種類ごとに1つ）。

- **[解答訳]** もちろん、多くの答えが考えられます。以下はその一例です。
    - **メインフレームOS**: 保険会社での請求処理。
    - **サーバーOS**: Siriのための音声テキスト変換サービス。
    - **マルチプロセッサOS**: ビデオ編集とレンダリング。
    - **パーソナルコンピュータOS**: ワードプロセッシングアプリケーション。
    - **ハンドヘルドコンピュータOS**: 文脈に応じた推薦システム。
    - **組み込みOS**: テレビ録画のためのDVDレコーダーのプログラミング。
    - **センサーノードOS**: 原生地での温度監視。
    - **リアルタイムOS**: 航空交通管制システム。
    - **スマートカードOS**: 電子決済。
- **[より詳しい解説]** この問題は、教科書1.4節「The Operating System Zoo」で紹介されている多様なOSの具体的な利用例を理解しているかを確認するものです。
    - **メインフレームOS**は、大量のI/Oを伴う多数のジョブを同時に処理することに特化しています。
    - **サーバーOS**は、ネットワークを通じて複数のユーザーにサービスを提供します。
    - **マルチプロセッサOS**は、複数のCPUを単一システムに接続し、大規模な計算能力を提供します。
    - **PC OS**は、単一ユーザーに良いサポートを提供することに重点を置いています。
    - **ハンドヘルドOS**（スマートフォンやタブレット）は、携帯デバイス上で動作します。
    - **組み込みOS**は、コンピュータとして一般的に認識されていないデバイスを制御します。
    - **センサーノードOS**は、無線で通信する小型のバッテリー駆動コンピュータ上で動作し、環境データを収集します。
    - **リアルタイムOS**は、時間が重要なパラメータであり、厳密なデッドラインを守る必要があります。
    - **スマートカードOS**は、クレジットカードサイズのデバイス上で動作し、非常に厳しい電力とメモリの制約があります。

3. タイムシェアリングシステムとマルチプログラミングシステムの違いは何ですか。

- **[解答訳]** タイムシェアリングシステムでは、複数のユーザーが同時に自分の端末を使ってコンピューティング

システムにアクセスし、計算を実行できます。マルチプログラミングシステムでは、1人のユーザーが同時に複数のプログラムを実行できます。全てのタイムシェアリングシステムはマルチプログラミングシステムですが、全てのマルチプログラミングシステムがタイムシェアリングシステムであるとは限りません。なぜなら、マルチプログラミングシステムは1人のユーザーしかいないPC上でも動作する可能性があるからです。

- **[より詳しい解説]** この2つの用語は、コンピュータの歴史の中で重要な概念です。
    - **マルチプログラミング**は、メモリを複数の区画に分割し、それぞれに異なるジョブ（プロセス）を配置する技術です。あるジョブがI/O待ちでCPUがアイドル状態になると、別のジョブにCPUを割り当て、CPUの利用率を高めます。これは第3世代コンピュータ（1965-1980）の重要な特徴でした。
    - **タイムシェアリング**は、マルチプログラミングの一形態で、各ユーザーがオンライン端末を持ち、対話的にシステムを利用できるようにしたものです。CPUは複数のユーザーのジョブを素早く切り替えながら実行することで、各ユーザーにあたかも自分専用のコンピュータがあるかのような錯覚を与え、短いコマンドに対する高速な応答時間を実現します。 解答にあるように、タイムシェアリングは「複数ユーザー」を前提とした対話的な利用形態を指すのに対し、マルチプログラミングは「複数プロセス」を同時にメモリに置いてCPU利用率を上げるという、より基本的な技術を指します。

4. キャッシュメモリを使用するため、メインメモリは通常32バイトまたは64バイト長のキャッシュラインに分割されます。キャッシュライン全体を一度にキャッシュすることには、単一のバイトやワードを一度にキャッシュするのと比べてどのような利点がありますか。

- **[解答訳]** 経験的な証拠から、メモリアクセスには**参照の局所性(locality of reference)** の原則が働くことがわかっています。これは、ある場所が読み込まれると、そのすぐ近くの場所（特にすぐ後の場所）が次にアクセスされる確率が非常に高いというものです。したがって、キャッシュライン全体をキャッシュすることで、次のキャッシュヒットの確率が高まります。また、現代のハードウェアは、個々のワードとして同じデータを読み込むよりも、32バイトや64バイトのブロック転送をキャッシュラインに行う方がはるかに高速です。

- **[より詳しい解説]** この問題は、キャッシュシステムの性能向上の基本原理を問うものです。

    1. **参照の局所性**: 教科書では、ほとんどのプログラムが少数のページに多数の参照を集中させる傾向があると説明されています。これはより小さな単位であるキャッシュラインにも当てはまります。例えば、プログラムの命令は通常シーケンシャルに実行され、配列の要素も連続してアクセスされることが多いです。あるメモリアドレスが参照されたとき、そのアドレスを含むキャッシュライン全体（例えば64バイト）をまとめてキャッシュに読み込むことで、次に参照される可能性が高いデータも同時に読み込まれ、キャッシュヒット率が向上します。
    2. **ハードウェアの効率**: メモリアクセスには、アドレスを指定してからデータが転送されるまでのレイテンシ（遅延）があります。このレイテンシは、転送するデータ量にはあまり依存しません。そのため、1ワードだけ転送するのも、64バイトのブロックをまとめて転送するのも、開始までの時間はほぼ同じです。ブロック転送を利用すれば、一度のアクセスで多くのデータを効率よく転送できるため、全体的なメモリアクセス性能が向上します。

5. 初期のコンピュータでは、読み書きされるデータの各バイトはCPUによって処理されていました（つまり、DMAはありませんでした）。このことは、マルチプログラミングにどのような影響を与えますか。

- **[解答訳]** マルチプログラミングの主な理由は、I/Oが完了するのを待つ間にCPUに何か他の仕事をさせることです。もしDMAがなければ、CPUはI/Oの処理で完全に占有されてしまうため、マルチプログラミングによって得られるもの（少なくともCPU利用率の観点では）は何もありません。プログラムがどれだけI/Oを行おうとも、CPUは100%ビジー状態になります。もちろん、I/Oが他の理由（例えばシリアル回線からのデータ到着）で遅い場合は、CPUが他の作業を行うことは可能です。

- **[より詳しい解説]** DMA（Direct Memory Access）は、CPUを介さずにコントローラが直接メインメモリとの間でデータを転送する仕組みです。DMAがない場合、CPUが1バイトずつデータを転送する**プログラムドI/O**を行う必要があります。 マルチプログラミングの目的は、あるプロセスがI/O待ちでブロック状態になったときに、

CPUを別の実行可能なプロセスに割り当てることで、CPUのアイドル時間を減らすことです。 しかし、DMAがない環境では、CPU自身がI/Oデータのコピー作業を行うため、「I/O待ち」の間もCPUはビジー状態になります。したがって、他のプロセスに切り替えるべき「CPUのアイドル時間」が存在しません。このため、マルチプログラミングを導入しても、CPU利用率の向上という主要なメリットは得られなくなります。

6. I/Oデバイスへのアクセスに関連する命令は、通常は特権命令です。つまり、カーネルモードでは実行できますが、ユーザーモードでは実行できません。これらの命令が特権命令である理由を説明しなさい。

- **[解答訳]** I/Oデバイス（例：プリンタ）へのアクセスは、通常、異なるユーザーに対して制限されています。あるユーザーは好きなだけ印刷でき、あるユーザーは全く印刷できず、またあるユーザーは特定のページ数までしか印刷できないかもしれません。これらの制限は、システム管理者によって何らかの方針に基づいて設定されます。ユーザーレベルのプログラムがそれらのポリシーを妨害できないように、ポリシーを強制する必要があります。

- **[より詳しい解説]** 解答はポリシーの観点から説明していますが、より根本的な理由は**OSによるリソースの保護と調停**です。

    - **保護**: ユーザープログラムが直接I/O命令を実行できると、ファイルシステムのアクセス権を無視してディスクの任意のセクタを読み書きしたり、他のユーザーの印刷ジョブに割り込んだり、ネットワーク上の他人のパケットを盗聴したりすることが可能になり、システムのセキュリティと安定性が崩壊します。
    - **調停**: 複数のプロセスが同時に同じデバイス（例：プリンタ）を使おうとすると、出力が混ざってしまい、意味をなさなくなります。OSは、印刷要求をスプーリング（一時的にディスクに保存）するなどして、各プロセスの要求を順番に処理し、混乱を防ぎます。 これらの保護と調停を実現するため、OSはI/Oデバイスへのアクセスを一手 に引き受ける必要があります。そのため、直接I/Oを行う命令はカーネルモードでのみ実行可能な特権命令とし、ユーザープログラムはシステムコールを通じてOSにI/Oを依頼する仕組みになっています。

7. 1960年代にIBM System/360メインフレームで導入された「コンピュータファミリー」という考え方は、現在では完全に時代遅れのものなのでしょうか、それとも生き続けているのでしょうか。

- **[解答訳]** それはまだ生き続けています。例えば、IntelはCore i3, i5, i7といった、速度や消費電力など様々な特性を持つCPUを製造しています。これらのマシンはすべてアーキテクチャ的に互換性があります。それらは価格と性能においてのみ異なりますが、これはファミリーという考え方の本質です。

- **[より詳しい解説]** コンピュータファミリーとは、同じ命令セットアーキテクチャを持つ一連のコンピュータで、価格と性能だけが異なるものです。これにより、顧客は小規模なモデルから始めて、ビジネスが成長するにつれて、既存のソフトウェアをすべて実行できるより大規模なモデルにアップグレードできます。 この考え方はIntelのx86アーキテクチャ（Core i3/i5/i7/i9シリーズなど）やARMアーキテクチャ（スマートフォンからサーバーまで）に明確に受け継がれています。例えば、あるCore i3プロセッサ向けに書かれたプログラムは、より高性能なCore i9プロセッサでも（より高速に）動作します。これにより、ソフトウェアの互換性が保たれ、広範な市場でソフトウェア資産を活かすことができます。これはIBM System/360が確立したビジネスモデルが現代でも有効であることを示しています。

8. GUIが当初なかなか採用されなかった理由の一つは、それをサポートするために必要なハードウェアのコストでした。25行×80列の文字モノクロテキストスクリーンをサポートするには、どれくらいのビデオRAMが必要ですか。1200×900ピクセルの24ビットカラービットマップではどうですか。1980年当時の価格（$5/KB）では、このRAMのコストはいくらでしたか。現在ではいくらですか。

- **[解答訳]** 25×80の文字モノクロテキストスクリーンには2000バイトのバッファが必要です。1200×900ピクセルの24ビットカラービットマップには3,240,000バイトが必要です。1980年当時、これら2つの選択肢のコストはそれぞれ$10と$15,820でした。現在の価格については、RAMが現在いくらで売られているか確認してください。おそらく1MBあたり数ペニーでしょう。

- **[より詳しい解説]** この問題は、技術の進歩がOSの機能（この場合はGUI）の普及にどれほど大きな影響を与え

たかを具体的に示すものです。

    - **テキストスクリーン**:
        - 必要なRAM: 25行 × 80列 = 2000文字。モノクロテキストは通常、1文字あたり2バイト（1バイトで文字コード、1バイトで属性）必要なので、4000バイト（約4KB）ですが、解答では2000バイトとしています。これは最もシンプルな実装を想定しているのでしょう。
        - 1980年のコスト: `2000 bytes / 1024 bytes/KB * $5/KB ≈ $9.77`。解答では$10としています。
    - **カラービットマップ (GUI)**:
        - 必要なRAM: 1200ピクセル × 900ピクセル × 24ビット/ピクセル = 25,920,000ビット。これをバイトに変換すると `25,920,000 / 8 = 3,240,000` バイト。
        - 1980年のコスト: `3,240,000 bytes / 1024 bytes/KB * $5/KB ≈ $15,820`。 1980年当時、PC本体よりも高価なRAMがGUIのためだけに必要だったことがわかります。RAM価格の劇的な低下が、GUIの普及を可能にした重要な要因の一つです。

9. オペレーティングシステムを構築する上での設計目標には、リソース使用率、適時性、堅牢性など、いくつかのものがあります。互いに矛盾する可能性のある2つの設計目標の例を挙げなさい。

- **[解答訳]** 公平性とリアルタイム性を考えます。公平性は、各プロセスがそのリソースを公平に割り当てられることを要求します。一方、リアルタイム性は、異なるプロセスが実行を完了しなければならない時間に基づいてリソースが割り当てられることを要求します。リアルタイムプロセスは、リソースの不釣り合いなシェアを得る可能性があります。

- **[より詳しい解説]** OS設計はトレードオフの連続です。解答の例以外にも、多くの矛盾する目標が存在します。

    - **応答時間とスループット**: 対話型システムでは、ユーザーの要求に素早く応答すること（応答時間の最小化）が重要です。しかし、短い時間で頻繁にプロセスを切り替えると、コンテキストスイッチのオーバーヘッドが増え、単位時間あたりに完了できる仕事の総量（スループット）は低下します。
    - **性能と移植性**: 特定のハードウェア機能を最大限に活用するようにOSを設計すれば性能は向上しますが、他のハードウェアへの移植性は著しく低下します。移植性を高めるには、ハードウェアを抽象化する必要がありますが、これは性能のオーバーヘッドを生む可能性があります。
    - **セキュリティと利便性**: セキュリティを強化すると、パスワードの要求、アクセスチェック、権限の制限などが増え、ユーザーの利便性は低下する傾向にあります。

10. カーネルモードとユーザーモードの違いは何ですか。2つの異なるモードを持つことが、オペレーティングシステムの設計にどのように役立つか説明しなさい。

- **[解答訳]** 現代のCPUの多くは、カーネルモードとユーザーモードという2つの実行モードを提供します。CPUは、カーネルモードで実行されているとき、その命令セット内のすべての命令を実行し、ハードウェアのすべての機能を使用できます。しかし、ユーザーモードで実行されているときは、命令のサブセットのみを実行でき、機能のサブセットのみを使用できます。2つのモードを持つことで、設計者はユーザープログラムをユーザーモードで実行させ、重要な命令へのアクセスを拒否することができます。

- **[より詳しい解説]** カーネルモードとユーザーモードの分離は、OSの**保護**機能の根幹をなす仕組みです。

    - **カーネルモード**: OS本体が動作するモードです。すべてのハードウェアにアクセスでき、メモリ管理やI/O、割り込みの禁止など、すべての特権命令を実行できます。スーパーバイザーモードとも呼ばれます。
    - **ユーザーモード**: アプリケーションプログラムが動作するモードです。実行できる命令やアクセスできるリソースが制限されています。例えば、他のプロセスのメモリ空間に直接アクセスしたり、ハードウェアを直接操作したりすることはできません。 この分離により、OSは以下のような利点を得ます。

1. **システムの保護**: 悪意のある、あるいはバグのあるユーザープログラムが、OS本体や他のプロセスを破壊することを防ぎます。
2. **安定性の向上**: ユーザープログラムがクラッシュしても、システム全体が停止することは稀です。OSは該当プロセスを終了させるだけで済みます。

3. **抽象化の実現**: OSはカーネルモードでのみ実行できる特権命令を使い、ハードウェアを管理します。そして、ユーザープログラムに対しては、システムコールという制御されたインターフェースを通じて、抽象化されたサービス（ファイル操作など）を提供します。

11. 255GBのディスクには65,536個のシリンダがあり、1トラックあたり255セクタ、1セクタあたり512バイトです。このディスクにはいくつのプラッタとヘッドがありますか。平均シリンダシーク時間が11ミリ秒、平均回転遅延が7ミリ秒、読み取り速度が100MB/秒と仮定して、1つのセクタから400KBを読み取るのにかかる平均時間を計算しなさい。

- **[解答訳]** ヘッド数 = 255 GB / (65536 * 255 * 512) = 16 プラッタ数 = 16 / 2 = 8 読み取り操作が完了するまでの時間 = シーク時間 + 回転遅延 + 転送時間。 シーク時間は11ms、回転遅延は7ms、転送時間は4msなので、平均的な転送には22msかかります。

- **[より詳しい解説]** この問題は、ディスクの物理的構造とアクセス時間の計算に関するものです。

    - **ヘッド数とプラッタ数の計算**:
        - **注意**: 問題文には矛盾があります。65,536シリンダ × 255セクタ/トラック × 512バイト/セクタ にヘッド数を掛けると255GBになる、とされています。しかし、65536 × H × 255 × 512 = 255 × 1024^3 を解くと、H は約32になります。解答の「16ヘッド」という答えから逆算すると、ディスク容量は約127.5GiBとなり、問題文の255GBとは一致しません。ここでは、解答集の計算に従います。
        - 解答集は 255 GB を何らかの値と解釈して ヘッド数 = 16 という答えを出しているようです。物理的には、トラック数はヘッド数と同じなので、1シリンダあたりのトラック数は16です。
        - プラッタ数は通常、ヘッド数の半分です（両面に記録するため）。16ヘッド / 2 = 8プラッタ となります。
    - **読み取り時間の計算**:
        - **平均シーク時間**: アームが目的のシリンダに移動する時間。問題文より 11 ms。
        - **平均回転遅延**: 目的のセクタがヘッドの下に来るまでの待ち時間。問題文より 7 ms。
        - **転送時間**: 実際にデータを読み取る時間。 転送時間 = 転送データ量 / 転送速度 = 400 KB / (100 MB/秒) = (400 × 1024 バイト) / (100 × 1024 × 1024 バイト/秒) = 400 / (100 × 1024) 秒 ≈ 0.0039 秒 = 3.9 ms。解答では4msとしています。
        - **合計時間** = 11 ms + 7 ms + 4 ms = 22 ms。

12. 以下の命令のうち、カーネルモードでのみ許可されるべきものはどれですか。 (a) すべての割り込みを無効にする。 (b) 時刻時計を読み取る。 (c) 時刻時計を設定する。 (d) メモリマップを変更する。

- **[解答訳]** (a), (c), (d) はカーネルモードに制限されるべきです。

- **[より詳しい解説]**

    - **(a) すべての割り込みを無効にする**: これをユーザープログラムが実行できると、OSのスケジューラ（クロック割り込みに依存）が動作しなくなり、システムを独占できてしまいます。これはシステムの安定性を損なうため、特権命令であるべきです。
    - **(b) 時刻時計を読み取る**: 時刻の読み取りは、システムに害を及ぼす操作ではないため、一般的にユーザーモードでも許可されます。
    - **(c) 時刻時計を設定する**: これをユーザープログラムが実行できると、課金情報やファイルのタイムスタンプなどを不正に操作できてしまいます。システムの整合性を保つため、特権命令であるべきです。
    - **(d) メモリマップを変更する**: これをユーザープログラムが実行できると、他のプロセスやOS自身のメモリ空間にアクセスできるようになり、システムの保護機構が完全に無効化されます。これは最も危険な操作の一つであり、絶対に特権命令でなければなりません。

13. あるシステムには2つのCPUがあり、各CPUは2つのスレッド（ハイパースレッディング）を持っています。3つのプログラムP0、P1、P2が、実行時間はそれぞれ5、10、20ミリ秒で開始されたとします。これらのプログラムの

実行が完了するまでにどれくらいの時間がかかりますか。3つのプログラムはすべて100% CPUバウンドであり、実行中にブロックせず、一度割り当てられたCPUを変更しないものと仮定します。

- **[解答訳]** これらのプログラムの実行が完了するまでにかかる時間は、OSがそれらをどのようにスケジュールするかによって、20、25、または30ミリ秒になります。もしP0とP1が同じCPUに、P2がもう一方のCPUにスケジュールされた場合、20ミリ秒かかります。もしP0とP2が同じCPUに、P1がもう一方のCPUにスケジュールされた場合、25ミリ秒かかります。もしP1とP2が同じCPUに、P0がもう一方のCPUにスケジュールされた場合、30ミリ秒かかります。もし3つすべてが同じCPU上にある場合は、35ミリ秒かかります。

- **[より詳しい解説]** この問題は、マルチプロセッサ環境でのスケジューリングが全体の完了時間にどう影響するかを示すものです。システムには2つのCPUがあり、それぞれが独立してプロセスを実行できます。（ハイパースレッディングは1つの物理コアで2つのスレッドを同時に実行する技術ですが、この問題では簡単化のため2つの独立したCPUとして考えます。）

    - **ケース1 (完了時間: 20ms)**:
        - CPU 1: P0 (5ms) + P1 (10ms) = 実行時間合計 15ms
        - CPU 2: P2 (20ms)
        - システム全体の完了時間は、両方のCPUの処理が終わるまでなので、`max(15, 20) = 20ms` となります。これが最も効率的な割り当てです。
    - **ケース2 (完了時間: 25ms)**:
        - CPU 1: P0 (5ms) + P2 (20ms) = 実行時間合計 25ms
        - CPU 2: P1 (10ms)
        - 全体の完了時間は `max(25, 10) = 25ms` です。
    - **ケース3 (完了時間: 30ms)**:
        - CPU 1: P1 (10ms) + P2 (20ms) = 実行時間合計 30ms
        - CPU 2: P0 (5ms)
        - 全体の完了時間は `max(30, 5) = 30ms` です。 このように、スケジューラによるプロセスの割り当て方（ポリシー）がシステム全体の性能を大きく左右します。

14. あるコンピュータには4ステージのパイプラインがあります。各ステージがその処理にかかる時間は同じで、1ナノ秒です。このマシンは1秒あたり何命令を実行できますか。

- **[解答訳]** 1ナノ秒ごとに1つの命令がパイプラインから出てきます。これは、マシンが1秒あたり10億命令を実行していることを意味します。パイプラインのステージ数は全く関係ありません。10ステージのパイプラインで各ステージが1ナノ秒かかる場合でも、1秒あたり10億命令を実行します。重要なのは、完成した命令がパイプラインの最後からどれくらいの頻度で出てくるかだけです。

- **[より詳しい解説]** パイプライン処理は、命令の実行を複数のステージ（例：命令フェッチ、デコード、実行、書き込み）に分割し、それらを流れ作業のように並行して処理する技術です。 パイプラインが完全に満たされた状態では、クロックサイクルごとに1つの命令が完了して出てきます。この問題では、1ステージが1ナノ秒かかるので、クロックサイクルは1ナノ秒です。したがって、1ナノ秒ごとに1つの命令が完了します。 スループット = 1命令 / 1ナノ秒 = 1命令 / (1 × 10^-9 秒) = 1 × 10^9 命令/秒 = 10億命令/秒 (1 GIPS) パイプラインのステージ数は、最初の命令が完了するまでの時間（レイテンシ）には影響しますが（この場合は4ナノ秒）、定常状態でのスループットには影響しません。

15. あるコンピュータシステムには、キャッシュメモリ、メインメモリ（RAM）、ディスクがあり、オペレーティングシステムは仮想メモリを使用しています。キャッシュから1ワードにアクセスするのに1ナノ秒、RAMから1ワードにアクセスするのに10ナノ秒、ディスクから1ワードにアクセスするのに10ミリ秒かかります。キャッシュヒット率が95%で、メインメモリヒット率（キャッシュミスの後）が99%の場合、1ワードにアクセスするための平均時間はどれくらいですか。

- **[解答訳]** 平均アクセス時間 = 0.95 × 1 ns (ワードがキャッシュにある場合) + 0.05 × 0.99 × 10 ns (ワードがRAMにあるがキャッシュにはない場合) + 0.05 × 0.01 × 10,000,000 ns (ワードがディスクにしかない場合) =

5001.445 ns = 5.001445 µs

- **[より詳しい解説]** この問題は、メモリ階層における平均アクセス時間の計算方法を問うものです。平均アクセス時間は、各階層でのアクセス確率（ヒット率）と、その階層でのアクセス時間の積を合計することで求められます。

    1. **キャッシュヒット**: 95% (0.95) の確率で発生し、コストは 1 ns。
        - 寄与: $0.95 \times 1 \text{ ns} = 0.95 \text{ ns}$
    2. **キャッシュミス、RAMヒット**: 5% (0.05) の確率でキャッシュミスが起き、そのうちの99% (0.99) の確率でRAMにヒットします。コストは 10 ns。
        - 寄与: $0.05 \times 0.99 \times 10 \text{ ns} = 0.495 \text{ ns}$
    3. **キャッシュミス、RAMミス（ディスクアクセス/ページフォールト）**: 5% (0.05) の確率でキャッシュミスが起き、さらにそのうちの1% (0.01) の確率でRAMにもミスします。コストは 10 ms = 10,000,000 ns。
        - 寄与: $0.05 \times 0.01 \times 10{,}000{,}000 \text{ ns} = 5000 \text{ ns}$ これらを合計すると、$0.95 + 0.495 + 5000 = 5001.445 \text{ ns}$ となります。この計算から、ごく稀にしか発生しないディスクアクセス（ページフォールト）が、平均アクセス時間に支配的な影響を与えることが分かります。

16. ユーザープログラムがディスクファイルを読み書きするためにシステムコールを行う際、どのファイルを対象とするか、データバッファへのポインタ、そしてバイト数を指定します。その後、制御はオペレーティングシステムに移り、適切なドライバが呼び出されます。ドライバがディスクを起動し、割り込みが発生するまで終了すると仮定します。ディスクからの読み込みの場合、呼び出し元は（データがないため）ブロックされる必要があります。ディスクへの書き込みの場合はどうでしょうか。呼び出し元はディスク転送の完了を待ってブロックされる必要がありますか。

- **[解答訳]** おそらく。呼び出し元が制御を取り戻し、すぐにデータを上書きした場合、書き込みが最終的に行われると、間違ったデータが書き込まれることになります。しかし、ドライバが最初にデータをプライベートバッファにコピーしてから返すのであれば、呼び出し元はすぐに続行できます。もう一つの可能性は、呼び出し元を続行させ、バッファが再利用可能になったときにシグナルを与えることですが、これはトリッキーでエラーが発生しやすいです。

- **[より詳しい解説]** この問題は、I/O操作における同期（ブロッキング）と非同期（ノンブロッキング）のトレードオフを問うています。

    - **ブロッキング書き込み**: システムコールがディスクへの書き込み完了まで戻らない方式です。プログラミングは簡単で安全です。呼び出し元は、システムコールから戻った時点でバッファを安全に再利用できます。しかし、ディスクI/Oは遅いため、その間プロセスは待機状態となり、CPU時間を有効活用できません。
    - **ノンブロッキング書き込み**: システムコールがすぐに戻る方式です。
        1. **カーネルバッファへのコピー**: OSがユーザーのバッファをカーネル内のバッファにコピーし、すぐに制御を返します。ユーザーはバッファをすぐに再利用でき、プログラミングも比較的容易です。しかし、カーネルへのデータコピーというオーバーヘッドが発生します。
        2. **シグナル/コールバック**: OSはコピーを行わず、ディスクへの書き込みが完了した時点で、シグナルやコールバック関数でユーザープロセスに通知します。コピーのオーバーヘッドはありませんが、非同期プログラミングは複雑で、競合状態などのバグを生みやすくなります。

17. トラップ命令とは何ですか。オペレーティングシステムにおけるその使用法を説明しなさい。

- **[解答訳]** トラップ命令は、CPUの実行モードをユーザーモードからカーネルモードに切り替えます。この命令により、ユーザープログラムはオペレーティングシステムのカーネル内の関数を呼び出すことができます。

- **[より詳しい解説]** トラップ命令は、ユーザープログラムがOSのサービスを要求するための**システムコール**の基本的な仕組みです。

    1. ユーザープログラムは、OSのサービス（例：ファイルの読み込み）が必要になると、ライブラリ関数を呼

び出します。
2. ライブラリ関数は、要求されたサービスの番号を特定のレジスタに設定し、TRAP命令を実行します。
3. TRAP命令が実行されると、CPUはハードウェア的に以下の動作をします。
    - モードをユーザーモードからカーネルモードに切り替える。
    - 現在のプログラムカウンタをスタックなどに保存する。
    - あらかじめ定められたOS内のアドレス（トラップハンドラ）にジャンプする。
4. OSのトラップハンドラは、レジスタの番号を見て、どのサービスが要求されたかを判断し、対応する処理を実行します。
5. 処理が完了すると、OSはモードをユーザーモードに戻し、保存されていたプログラムカウンタを使ってユーザープログラムのTRAP命令の次の命令から実行を再開します。 このように、トラップ命令はユーザーモードとカーネルモードの間の制御された「橋渡し」として機能し、OSの保護機構を維持しつつサービスを提供することを可能にします。

18. タイムシェアリングシステムにおいて、なぜプロセステーブルが必要なのですか。単一ユーザーでUNIXやWindowsを実行しているパーソナルコンピュータシステムでも必要ですか。

- **[解答訳]** プロセステーブルは、現在中断されているプロセス（レディ状態またはブロック状態）の状態を保存するために必要です。現代のパーソナルコンピュータシステムでは、ユーザーが何もしておらず、プログラムも開いていないときでさえ、何十ものプロセスが実行されています。それらは更新をチェックしたり、メールを読み込んだり、その他多くのことをしています。UNIXシステムではps -aコマンドを、Windowsシステムではタスクマネージャを使ってみてください。

- **[より詳しい解説]** プロセステーブルは、OSが各プロセスを管理するための中心的なデータ構造です。各エントリ（プロセス制御ブロックとも呼ばれる）には、プロセスの状態（実行中、レディ、ブロック）、プログラムカウンタ、レジスタの値、メモリ割り当て情報、開いているファイルなど、プロセスを再開するために必要なすべての情報が格納されています。

    - **タイムシェアリングシステム**では、CPUを複数のプロセスで素早く切り替える（コンテキストスイッチ）ことで、並列実行の幻想を作り出します。あるプロセスを中断して別のプロセスを動かす際に、中断したプロセスの状態をプロセステーブルに保存し、次に動かすプロセスの状態をテーブルから復元する必要があります。これがなければ、マルチプログラミングは不可能です。
    - **単一ユーザーのPC**でも、プロセステーブルは**不可欠**です。解答が指摘するように、現代のPCはバックグラウンドで多数のプロセス（デーモンやサービス）を常に実行しています（例：ネットワーク接続の管理、ウイルススキャン、ファイルのインデックス作成など）。ユーザーがワードプロセッサを起動すれば、それは新しいプロセスとして生成され、プロセステーブルにエントリが作られます。したがって、単一ユーザーであっても、システムは常にマルチプログラミング状態にあり、プロセステーブルは必須です。

19. 空ではないディレクトリにファイルシステムをマウントしたい理由はありますか。もしあるなら、それは何ですか。

- **[解答訳]** ファイルシステムをマウントすると、マウントポイントのディレクトリに既にあったファイルはアクセスできなくなるため、マウントポイントは通常空です。しかし、システム管理者は、マウントされたディレクトリで通常見つかる最も重要なファイルの一部をマウントポイントにコピーしておきたいと思うかもしれません。そうすれば、マウントされたデバイスが修理中の緊急時に、それらのファイルを通常のパスで見つけることができます。

- **[より詳しい解説]** この問題は、mountコマンドの動作の深い理解を問うています。 mountは、あるファイルシステム（例：USBメモリのファイルシステム）を、既存のファイルシステムの特定のディレクトリ（マウントポイント）に重ね合わせる操作です。マウント後、マウントポイントのディレクトリにアクセスすると、マウントされたファイルシステムのルートディレクトリが見えるようになり、元々そこにあったファイルやサブディレクトリは一時的に隠されてしまいます。 解答にあるシナリオは、一種の**フェイルセーフ機構**として機能します。例えば、/usrディレクトリが通常ネットワーク上のファイルサーバーからマウントされているとします。

もしネットワークに障害が発生した場合、/usrにアクセスできなくなると、システムは多くの基本的なコマンド（/usr/binにあるものなど）を実行できず、機能不全に陥ります。この事態を避けるため、ローカルディスクの/usrディレクトリ（マウントポイント）に、最低限必要なコマンド（ls, cpなど）のコピーを置いておくことがあります。通常時はネットワーク上の/usrがマウントされてそれらが使われますが、マウントに失敗した緊急時には、ローカルのコピーにアクセスできる、というわけです。

20. 以下のシステムコールそれぞれについて、失敗する原因となる条件を挙げなさい：fork、exec、unlink。

- **[解答訳]** forkは、プロセステーブルに空きスロットがない場合（そして、メモリやスワップスペースが残っていない可能性もある場合）に失敗することがあります。execは、指定されたファイル名が存在しないか、有効な実行可能ファイルでない場合に失敗することがあります。unlinkは、アンリンクしようとするファイルが存在しないか、呼び出し元プロセスにそれをアンリンクする権限がない場合に失敗することがあります。

- **[より詳しい解説]** これらのシステムコールはプロセス管理とファイル管理の基本です。

    - **fork**: 新しいプロセスを生成します。OSが管理できるプロセス数には上限があり、これはプロセステーブルのサイズによって決まります。テーブルが満杯のときにforkを呼び出すと、新しいプロセスのエントリを作成できず失敗します。また、プロセスのメモリイメージを確保するためのメモリやスワップ領域が不足している場合も失敗します。
    - **exec**: 現在のプロセスのメモリイメージを指定されたプログラムで置き換えます。失敗する主な理由は以下の通りです。
        - **ENOENT**: 指定されたファイルが存在しない。
        - **EACCES**: ファイルに対する実行権限がない。
        - **ENOEXEC**: ファイルが実行可能なバイナリ形式ではない（例：マジックナンバーが違う、シェルスクリプトでインタプリタ指定がないなど）。
    - **unlink**: ディレクトリからファイルエントリを削除します。失敗する主な理由は以下の通りです。
        - **ENOENT**: 指定されたファイルが存在しない。
        - **EACCES**: ファイルが含まれているディレクトリに対する書き込み権限がない。UNIXでは、ファイルの削除はディレクトリの変更と見なされるため、ディレクトリの書き込み権限が必要です。

21. 以下のリソースを共有するために、どの種類の多重化（時間、空間、または両方）が使用できますか：CPU、メモリ、ディスク、ネットワークカード、プリンタ、キーボード、ディスプレイ。

- **[解答訳]** 時間的多重化: CPU, ネットワークカード, プリンタ, キーボード。 空間的多重化: メモリ, ディスク。 両方: ディスプレイ。

- **[より詳しい解説]** OSのリソース管理の基本は、時間的多重化と空間的多重化です。

    - **時間的多重化 (Time Multiplexing)**: 複数のプロセスがリソースを交代で使います。
        - **CPU**: タイムシェアリングシステムでは、CPU時間は短いクォンタムに分割され、各プロセスに順番に割り当てられます。
        - **ネットワークカード/プリンタ/キーボード**: これらは一度に一つの仕事しかできません。複数の要求はキューに入れられ、順番に処理されます。
    - **空間的多重化 (Space Multiplexing)**: 各プロセスがリソースの一部を同時に割り当てられます。
        - **メモリ**: 物理メモリは複数のプロセスに分割して割り当てられます。
        - **ディスク**: ディスク容量は複数のファイル（異なるプロセスやユーザーに属するもの）に分割して割り当てられます。
    - **両方**:
        - **ディスプレイ**: GUI環境では、画面（空間）が複数のウィンドウに分割されます。しかし、一度にアクティブにできるウィンドウは通常一つであり、CPU時間やGPU時間はウィンドウごとに時間的に切り替えられます。

22. `count = write(fd, buffer, nbytes);` という呼び出しは、countにnbytes以外の値を返すことがありますか。も

しあるなら、それはなぜですか。

- **[解答訳]** 呼び出しが失敗した場合、例えばfdが不正である場合、-1を返すことがあります。また、ディスクが満杯で要求されたバイト数を書き込めない場合にも失敗することがあります。正常終了した場合、常にnbytesを返します。

- **[より詳しい解説]** この解答は少し不正確です。writeシステムコールの仕様（特にUNIX系）では、正常終了した場合でもnbytesより小さい値を返すことがあります。これは**部分的書き込み (partial write)** と呼ばれます。

    - **-1を返す場合（エラー）**:
        - EBADF: ファイルディスクリプタfdが無効。
        - ENOSPC: デバイスに空き容量がない。
        - EPIPE: パイプやソケットの読み取り側が閉じられている。
        - その他多数のエラーがあります。
    - **nbytesより小さい正の値を返す場合**:
        - 書き込み中にシグナルによって中断された場合。
        - カーネル内のバッファがいっぱいになった場合（特にネットワークソケットなど）。
        - 書き込むデバイスの制限（例：一度に書き込める最大サイズ）。 したがって、信頼性の高いプログラムを書くためには、writeの戻り値がnbytesより小さい場合、残りのデータを書き込むためのループを実装する必要があります。ただし、通常のディスクファイルへの書き込みでは、エラーがない限りnbytesが返されることがほとんどです。

23. ファイルディスクリプタfdを持つファイルに、3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 というバイト列が含まれているとします。以下のシステムコールが実行されます。 `lseek(fd, 3, SEEK_SET); read(fd, &buffer, 4);` ここで`lseek`呼び出しは、ファイルの3バイト目にシークします。`read`が完了した後、`buffer`には何が含まれていますか。

- **[解答訳]** これには、1, 5, 9, 2 のバイトが含まれています。

- **[より詳しい解説]** この問題は`lseek`と`read`の動作を理解しているかを確認するものです。

    1. **ファイルポインタ**: OSは開いているファイルごとに「ファイルポインタ」を保持しており、これが次に読み書きする位置を示します。ファイルを開いた直後は、ポインタはファイルの先頭（バイト0）を指しています。
    2. **`lseek(fd, 3, SEEK_SET)`**: この呼び出しは、ファイルポインタをファイルの先頭(SEEK_SET)から3バイト目に移動させます。バイトは0から数えるので、ポインタは4番目のバイトを指すことになります。 ファイル内容: `3, 1, 4, 1, 5, 9, 2, 6, ...`　　バイト位置:　`0 1 2 3 4 5 6 7`
    `^　　　　　　　　　　　　| ファイルポインタはここ`
    3. **`read(fd, &buffer, 4)`**: この呼び出しは、現在のファイルポインタの位置から4バイトを読み込み、`buffer`に格納します。読み込まれるのはバイト3, 4, 5, 6です。
        - バイト3の値は 1
        - バイト4の値は 5
        - バイト5の値は 9
        - バイト6の値は 2 したがって、`buffer`の内容は `1, 5, 9, 2` となります。`read`が完了すると、ファイルポインタは読み込んだ分だけ進み、バイト7（値は6）の位置を指すことになります。

24. 10MBのファイルがディスクの同じトラック（トラック50）の連続したセクタに保存されているとします。ディスクアームは現在、トラック番号100の上にあります。このファイルをディスクから取得するのにどれくらいの時間がかかりますか。アームをあるシリンダから次のシリンダへ移動するのに約1ミリ秒、ファイルの先頭が保存されているセクタがヘッドの下に回転してくるのに約5ミリ秒かかると仮定します。また、読み取りは200MB/秒の速度で行われるとします。

- **[解答訳]** ファイルの取得時間 = 1 * 50 ms (アームをトラック50の上に移動する時間) + 5 ms (最初のセクタがヘッドの下に回転してくる時間) + 10/200 * 1000 ms (10MBの読み取り) = 105 ms。

- **[より詳しい解説]** ディスクアクセス時間は、シーク時間、回転遅延、転送時間の合計です。
    - **シーク時間**: アームが現在のトラック100から目的のトラック50まで移動する時間。
        - 移動するトラック数: 100 - 50 = 50 トラック
        - シーク時間: 50トラック × 1 ms/トラック = 50 ms
    - **回転遅延**: 目的のセクタがヘッドの下に到達するまでの待ち時間。
        - 問題文より 5 ms
    - **転送時間**: 実際にデータをディスクから読み取る時間。
        - 転送時間 = 転送データ量 / 転送速度
        - = 10 MB / (200 MB/秒) = 0.05 秒 = 50 ms
    - **合計時間** = 50 ms（シーク）+ 5 ms（回転遅延）+ 50 ms（転送）= 105 ms。

25. ブロックスペシャルファイルとキャラクタスペシャルファイルの根本的な違いは何ですか。

- **[解答訳]** ブロックスペシャルファイルは、番号付きのブロックで構成されており、各ブロックは他のすべてのブロックとは独立して読み書きできます。任意のブロックにシークして読み書きを開始することが可能です。キャラクタスペシャルファイルではこれは不可能です。

- **[より詳しい解説]** UNIX系OSでは、デバイスはファイルとして扱われます。これらは/devディレクトリに置かれ、スペシャルファイルと呼ばれます。
    - **ブロックスペシャルファイル (Block Special File)**:
        - **対象デバイス**: ハードディスク、SSD、CD-ROMドライブなど。
        - **特徴**: データは固定長の**ブロック**単位で管理されます。**ランダムアクセス**が可能で、lseekシステムコールを使ってファイル内の任意の位置（ブロック）に直接アクセスできます。OSは通常、これらのデバイスに対してバッファキャッシュを使用し、I/O性能を最適化します。
    - **キャラクタスペシャルファイル (Character Special File)**:
        - **対象デバイス**: キーボード、マウス、プリンタ、シリアルポート、端末など。
        - **特徴**: データは**文字のストリーム**として扱われ、ブロックという概念はありません。**シーケンシャルアクセス**のみが可能で、シークすることはできません。データは到着した順に処理されます。

26. 図1-17の例では、ライブラリプロシージャはreadと呼ばれ、システムコール自体もreadと呼ばれています。これら両方が同じ名前であることが不可欠ですか。もしそうでなければ、どちらがより重要ですか。

- **[解答訳]** システムコールは、ドキュメント上の意味合いを除いて、実際には名前を持っていません。ライブラリプロシージャ read がカーネルにトラップするとき、それはシステムコールの番号をレジスタかスタックに置きます。この番号がテーブルへのインデックスとして使われます。実際にはどこにも名前は使われません。一方、ライブラリプロシージャの名前は非常に重要です。なぜなら、それがプログラム中に現れるものだからです。

- **[より詳しい解説]** この問題は、システムコールの仕組みの核心を突いています。
    1. **プログラマが見る世界**: プログラマはC言語で read() という**ライブラリ関数**を呼び出します。この「名前」がプログラマにとってのインターフェースであり、可読性と移植性のために非常に重要です。
    2. **ライブラリ関数の内部**: 呼び出されたライブラリ関数 read は、readシステムコールに対応する**システムコール番号**（例：3番）を特定のレジスタに設定します。
    3. **カーネルの動作**: ライブラリ関数が TRAP（または SYSCALL）命令を実行すると、OSに制御が移ります。OSはレジスタ内の「番号」を見て、システムコールテーブルをその番号でインデックスし、対応するハンドラ（実際の処理を行うカーネル内の関数）を呼び出します。 このように、カーネルは「名前」ではなく「番号」でシステムコールを識別するため、両者が同じ名前である必要はありません。プログラマが直接使う**ライブラリプロシージャの名前**の方が重要です。

27. 現代のオペレーティングシステムは、プロセスのアドレス空間をマシンの物理メモリから分離しています。この設計の利点を2つ挙げなさい。

- **[解答訳]** これにより、実行可能プログラムをマシンのメモリの異なる部分に、異なる実行回でロードすることができます。また、プログラムのサイズがマシンのメモリサイズを超えることも可能になります。

- **[より詳しい解説]** アドレス空間というメモリ抽象化は、現代OSの根幹をなす概念です。

    1. **保護 (Protection)**: 各プロセスは独立したアドレス空間を持つため、あるプロセスが他のプロセスやOSのメモリを（意図的または誤って）破壊することを防ぎます。これにより、システムの安定性とセキュリティが大幅に向上します。
    2. **再配置 (Relocation)**: 解答が指摘するように、プロセスは物理メモリの任意の位置にロードできます。OSは、各プロセスのアドレス0を、物理メモリの空いている任意のアドレスにマッピングできます。これにより、メモリ管理が柔軟になります。
    3. **仮想メモリの実現**: 解答の2番目の点です。プロセスのうち、現在必要な部分だけを物理メモリに置き、残りをディスクに置くことができます。これにより、物理メモリのサイズよりも大きなプログラムを実行したり、より多くのプログラムを同時にメモリに置いてマルチプログラミングの効率を上げたりできます。

28. プログラマにとって、システムコールは他のライブラリプロシージャの呼び出しと同じように見えます。プログラマがどのライブラリプロシージャがシステムコールになるかを知ることは重要ですか。どのような状況で、そしてなぜですか。

- **[解答訳]** プログラムのロジックに関する限り、ライブラリプロシージャの呼び出しがシステムコールになるかどうかは問題ではありません。しかし、性能が問題になる場合、どのタスクがシステムコールなしで達成できるかを知っていれば、プログラムはより速く実行されます。すべてのシステムコールには、ユーザーコンテキストからカーネルコンテキストへの切り替えに伴うオーバーヘッド時間が発生します。さらに、マルチユーザーシステムでは、システムコールが完了すると、OSが別のプロセスを実行するようにスケジュールする可能性があり、呼び出し元プロセスの実時間での進行をさらに遅らせます。

- **[より詳しい解説]** プログラマがシステムコールを意識すべき状況は、主に以下の3点です。

    1. **パフォーマンス**: 解答の通り、システムコールはユーザーモードとカーネルモードの間のコンテキストスイッチを伴い、これは通常の関数呼び出しに比べて桁違いに遅いです。例えば、1文字ずつファイルを読み書きするプログラムは、バッファリングを使って一度に多くの文字を読み書きするプログラムよりも、システムコールの呼び出し回数が多くなり、性能が著しく低下します。
    2. **エラー処理**: システムコールは、OSが管理するリソース（ファイル、ネットワーク、プロセスなど）と対話するため、失敗する可能性があります（例：ファイルが存在しない、権限がない、ディスクが満杯など）。プログラマは、システムコールを呼び出すライブラリ関数の戻り値を常にチェックし、エラーを適切に処理する必要があります。
    3. **セキュリティと移植性**: システムコールはOSの核となる機能と直接結びついています。OSに依存した、あるいは非標準のシステムコールを使うと、プログラムの移植性が損なわれます。また、システムコールの使い方によっては、セキュリティ上の脆弱性を生む可能性もあります（例：TOCTOU攻撃）。

29. 図1-23は、多くのUNIXシステムコールには対応するWin32 APIがないことを示しています。Win32に対応するものがないとリストされている各コールについて、UNIXプログラムをWindowsで実行するために変換するプログラマにとってどのような影響がありますか。

- **[解答訳]** いくつかのUNIXコールにはWin32 APIの対応物がありません。
    - `Link`: Win32プログラムは、ファイルに代替名で参照したり、複数のディレクトリで見たりすることはできません。また、リンクを作成しようとすることは、ファイルのロックをテストして作成する便利な方法です。
    - `Mount`と`umount`: Windowsプログラムは、複数のディスクドライブを持つシステムでは、パス名のドライブ名部分が異なる可能性があるため、標準的なパス名を仮定することはできません。
    - `Chmod`: Windowsはアクセス制御リストを使用します。
    - `Kill`: Windowsプログラマは、協力していない不正なプログラムを強制終了させることはできません。
- **[より詳しい解説]** これは、UNIXとWindowsの設計思想の違いを反映しています。

- **`link`**: UNIXのファイルシステムはi-nodeベースであり、ファイルの実体と名前が分離しています。`link`（ハードリンク）はこの構造を活用します。WindowsのNTFSもハードリンクをサポートしていますが、Win32 APIの初期の設計では、より単純なファイルモデルが想定されていたため、標準APIには含まれていませんでした。プログラマは、ファイルの共有やアトミックな名前の変更といった`link`の機能を使えず、代替手段（コピーやより複雑なAPI）を探す必要があります。
- **`mount/umount`**: UNIXは単一の統一されたファイルツリーを目指しますが、Windowsはドライブレター（`C:`, `D:`など）でファイルシステムを分離します。UNIXから移植されたプログラムは、パス名を柔軟に扱えるように修正する必要があります。
- **`chmod`**: UNIXのパーミッションモデル（所有者/グループ/その他）は単純ですが、WindowsのACL（アクセス制御リスト）はより細かく複雑な権限設定が可能です。移植時には、パーミッションの概念をACLにマッピングする必要があります。
- **`kill`**: UNIXのシグナル機構は、プロセス間通信と例外処理のための強力なツールです。Windowsにも類似の概念はありますが、`kill`のように他のプロセスに任意のシグナルを送る直接的な対応物はありません。プロセスの終了や通信には、異なるAPI（例：`TerminateProcess`）を使う必要があります。

30. ポータブルなオペレーティングシステムとは、あるシステムアーキテクチャから別のアーキテクチャへ何の変更もなしに移植できるものです。完全にポータブルなオペレーティングシステムを構築することがなぜ実行不可能であるかを説明しなさい。移植性の高いオペレーティングシステムを設計する際に持つであろう2つの高レベルなレイヤーを説明しなさい。

- **[解答訳]** すべてのシステムアーキテクチャには、実行できる独自の命令セットがあります。したがって、PentiumはSPARCプログラムを実行できず、SPARCはPentiumプログラムを実行できません。また、異なるアーキテクチャは、使用されるバスアーキテクチャ（VME, ISA, PCI, MCA, SBusなど）やCPUのワードサイズ（通常32ビットまたは64ビット）も異なります。これらのハードウェアの違いにより、完全にポータブルなオペレーティングシステムを構築することは不可能です。移植性の高いオペレーティングシステムは、2つの高レベルレイヤー、すなわちマシン依存レイヤーとマシン非依存レイヤーで構成されます。マシン依存レイヤーはハードウェアの仕様に対処し、すべてのアーキテクチャで個別に実装する必要があります。このレイヤーは、マシン非依存レイヤーが構築される統一されたインターフェースを提供します。マシン非依存レイヤーは一度だけ実装する必要があります。移植性を高めるためには、マシン依存レイヤーのサイズをできるだけ小さく保つ必要があります。

- **[より詳しい解説]** この解答は、移植性の高いOS設計の中心的な考え方である**HAL (Hardware Abstraction Layer)** の概念を説明しています。

  - **マシン依存レイヤー (HAL)**: CPUの命令セット、メモリ管理ユニット(MMU)の動作、割り込みコントローラ、バスアーキテクチャなど、ハードウェアに直接依存する部分をすべてこのレイヤーにカプセル化します。新しいハードウェアにOSを移植する際には、主にこのレイヤーを書き直すことになります。
  - **マシン非依存レイヤー**: プロセス管理、ファイルシステム、スケジューリングアルゴリズム、ネットワークプロトコルスタックなど、ハードウェアの具体的な仕様に直接依存しないOSの大部分です。このレイヤーは、HALが提供する標準化されたインターフェースの上で動作するため、一度書けばどのハードウェア上でも（理論的には）変更なしで動作します。 この構造により、移植作業はマシン依存レイヤーの再実装に限定され、OS全体の移植コストを大幅に削減できます。

31. ポリシーとメカニズムの分離が、マイクロカーネルベースのオペレーティングシステムの構築にどのように役立つか説明しなさい。

- **[解答訳]** ポリシーとメカニズムの分離により、OS設計者はカーネル内に少数の基本的なプリミティブを実装することができます。これらのプリミティブは、特定のポリシーに依存しないため、単純化されています。そして、これらのプリミティブを使って、より複雑なメカニズムやポリシーをユーザーレベルで実装することができます。

- **[より詳しい解説]** これはOS設計における重要な原則です。

- **メカニズム (Mechanism)**:「どのように行うか」という機能そのものを指します。例えば、「プロセスに優先度を割り当て、最も優先度の高いプロセスを実行する」という機能です。
- **ポリシー (Policy)**:「何をすべきか」という判断基準を指します。例えば、「どのプロセスにどの優先度を割り当てるか」という決定です。 **マイクロカーネル**は、カーネルの機能を最小限（プロセス間通信、基本的なスケジューリング、低レベルメモリ管理など）に絞り、ファイルシステムやデバイスドライバなどの多くの機能をユーザー空間のサーバプロセスとして実装する設計です。 この設計では、カーネルが提供するのは基本的な**メカニズム**のみです。例えば、カーネルはプロセスの切り替え機能を提供しますが、どのプロセスを次に実行するかという**ポリシー**は、ユーザー空間にあるスケジューリングサーバが決定します。これにより、カーネルは非常にシンプルで高信頼になり、ユーザーは異なるスケジューリングポリシーを実装したサーバを柔軟に選択できるようになります。

32. 仮想マシンはさまざまな理由で非常に人気が出ています。しかし、いくつかの欠点もあります。1つ挙げなさい。

- **[解答訳]** 仮想化レイヤーは、メモリ使用量とプロセッサのオーバーヘッドを増加させ、パフォーマンスのオーバーヘッドも増加させます。

- **[より詳しい解説]** 仮想マシン（VM）は、物理ハードウェア上にハイパーバイザと呼ばれるソフトウェア層を置き、その上に複数のゲストOSを動かす技術です。 解答が指摘するパフォーマンスのオーバーヘッドは、VMの主要な欠点です。具体的には、以下のような要因で発生します。

  - **CPUのオーバーヘッド**: ゲストOSが特権命令を実行しようとすると、ハイパーバイザへのトラップが発生し、ハイパーバイザがその命令をエミュレートする必要があります。このトラップとエミュレーションには時間がかかります。
  - **メモリのオーバーヘッド**: ハイパーバイザ自身がメモリを消費します。さらに、ゲストOSの物理メモリアドレスをホストの物理メモリアドレスに変換するための追加のページテーブル（シャドウページテーブルなど）が必要になり、メモリアクセスが遅くなる可能性があります。
  - **I/Oのオーバーヘッド**: ゲストOSからのI/O要求は、ハイパーバイザを介してホストOSまたは物理デバイスに渡されるため、直接アクセスするよりも遅延が大きくなります。

33. 単位変換を練習するための質問です：(a) 1ナノ年は何秒ですか。(b) マイクロメートルはしばしばミクロンと呼ばれます。1メガミクロンはどれくらいの長さですか。(c) 1ペタバイトのメモリには何バイトありますか。(d) 地球の質量は6000ヨタグラムです。これは何キログラムですか。

- **[解答訳]** 変換は単純です。
  a. ナノ年は 10^-9 × 365 × 24 × 3600 = 31.536 ミリ秒 です。
  b. 1メートル。
  c. 2^50バイト、つまり1,099,511,627,776バイトです。
  d. 6 × 10^24 kg、または 6 × 10^27 gです。
- **[より詳しい解説]** この問題は、教科書で使われるSI接頭語（metric prefix）の理解を確認するものです。
  a. 1ナノ年 = 10^-9 年 1年 ≈ 365.24日 × 24時間/日 × 3600秒/時間 ≈ 3.1557 × 10^7 秒 1ナノ年 ≈ 10^-9 × 3.1557 × 10^7 秒 ≈ 0.031557 秒 = 31.557 ミリ秒
  b. 1ミクロン = 1マイクロメートル = 10^-6 メートル 1メガミクロン = 10^6 ミクロン = 10^6 × 10^-6 メートル = 1 メートル
  c. コンピュータの分野では、キロ、メガ、ギガ、テラ、ペタは通常10のべき乗ではなく2のべき乗を指します。 1ペタバイト（PB）= 2^50 バイト (= 1024^5 バイト)
  d. ヨタ（Yotta）= 10^24 6000 ヨタグラム = 6000 × 10^24 グラム = 6 × 10^27 グラム 1キログラム = 1000 グラム = 10^3 グラム 6 × 10^27 グラム / (10^3 グラム/キログラム) = 6 × 10^24 キログラム

34. 図1-19に似ているが、実際に動作してテストできるだけのコードを含むシェルを作成しなさい。入力と出力のリダイレクション、パイプ、バックグラウンドジョブなどの機能を追加してもよいでしょう。

- **[解答訳]** （この問題はプログラミング演習のため、解答集に解答はありません）

- **[より詳しい解説]** この演習の目的は、fork, exec, waitpidといった基本的なプロセス管理のシステムコールと、open, close, dup2といったファイル管理のシステムコールを組み合わせて、OSの基本的な機能であるシェルを実装することです。

    - **基本ループ**: while(1)ループでプロンプトを表示し、コマンドラインを読み取ります。
    - **コマンド実行**: fork()で子プロセスを作成し、親プロセスはwaitpid()で子の終了を待ちます。子プロセスはexecve()でコマンドを実行します。
    - **リダイレクション (<, >)**: fork()とexecve()の間で、子プロセスがclose()で標準入出力（ファイルディスクリプタ0または1）を閉じ、open()で指定されたファイルを開きます。dup2()を使うとより簡単に実装できます。
    - **パイプ (|)**: pipe()システムコールでパイプを作成し、fork()を2回行います。一方の子プロセスの標準出力をパイプの書き込み側に、もう一方の子プロセスの標準入力をパイプの読み込み側にdup2()で接続します。
    - **バックグラウンドジョブ (&)**: コマンドの最後に&がある場合、親プロセスはwaitpid()を呼び出さずに、すぐに次のコマンドのプロンプトを表示します。 この演習を通じて、プロセスとファイルディスクリプタの概念がどのように連携して強力な機能を実現するかを実践的に学ぶことができます。

35. 安全にクラッシュさせて再起動できる個人用のUNIXライクなシステム（Linux、MINIX 3、FreeBSDなど）が利用できる場合、無限に子プロセスを作成しようとするシェルスクリプトを書き、何が起こるか観察しなさい。実験を実行する前に、シェルにsyncと入力してファイルシステムのバッファをディスクにフラッシュし、ファイルシステムを破壊しないようにしてください。仮想マシン内で安全に実験を行うこともできます。注意：共有システムでこれを試す場合は、まずシステム管理者の許可を得てください。結果はすぐに明らかになるため、捕まる可能性が高く、制裁が科されるかもしれません。

- **[解答訳]** （この問題は実験のため、解答集に解答はありません）

- **[より詳しい解説]** この実験は、一般に**フォーク爆弾 (fork bomb)** として知られる現象を観察するものです。以下のような簡単なシェルスクリプトで実現できます。 bash　　#!/bin/bash　　:(){ :|:& };: このスクリプトは、自身を再帰的に呼び出し、その出力をパイプで別の自分自身に渡し、バックグラウンドで実行します。これにより、プロセスが指数関数的に増加します。

    - **予想される結果**:
        1. システムが管理できるプロセス数には上限があり、これは**プロセステーブル**のサイズによって決まります。
        2. スクリプトを実行すると、新しいプロセスが急速に生成され、プロセステーブルはすぐに満杯になります。
        3. プロセステーブルが満杯になると、forkシステムコールは失敗し始めます。
        4. 大量のプロセスがメモリとCPU時間を奪い合うため、システムは極端に遅くなり、最終的には応答不能（フリーズ）状態に陥る可能性があります。新しいプロセス（loginプロセスさえも）を生成できなくなるため、管理者もログインして修復することが困難になります。
    - **syncの目的**: システムがクラッシュする前に、メモリ上のファイルシステムキャッシュ（バッファ）の内容をディスクに書き込むことで、ファイルシステムの破損を防ぎます。 この実験は、OSにおけるリソースの有限性と、それを使い果たすことの影響を実感するためのものです。

36. UNIXライクなシステムまたはWindowsディレクトリの内容を、UNIXのodプログラムのようなツールで調べて解釈してみてください。（ヒント：これをどのように行うかは、OSが何を許可するかに依存します。うまくいくかもしれない一つの方法は、あるオペレーティングシステムでUSBスティックにディレクトリを作成し、次にそのようなアクセスを許可する別のオペレーティングシステムを使用して生のデバイスデータを読み取ることです）。

- **[解答訳]** （この問題は実験のため、解答集に解答はありません）

- **[より詳しい解説]** この演習の目的は、ディレクトリが抽象化された概念であると同時に、ディスク上では特定のフォーマットを持つ単なる「ファイル」であることを理解することです。

- **実験方法**:
  1. USBメモリを特定のファイルシステム（例：FAT32やext2）でフォーマットします。
  2. いくつかのファイルとサブディレクトリを作成します。
  3. Linuxなどのシステムで、このUSBメモリをデバイスファイル（例：`/dev/sdb1`）として直接読み取ります。`od -c /dev/sdb1 | less` のようなコマンドを使うと、ディスクの生の内容を文字として表示できます。
- **予想される結果**:
  - ディスクの先頭には、ブートセクタやスーパーブロックのようなファイルシステムの管理情報が見つかります。
  - 次に、FAT（File Allocation Table）やi-nodeテーブルなど、ファイルのブロック配置を管理する領域が見られます。
  - そして、ディレクトリのエントリが見つかるはずです。FATファイルシステムでは、ディレクトリはファイル名、属性、開始ブロック番号などを含む32バイトのエントリのリストとして格納されています。`od`の出力の中に、作成したファイル名が（おそらく他のバイナリデータに混じって）見えるはずです。 この実験により、OSが提供する「ディレクトリ」というきれいな抽象化の裏側にある、ディスク上の生のデータ構造を垣間見ることができます。