第4章 ファイルシステム 演習問題の解答と解説

1. ファイル /etc/passwd に対する5つの異なるパス名を挙げなさい。（ヒント：ディレクトリのエントリ「.」と「..」について考えなさい。）

- **[解答訳]** カレントディレクトリが / (ルート) の場合: etc/passwd, /etc/passwd, ./etc/passwd カレントディレクトリが /etc の場合: passwd, ../etc/passwd この他にも多数の可能性があります。

- **[より詳しい解説]** この問題は、絶対パスと相対パス、そして特殊なディレクトリ名「.」(カレントディレクトリ)と「..」(親ディレクトリ)の理解を問うものです。以下に解答例をいくつか示します。

    - /etc/passwd (絶対パス)
    - /etc/../etc/passwd (一度親ディレクトリ/に戻ってから、再度/etcに入る)
    - /./etc/passwd (.はカレントディレクトリを指すので、ルートディレクトリからの相対パスになる)
    - もしカレントワーキングディレクトリが /usr であれば、../etc/passwd も有効なパスです。
    - もしカレントワーキングディレクトリが /etc であれば、単純に passwd と指定できます。

2. Windowsでは、ユーザーがWindowsエクスプローラにリストされたファイルをダブルクリックすると、プログラムが実行され、そのファイルがパラメータとして渡されます。オペレーティングシステムがどのプログラムを実行すべきかを知るための2つの異なる方法を挙げなさい。

- **[解答訳]** Windowsではファイル拡張子が使われます。各拡張子はファイルの種類と、それを扱うプログラムに対応しています。もう一つの方法は、ファイルを作成したプログラムを記憶しておき、そのプログラムを実行する方法です。Macintoshはこの方式を採用しています。

- **[より詳しい解説]**

    1. **ファイル拡張子による関連付け**: Windowsでは、ファイル拡張子（例：.docx, .pdf, .exe）が**レジストリ**に登録されており、各拡張子に対してどのプログラムを起動するかが関連付けられています。ユーザーがファイルをダブルクリックすると、OSはレジストリを参照して対応するプログラムを見つけ、そのプログラムにファイル名を引数として渡して起動します。
    2. **ファイルメタデータによる関連付け**: もう一つの方法は、ファイルのメタデータ（属性情報）内に、そのファイルを作成したアプリケーションの情報（クリエータコードなど）を記録しておく方式です。ファイルをダブルクリックすると、OSはこのメタデータを読み取り、指定されたアプリケーションを起動します。この方式は初期のMacintosh OSで採用されていました。

3. 初期のUNIXシステムでは、実行可能ファイル（a.outファイル）は非常に特定のマジックナンバーで始まっていました…。なぜ実行可能ファイルには非常に特定の番号が選ばれたのだと思いますか。

- **[解答訳]** これらのシステムはプログラムを直接メモリにロードし、ワード0から実行を開始しました。ヘッダをコードとして実行しようとするのを避けるため、マジックナンバーはヘッダをスキップするBRANCH命令でした。これにより、ヘッダのサイズを知らなくても、バイナリファイルを直接メモリに読み込んで実行することが可能でした。

- **[より詳しい解説]** 初期のUNIXシステムでは、実行可能ファイルの最初のワード（マジックナンバー）は、単なる識別子ではなく、**実行可能な機械語命令そのもの**でした。具体的には、ファイルのヘッダ部分を飛び越えて、実際のプログラムコードの開始点（テキストセグメントの先頭）へジャンプする**無条件分岐 (BRANCH) 命令**が格納されていました。 これにより、OSのローダは非常に単純な実装で済みました。ローダは実行可能ファイルを物理アドレス0番地からメモリにコピーし、CPUのプログラムカウンタを0にセットして実行を開始するだけです。CPUは最初にマジックナンバーである分岐命令を実行し、自動的にヘッダをスキップしてプログラム本体の実行を開始しました。他のファイルタイプにはこのような仕組みは不要なため、任意のランダムな値をマジックナンバーとして使用できました。

4. UNIXのopenシステムコールは絶対に不可欠ですか。それがない場合、どのような結果になりますか。

- **[解答訳]** openがなければ、すべてのreadコールでファイル名を指定する必要があります。システムはその後i-nodeを取得する必要がありますが、これはキャッシュされているかもしれません。i-nodeをディスクに書き戻すタイミングが問題になります。それはタイムアウトするかもしれません。これは少し不便ですが、機能する可能性があります。

- **[より詳しい解説]** openシステムコールは、ファイル名（パス名）を効率的な**ファイルディスクリプタ**に変換する重要な役割を担います。openがない場合、以下のような問題が生じます。

    - **性能の低下**: readやwriteのたびに、毎回ファイルへのフルパス名を指定する必要があります。OSはその都度、ルートディレクトリからパス名を解決し、目的のファイルのi-nodeを見つけなければなりません。これは非常に多くのディスクアクセスを伴い、非効率です。openは一度だけパス名を解決し、以降は高速なファイルディスクリプタでファイルにアクセスできるようにします。
    - **状態管理の困難さ**: 各プロセスはオープンしたファイルごとに「現在の読み書き位置（ファイルポインタ）」を保持しています。openがない場合、この状態をどこでどのように管理するかが問題になります。OSはプロセスごと、かつファイル名ごとにファイルポインタを管理する必要があり、実装が複雑になります。

5. シーケンシャルファイルをサポートするシステムには、常にファイルを巻き戻す操作があります。ランダムアクセスファイルをサポートするシステムも、これを必要としますか。

- **[解答訳]** いいえ。ファイルの先頭バイトにランダムアクセス（シーク）すれば、巻き戻し操作と同じことができます。

- **[より詳しい解説]** ランダムアクセスファイルは、ファイルの任意の位置に直接アクセスする機能を持っています。これは通常、seek（またはLinuxではlseek）システムコールによって実現されます。ファイルの先頭（バイト0）にシークする操作は、シーケンシャルファイルの「巻き戻し」操作と機能的に等価です。したがって、ランダムアクセスファイルをサポートするシステムでは、専用の巻き戻し操作は不要です。

6. renameシステムコールを使ってファイルの名前を変更することと、単に新しい名前で新しいファイルにコピーし、古いファイルを削除することとの間に何か違いはありますか。

- **[解答訳]** はい。renameコールは作成時刻や最終更新時刻を変更しませんが、新しいファイルを作成すると、作成時刻と最終更新時刻として現在の時刻が設定されます。また、ディスクがほぼ満杯の場合、コピーが失敗する可能性があります。

- **[より詳しい解説]** renameと「コピーして削除」には、以下の重要な違いがあります。

    - **メタデータの保持**: renameはディレクトリ内のファイル名エントリを変更するだけで、ファイルのi-node（所有者、パーミッション、タイムスタンプなど）は変更しません。一方、コピーすると新しいファイルが作成されるため、作成時刻は現在の時刻になり、所有者もコピーしたユーザーになります。
    - **効率性**: renameはi-nodeを共有する別のディレクトリにファイルを移動する場合でも、ディレクトリ情報を数バイト書き換えるだけで完了します。データブロックのコピーは発生しません。一方、コピーはファイルの全データブロックを読み書きする必要があり、特に大きなファイルでは非常に時間がかかります。
    - **ディスク容量**: コピー操作は、一時的にファイルのコピーを保持するための追加のディスクスペースを必要とします。ディスクの空き容量が少ない場合、コピーは失敗する可能性がありますが、renameはデータブロックを消費しないため成功します。

7. 一部のシステムでは、ファイルの一部をメモリにマップすることが可能です。そのようなシステムはどのような制限を課さなければなりませんか。この部分的なマッピングはどのように実装されますか。

- **[解答訳]** マップされるファイルの開始点はページの境界でなければならず、長さもページの整数倍でなければなりません。各マップされたページは、ファイル自体をバッキングストアとして使用します。マップされてい

ないメモリは、スクラッチファイルやパーティションをバッキングストアとして使用します。

- **[より詳しい解説]** メモリマップドファイルは、ファイルの内容をプロセスの仮想アドレス空間に直接マッピングする機能です。

    - **制限**: OSとMMUはページ単位でメモリを管理するため、マッピングの開始アドレスは**ページサイズの倍数**でなければならず、マッピングされる領域の長さも**ページサイズの整数倍**に切り上げられます。
    - **実装**: プロセスがメモリマップドファイルのためのシステムコール（例：`mmap`）を発行すると、カーネルはそのプロセスのページテーブルに新しいエントリを作成します。このエントリは、プロセスの仮想アドレス空間の特定のページを、ディスク上のファイルの特定のブロックに対応付けます。ただし、この時点では物理メモリには何もロードされません（デマンドページング）。プロセスがそのマッピングされたアドレスに初めてアクセスすると**ページフォールト**が発生し、OSがディスクから対応するファイルブロックを物理メモリに読み込み、ページテーブルを更新してアクセスを完了させます。

8. ある単純なオペレーティングシステムは単一のディレクトリしかサポートしませんが、任意に長いファイル名を持つファイルを任意に多数持つことを許可します。階層的なファイルシステムに近似するものをシミュレートできますか。どのようにしますか。

- **[解答訳]** `/usr/ast/file` のようなファイル名を使用します。これは階層的なパス名のように見えますが、実際にはスラッシュが埋め込まれた単一の名前です。

- **[より詳しい解説]** このシミュレーションは、**ファイル名の命名規則**によって実現します。OS自体はファイル名を単なる文字列としてしか認識しませんが、ユーザーやアプリケーションはファイル名に `/` のような区切り文字を使い、それを階層構造として解釈します。

  例えば、`usr_ast_file` や `C:\Users\ast\file` といったファイル名は、この単純なOSにとっては単なる長い名前です。しかし、これを扱うプログラム（例えば、カスタムされたシェルなど）が区切り文字を解釈することで、あたかも `/usr/ast/` というディレクトリに `file` が存在するかのうように振る舞うことができます。ただし、`cd` のようなディレクトリ移動の概念はOSレベルではサポートされないため、そのカスタムプログラム内でシミュレートする必要があります。

9. UNIXとWindowsでは、ランダムアクセスは、ファイルに関連付けられた「現在位置」ポインタをファイル内の指定されたバイトに移動させる特別なシステムコールを持つことによって行われます。このシステムコールなしでランダムアクセスを行う代替方法を提案しなさい。

- **[解答訳]** `read` システムコールに、読み込みを開始するアドレスを指定するパラメータを一つ追加する方法があります。これにより、事実上すべての `read` がファイル内のシークを伴う可能性を持ちます。

- **[より詳しい解説]** この代替案は、`read` システムコール自体にランダムアクセス機能を持たせるものです。現在の `read(fd, buffer, nbytes)` に加えて、`read(fd, buffer, nbytes, position)` のような形式にします。

    - **利点**: `lseek` システムコールが不要になり、APIが少しだけ単純になります。
    - **欠点**:
        1. すべての `read` 呼び出しに余分なパラメータが必要になります。
        2. **ファイルポインタの状態管理**というOSが担っていた重要な役割を、ユーザープログラム自身が担うことになります。現在の読み書き位置を常にアプリケーション側で追跡する必要があるため、プログラミングが煩雑になり、バグも発生しやすくなります。 このため、現在のようにファイルポインタを管理する `lseek` と、そのポインタから読み書きする `read`/`write` を分離する方式が一般的となっています。

10. 図4-8のディレクトリツリーを考えます。もし `/usr/jim` がワーキングディレクトリである場合、相対パス名が `../ast/x` であるファイルの絶対パス名は何ですか。

- **[解答訳]** `..` の部分は検索を `/usr` に移動させ、`../ast` は `/usr/ast` になります。したがって、`../ast/x` は

/usr/ast/x と同じです。

- **[より詳しい解説]** この問題は、相対パスにおける特殊なディレクトリ名「..」（親ディレクトリを指す）の解釈を問うています。

    1. 現在のワーキングディレクトリは /usr/jim です。
    2. パス ../ast/x の最初の構成要素 .. は、親ディレクトリを意味します。/usr/jim の親ディレクトリは /usr です。したがって、ここまででパスは /usr を指します。
    3. 次の構成要素 ast は、/usr ディレクトリ内のサブディレクトリを指します。したがって、パスは /usr/ast となります。
    4. 最後の構成要素 x は、/usr/ast ディレクトリ内のファイルを指します。 以上より、相対パス ../ast/x の絶対パスは **/usr/ast/x** となります。

11. 本文で述べたように、ファイルの連続割り当てはディスクのフラグメンテーションを引き起こします。なぜなら、ファイルの長さがブロックの整数倍でない場合、最後のディスクブロックに無駄なスペースが生じるからです。これは内部フラグメンテーションですか、それとも外部フラグメンテーションですか。前の章で議論した何かと類推しなさい。

- **[解答訳]** これは**内部フラグメンテーション**です。割り当て単位（ファイル）の間に存在するのではなく、その内部にあるためです。これは、プロセスの最後のページでスペースが無駄になるのと全く同じです。

- **[より詳しい解説]**

    - **内部フラグメンテーション**: メモリやディスクを固定サイズのブロックで割り当てた際に、**割り当てられたブロックの内部に生じる未使用領域**のことです。例えば、4KBのディスクブロックに1KBのファイルを保存すると、3KBの領域がそのブロック内で無駄になります。
    - **外部フラグメンテーション**: メモリやディスク上に、**割り当てられた領域の間に点在する、どのプロセスにも割り当てられていない小さな空き領域**のことです。 問題のケースは、ファイルの最後のブロック内でデータが占めていない部分に関するものです。この未使用領域はファイルに割り当てられたブロックの「内部」にあるため、**内部フラグメンテーション**に分類されます。これは、ページング方式のメモリ管理において、プロセスの最後のページで発生する無駄な領域と同じ概念です。

12. 与えられたファイルに対する破損したデータブロックの影響を、（a）連続割り当て、（b）リンク割り当て、および（c）インデックス（またはテーブルベース）割り当てについて説明しなさい。

- **[解答訳]**
    a. **連続割り当て**: 破損したブロックは読み取れませんが、ファイルの残りの部分は正常です。
    b. **リンク割り当て**: 破損したブロックのデータが失われるだけでなく、そのブロックに含まれていた次のブロックへのポインタも失われます。これにより、ファイルの残りの部分全体が失われます。
    c. **インデックス割り当て**: 破損したブロックのデータのみが失われます。i-node（またはFAT）から他のブロックのアドレスは見つけることができます。
- **[より詳しい解説]** この問題は、各割り当て方式の耐障害性を比較するものです。
    - **(a) 連続割り当て**: ファイルの開始ブロックアドレスと長ささえわかれば、他のブロックの位置を計算できます。したがって、一つのブロックが破損しても、他のブロックへのアクセスには影響しません。
    - **(b) リンク割り当て**: 各ブロックが次のブロックへのポインタを持つため、鎖のようにつながっています。一つのブロックが破損してポインタが読み取れなくなると、その鎖が途切れてしまい、それ以降のすべてのブロックへのアクセス手段が失われます。
    - **(c) インデックス（テーブル）割り当て**: i-nodeやFATのような中央のテーブルに全ブロックへのポインタが格納されています。したがって、一つのデータブロックが破損しても、他のブロックのアドレスはテーブルから読み取れるため、影響は破損したブロックだけに限定されます。この方式が最も堅牢であると言えます。

13. ディスクの連続割り当てを使用し、ホールに悩まされない一つの方法は、ファイルが削除されるたびにディスク

をコンパクションすることです。…16GBのディスクの半分をコンパクションするのにどれくらいの時間がかかりますか。

- **[解答訳]** 1回の転送を開始するのに9ミリ秒かかります。80MB/秒の転送レートで8192バイトを読み取るには、0.0977ミリ秒、合計で9.0977ミリ秒かかります。それを書き戻すにはさらに9.0977ミリ秒が必要です。したがって、1つのファイルをコピーするには18.1954ミリ秒かかります。16GBのディスクの半分である8GBのストレージをコンパクションするには、2^20個のファイルをコピーする必要があります。ファイルあたり18.1954ミリ秒で計算すると、19,079.25秒、つまり5.3時間かかります。明らかに、ファイルが削除されるたびにディスクをコンパクションするのは良い考えではありません。

- **[より詳しい解説]** この問題はディスクコンパクションの現実的なコストを計算させるものです。

    1. **1ファイルの移動時間**:
        - 読み込み時間 = シーク(5ms) + 回転遅延(4ms) + 転送時間(8KB / 80MB/s = 0.1ms) = 9.1ms
        - 書き込み時間 = シーク(5ms) + 回転遅延(4ms) + 転送時間(0.1ms) = 9.1ms
        - 合計 = `9.1 + 9.1 = 18.2ms` （解答集の計算とほぼ同じ）
    2. **コンパクション対象のファイル数**:
        - 対象データ量: 8GB = `8 * 1024 * 1024` KB
        - 平均ファイルサイズ: 8KB
        - ファイル数 = `8 * 1024 * 1024 / 8 = 1,048,576` ファイル (2^20)
    3. **総コンパクション時間**:
        - 総時間 = `1,048,576 ファイル * 18.2 ms/ファイル ≈ 19,086 秒 ≈ 5.3時間` この計算結果が示すように、ファイル削除のたびにディスク全体をコンパクションするのは非現実的なほど時間がかかります。

14. 前の問題の答えに照らして、ディスクのコンパクションは意味がありますか。

- **[解答訳]** 正しく行えば、はい。ファイルを連続的にするように再編成することで、性能を向上させることができます。Windowsにはディスクをデフラグし、再編成するプログラムがあります。ユーザーは定期的にそれを実行することが推奨されます。しかし、それがどれほど時間がかかるかを考えると、月に一度実行するのが良い頻度かもしれません。

- **[より詳しい解説]** 前問で見たように、ファイル削除のたびにコンパクションを行うのは現実的ではありません。しかし、**定期的なデフラグメンテーション**（断片化解消）として行うのであれば、大いに意味があります。

    - **利点**: ファイルが物理的に連続したブロックに配置されることで、ファイルの読み込みに必要なシーク回数が減り、アクセス性能が大幅に向上します。
    - **欠点**: 処理に非常に時間がかかります。 結論として、ディスクのコンパクションは、その実行頻度とタイミングが重要です。Windowsのdefragのように、システムのアイドル時間を利用してバックグラウンドで定期的に実行するのが、性能とオーバーヘッドのバランスをとる現実的なアプローチです。

15. 一部のデジタル家電は、例えばファイルとしてデータを保存する必要があります。ファイルの保存が必要で、連続割り当てが優れたアイデアとなる現代のデバイスを挙げなさい。

- **[解答訳]** デジタルスチルカメラは、一連の写真を不揮発性ストレージ媒体（例：フラッシュメモリ）に記録します。カメラがリセットされると媒体は空になります。その後、写真は媒体が満杯になるまで順番に一枚ずつ記録され、満杯になるとハードディスクにアップロードされます。このアプリケーションには、カメラ内部（写真ストレージ媒体上）の連続ファイルシステムが理想的です。

- **[より詳しい解説]** 連続割り当ては、**ファイルのサイズが作成後に変化せず、ファイルが順番に書き込まれる**ような用途で非常に有効です。デジタルカメラ以外にも、以下のようなデバイスが考えられます。

    - **DVD/Blu-rayレコーダー**: テレビ番組を録画する場合、ファイルサイズは録画開始時には不明ですが、一

度録画が完了すればファイルサイズは固定されます。録画中は連続した領域に書き込み続けるため、連続割り当ては効率的です。
    - **音楽プレイヤー**: 音楽ファイルは一度デバイスに転送されたら、そのサイズは変わりません。 これらのデバイスでは、ファイルの断片化が問題になる前にストレージ全体がフォーマットされることが多いため、連続割り当ての欠点が表面化しにくいのです。

16. 図4-13に示されているiノードを考えます。もしそれが10個のダイレクトアドレスを含み、これらがそれぞれ8バイトで、すべてのディスクブロックが1024KBだった場合、可能な最大のファイルは何ですか。

- **[解答訳]** 間接ブロックは128個のディスクアドレスを保持できます。10個の直接ディスクアドレスと合わせて、最大ファイルは138ブロックになります。各ブロックが1KBなので、最大ファイルは138KBです。 *（注：解答集はブロックサイズを1KBと解釈して計算しています。質問文の1024KBは一般的でないため、ここでは解答集の解釈に従います。）*

- **[より詳しい解説]** この問題は、i-nodeの構造から最大ファイルサイズを計算するものです。

    - **前提**:
        - 直接アドレス数: 10
        - 間接ブロックポインタ数: 1
        - ディスクアドレスのサイズ: 8バイト
        - ディスクブロックサイズ: 1KB = 1024バイト（解答集の解釈）
    - **計算**:
        1. **直接ブロック**: 10ブロック * 1 KB/ブロック = 10 KB
        2. **間接ブロック**: 1つの間接ブロックには 1024バイト / 8バイト/アドレス = 128 個のディスクアドレスが格納できます。
        3. 間接ブロックが指すデータ量: 128ブロック * 1 KB/ブロック = 128 KB
        4. **最大ファイルサイズ**: 10 KB（直接）+ 128 KB（間接）= 138 KB もし問題文の1024KBブロックサイズを文字通り解釈すると、間接ブロックには 1024 KB / 8バイト = 131,072 個のアドレスが格納でき、最大ファイルサイズは 10 * 1024KB + 131,072 * 1024KB となり、約128GBという巨大な値になります。

17. あるクラスについて、学生の記録がファイルに保存されています。記録はランダムにアクセスされ、更新されます。各学生の記録が固定サイズであると仮定します。3つの割り当て方式（連続、リンク、テーブル/インデックス）のうち、どれが最も適切ですか。

- **[解答訳]** ランダムアクセスには、テーブル/インデックス方式と連続割り当て方式が適しています。リンク割り当ては、特定のレコードに対して複数のディスク読み込みを必要とするため適していません。

- **[より詳しい解説]** 各方式のランダムアクセス性能を比較します。

    - **連続割り当て**: 記録が固定サイズなので、n番目の記録のディスクアドレスは (開始アドレス) + (n * 記録サイズ) で簡単に計算できます。非常に高速です。
    - **リンク割り当て**: n番目の記録にアクセスするには、ファイルの先頭から n-1 個のブロックを順にたどる必要があり、非常に低速です。
    - **テーブル/インデックス割り当て (i-node/FAT)**: n番目の記録がどのブロックにあるかを計算し、そのブロックのアドレスをテーブルから直接取得できます。高速です。 レコードサイズが固定で更新がある場合、連続割り当てはファイルの拡張が難しいため、**テーブル/インデックス割り当てが最も適切**と言えるでしょう。

18. その寿命の間にサイズが4KBから4MBの間で変動するファイルを考えます。3つの割り当て方式（連続、リンク、テーブル/インデックス）のうち、どれが最も適切ですか。

- **[解答訳]** ファイルサイズが大きく変動するため、連続割り当ては非効率です。ファイルが大きくなるにつれてディスクスペースの再割り当てが必要になり、ファイルが小さくなるにつれて空きブロックのコンパクション

が必要になります。リンク割り当てとテーブル/インデックス割り当てはどちらも効率的です。この2つのうち、ランダムアクセスのシナリオではテーブル/インデックス割り当てがより効率的です。

- **[より詳しい解説]** ファイルサイズの動的な変化に対応できるかが鍵となります。
    - **連続割り当て**: ファイルが大きくなるたびに、より大きな連続した空き領域を探してファイル全体を移動させる必要があり、非常に非効率です。外部フラグメンテーションも発生しやすくなります。
    - **リンク割り当てとテーブル/インデックス割り当て**: どちらもブロック単位で領域を確保・解放するため、ファイルサイズの変動に柔軟に対応できます。必要なブロックを空き領域から取り、リストやテーブルにつなぐだけです。 したがって、このケースでは**リンク割り当て**または**テーブル/インデックス割り当て**が適しています。特にランダムアクセスが必要な場合は、後者が優れています。

19. iノード内に短いファイルのデータを保存することで、効率を改善しディスクスペースを節約できると提案されています。図4-13のiノードの場合、iノード内に何バイトのデータを保存できますか。

- **[解答訳]** アドレスブロックポインタがデータを保持していることを示す何らかの方法が必要です。もし属性の中に未使用のビットが残っていれば、それを利用することができます。この場合、9つ全てのポインタをデータのために使用できます。ポインタがそれぞれkバイトであれば、格納できるファイルは最大9kバイト長になります。もし属性に未使用のビットがなければ、最初のディスクアドレスに無効なアドレスを格納し、それに続くバイトがポインタではなくデータであることを示すマーカーとして使用できます。この場合、最大ファイル長は8kバイトです。

- **[より詳しい解説]** この問題は、iノードの構造を最適化するアイデアについての考察を促すものです。教科書の図4-13 に示されているiノードには、属性情報とディスクアドレスを格納する領域があります。
    - **基本的なアイデア**: ファイルが非常に小さい場合、データブロックへのポインタを格納する代わりに、その領域に**ファイルデータそのもの**を格納してしまえば、データブロックを別途確保する必要がなくなり、ディスクアクセスの回数も減らせます（iノードの読み込みだけでファイル全体が読み込めるため）。
    - **実装上の課題**: i-nodeのディスクアドレス領域が「ポインタ」を保持しているのか「データ」を保持しているのかを区別する必要があります。
        1. **属性ビットを使用する方法**: i-nodeの属性フィールドに未使用のビットがあれば、それを「データ直接格納フラグ」として利用できます。このフラグがONであれば、ディスクアドレス領域はデータとして解釈されます。図4-13 のi-nodeには9つのディスクアドレス領域があるため、1つのアドレスがkバイトであれば、最大で 9 * k バイトのデータを格納できます。
        2. **無効アドレスを使用する方法**: 未使用ビットがない場合、最初のディスクアドレス領域に「無効なディスクアドレス」（例えば-1など）を格納し、これがデータ直接格納のマーカーであると定めます。この場合、マーカーとして1つのアドレス領域を使用するため、残りの8つの領域にデータを格納できます。最大ファイルサイズは 8 * k バイトとなります。 この最適化は**Immediate File**と呼ばれ、非常に小さいファイルに対するアクセス効率を大幅に向上させます。

20. 2人のコンピュータ科学の学生、キャロラインとエリナーがiノードについて議論しています。キャロラインは、メモリが非常に大きく安価になったため、ファイルが開かれるときに、iノードテーブル全体を検索して既に存在するかどうかを確認するよりも、新しいiノードのコピーをiノードテーブルにフェッチする方が単純で速いと主張しています。エリナーは反対です。どちらが正しいですか。

- **[解答訳]** エリナーが正しいです。i-nodeのコピーが2つテーブルに同時に存在することは、両方が読み取り専用でない限り、大惨事につながります。最悪のケースは、両方が同時に更新される場合です。i-nodeがディスクに書き戻されるとき、最後に書き込まれたものがもう一方による変更を消去してしまい、ディスクブロックが失われます。

- **[より詳しい解説]** この問題の核心は**ファイルシステムの一貫性(consistency)** [4.4.3] です。
    - **キャロラインの主張の問題点**: ファイルを開くたびにディスクからi-nodeの新しいコピーをメモリ上のi-nodeテーブルにロードすると、同じファイルに対して複数のi-nodeのコピーがメモリ上に存在し得ます。

- **なぜ危険か**: 2つのプロセスが同じファイルを同時に開き、それぞれが別々のi-nodeコピーを持つとします。プロセスAがファイルにブロックを追加すると、メモリ上のAのi-nodeコピーが更新されます。同時に、プロセスBもファイルにブロックを追加すると、Bのi-nodeコピーが更新されます。その後、Aがファイルを閉じるとAのi-nodeがディスクに書き戻されます。次にBがファイルを閉じるとBのi-nodeがディスクに書き戻され、**Aが行った変更（ブロックの追加）は上書きされて失われます**。結果として、ファイルシステムは矛盾した状態になり、データが失われます。
- **エリナーの正しい主張**: このような事態を防ぐため、OSはファイルが開かれる際に、まずメモリ上のi-nodeテーブルを検索し、対象のi-nodeが既に存在するかを確認しなければなりません。もし存在すれば、その既存のi-nodeの参照カウントを増やすだけで、新しいコピーは作りません。これにより、メモリ上には常に**1つのファイルに対して1つのi-nodeコピー**しか存在しないことが保証され、一貫性が保たれます。

21. ハードリンクがシンボリックリンクより優れている点を1つ、シンボリックリンクがハードリンクより優れている点を1つ挙げなさい。

- **[解答訳]** ハードリンクは余分なディスクスペースを必要としませんが、シンボリックリンクはリンク先のファイル名を格納するためのスペースが必要です。シンボリックリンクは、インターネットを介して他のマシンのファイルを指すことができます。ハードリンクは、自身のパーティション内のファイルにしかリンクできません。

- **[より詳しい解説]** ハードリンクとシンボリックリンクは、ファイルを共有するための異なるメカニズムです[4.3.4]。

  - **ハードリンクの利点**: ハードリンクは、単に既存のi-nodeを指す新しいディレクトリントリを作成するだけです。そのため、i-nodeやデータブロックといった追加のディスク領域を消費しません。i-node内のリンクカウントがインクリメントされるだけです。
  - **シンボリックリンクの利点**: シンボリックリンクは、リンク先の**パス名**を内容として持つ特殊なファイルです。パス名で指定するため、ファイルシステムの境界（パーティション）を越えたり、ネットワークを介してリモートマシン上のファイルを指したりすることも可能です。一方、ハードリンクはi-node番号でファイルを指しますが、i-node番号は単一のファイルシステム内でのみ有効なため、パーティションを越えることはできません。

22. ハードリンクとソフトリンクがiノードの割り当てに関してどのように異なるか説明しなさい。

- **[解答訳]** ハードリンクは、あるファイルに対する複数のディレクトリントリが同じi-nodeを指すものです。ソフトリンクの場合、リンク自体に新しいi-nodeが作成され、そのi-nodeがリンク先のファイルを指します。

- **[より詳しい解説]**

  - **ハードリンク**: 新しいハードリンクを作成する際、**新しいi-nodeは割り当てられません**。既存のファイルのi-node番号を使い、新しいディレクトリントリが作られるだけです。結果として、複数のディレクトリントリが単一のi-nodeを共有します。
  - **ソフト（シンボリック）リンク**: 新しいシンボリックリンクを作成する際、それは新しいファイルとして扱われます。そのため、**新しいi-nodeが割り当てられ**、さらにリンク先のパス名を格納するためのデータブロックも確保されます。この新しいi-nodeは、ファイルタイプが「シンボリックリンク」であることを示します。

23. 4KBのブロックとフリーリスト方式を使用する4TBのディスクを考えます。1つのブロックにいくつのブロックアドレスを保存できますか。

- **[解答訳]** ディスクのブロック数は 4TB / 4KB = 2^30 です。したがって、各ブロックアドレスは32ビット（4バイト）で、最も近い2のべき乗になります。各ブロックは 4KB / 4 = 1024個のアドレスを格納できます。

- **[より詳しい解説]**

1. **総ブロック数の計算**:
    - ディスクサイズ: 4TB = 4 * 2^40 バイト = 2^42 バイト
    - ブロックサイズ: 4KB = 4 * 2^10 バイト = 2^12 バイト
    - 総ブロック数 = ディスクサイズ / ブロックサイズ = 2^42 / 2^12 = 2^30 ブロック
2. **アドレスサイズの決定**:
    - 2^30個のブロックを区別するためには、最低でも30ビットのアドレスが必要です。通常、アドレスはバイト境界に沿って割り当てられるため、32ビット（4バイト）のアドレスが使用されます。
3. **1ブロックあたりのアドレス格納数**:
    - 1ブロックのサイズは4KB = 4096バイトです。
    - 1アドレスのサイズは4バイトです。
    - したがって、1ブロックに格納できるアドレス数は 4096バイト / 4バイト/アドレス = 1024 アドレスとなります。

24. 空きディスクスペースは、フリーリストまたはビットマップを使用して追跡できます。…フリーリストがビットマップより少ないスペースを使用する条件を述べなさい。…

- **[解答訳]** ビットマップはBビットを必要とします。フリーリストはDFビットを必要とします。フリーリストがより少ないビットを使用するのは `DF < B` の場合です。あるいは、フリーリストが短いのは `F/B < 1/D` の場合です。ここで F/B は空きブロックの割合です。16ビットのディスクアドレスの場合、ディスクの空き領域が6%以下であればフリーリストの方が短くなります。

- **[より詳しい解説]** この問題は、2つの空き領域管理手法のストレージ効率を比較するものです [4.4.1]。

    - **ビットマップのサイズ**: ディスク上の全ブロックBに対して、1ブロックあたり1ビットが必要です。したがって、合計で **Bビット** が必要です。
    - **フリーリストのサイズ**: 空きブロックFのそれぞれに対して、そのアドレス（Dビット長）を格納する必要があります。したがって、合計で **F * Dビット** が必要です。
    - **比較**: フリーリストの方が少ないスペースで済む条件は、`F * D < B` です。
    - この式を変形すると、`F / B < 1 / D` となります。F/Bはディスク全体における空きブロックの割合を示します。
    - **D=16ビットの場合**: `F / B < 1 / 16` となります。`1/16 = 0.0625` なので、ディスクの**空き領域が6.25%未満**の場合に、フリーリストの方がビットマップよりも少ないスペースで済みます。ディスクがほとんど満杯の状態ではフリーリストが効率的ですが、そうでなければビットマップの方が効率的です。

25. ディスクパーティションが最初にフォーマットされた後の空き領域ビットマップの始まりは…以下の追加アクションのそれぞれの後のビットマップを示しなさい…

- **[解答訳]** ビットマップの始まりは以下のようになります。
    a. ファイルB書き込み後: `1111 1111 1111 0000`
    b. ファイルA削除後: `1000 0001 1111 0000`
    c. ファイルC書き込み後: `1111 1111 1111 1100`
    d. ファイルB削除後: `1111 1110 0000 1100`
- **[より詳しい解説]** システムは常に最も低い番号の空きブロックから検索するというルールに従って、ビットマップ（1が使用中、0が空き）の変化を追跡します。
    - **初期状態**: `1000 0000 0000 0000` (ブロック0が使用中)
    - **ファイルA (6ブロック)書き込み後**: `1111 1110 0000 0000`
    - **(a) ファイルB (5ブロック)書き込み後**:
        - 空いている最も低いブロックは7番からです。ブロック7, 8, 9, 10, 11を使用します。
        - ビットマップ: `1111 1111 1111 0000`
    - **(b) ファイルA (ブロック1-6)削除後**:
        - ブロック1から6までを空き(0)にします。
        - ビットマップ: `1000 0001 1111 0000`

- **(c) ファイルC (8ブロック)書き込み後**:
  - 最も低い空きブロックは1番からです。ブロック1から6まで（6ブロック）と、ブロック12から13まで（2ブロック）を使用します。
  - ビットマップ: 1111 1111 1111 1100
- **(d) ファイルB (ブロック7-11)削除後**:
  - ブロック7から11までを空き(0)にします。
  - ビットマップ: 1111 1110 0000 1100

26. 空きディスクブロックに関する情報を含むビットマップまたはフリーリストがクラッシュにより完全に失われた場合、何が起こりますか。この災害から回復する方法はありますか…

- **[解答訳]** これは全く深刻な問題ではありません。修復は簡単ですが、時間がかかります。回復アルゴリズムは、全ファイルの全ブロックのリストを作成し、その補集合を新しいフリーリストとしてとることです。UNIXでは、全i-nodeをスキャンすることでこれを実現できます。FATファイルシステムでは、フリーリストがないためこの問題は起こりえません。しかし、もしあったとしても、FATをスキャンして空きエントリを探すだけで回復できます。

- **[より詳しい解説]** ファイルシステムの一貫性チェックユーティリティ（UNIXのfsckやWindowsのchkdskなど）がこの問題を解決します [4.4.3]。

  - **回復プロセス**:
    1. ファイルシステムチェッカーは、ルートディレクトリから始めてファイルシステム全体を走査します。
    2. 全てのディレクトリをたどり、全てのファイルに対応するi-node（またはFATエントリ）を読み取ります。
    3. 各i-nodeから、そのファイルが使用している全データブロックのリストを構築します。
    4. 全てのファイルについてこの作業を行い、**「現在使用中の全ブロック」のリスト**を作成します。
    5. ディスク上の全ブロックのリストから、この「使用中の全ブロック」リストを引くことで、**「空いているはずの全ブロック」のリスト**を再構築できます。
    6. この新しいリストを使って、ビットマップまたはフリーリストを再作成します。
  - **FAT-16の場合**: FAT-16には独立したフリーリストやビットマップは存在しません [4.5.1]。空きブロックの情報はFATテーブル自体に埋め込まれています（特定の値でマークされている）。したがって、この問題は発生しません。

27. オリバー・オウルの大学計算センターでの夜の仕事は、…バックアップ中のシステムでテキストプロセッサを実行します。この配置に問題はありますか。

- **[解答訳]** オリーの論文は、彼が望むほど確実にはバックアップされないかもしれません。バックアッププログラムは、書き込みのために開かれているファイルをスキップすることがあります。なぜなら、そのようなファイル内のデータの状態は不確定かもしれないからです。

- **[より詳しい解説]** これは**アクティブなファイルシステムのバックアップ**の問題点を示しています [4.4.2]。

  - **一貫性の問題**: バックアッププログラムがファイルをコピーしている最中に、オリバーのテキストプロセッサがそのファイルに書き込みを行うと、バックアップされたファイルは中途半端な状態（一部は更新前、一部は更新後）になる可能性があります。これは破損したバックアップにつながります。
  - **ファイルのスキップ**: このような一貫性のないバックアップを避けるため、多くのバックアップユーティリティは、現在書き込みのために開かれているファイル（ロックされているファイル）を**スキップ**するように設計されています。
  - **結論**: オリバーが論文を編集中（ファイルが開かれている状態）にバックアップが実行されると、そのファイルはバックアップ対象から外される可能性が高いです。したがって、この作業方法では、彼の論文はバックアップされないかもしれません。

28. Windowsでは、すべてのファイルにアーカイブビットがあるため、いつダンプするかを判断するのは簡単です。このビットはUNIXにはありません。UNIXのバックアッププログラムはどのファイルをダンプすべきかをどのように知りますか。

- **[解答訳]** 彼らはディスク上のファイルに最後のダンプ時刻を記録しておく必要があります。毎回のダンプで、このファイルにエントリが追加されます。ダンプ時には、ファイルが読み込まれ、最後のダンプ時刻が記録されます。その時刻以降に変更されたファイルは全てダンプされます。

- **[より詳しい解説]** UNIXのインクリメンタルバックアップは、ファイルの**最終更新時刻 (mtime)** を利用します [4.4.2]。

    1. バックアッププログラムは、前回バックアップを実行した時刻（例えば、`2023-10-27 22:00:00`）を記録しておきます。
    2. 次回のバックアップ時には、ファイルシステム上の全てのファイルを走査します。
    3. 各ファイルのi-nodeに記録されている最終更新時刻 (mtime) を、前回のバックアップ時刻と比較します。
    4. もしファイルの `mtime` が前回のバックアップ時刻よりも新しければ、そのファイルは変更されたと判断し、ダンプの対象とします。 この方法により、アーカイブビットがなくても、どのファイルをバックアップすべきかを確実に判断できます。

29. 図4-25のファイル21が最後のダンプ以降変更されていないと仮定します。図4-26の4つのビットマップはどのように異なりますか。

- **[解答訳]** (a)と(b)では、21はマークされません。(c)では変更はありません。(d)では、21はマークされません。

- **[より詳しい解説]** 論理ダンプアルゴリズムの各フェーズがどのように変化するかを見ていきます [4.4.2]。

    - **フェーズ1（図4-26(a)）**: このフェーズでは、変更されたファイルと全てのディレクトリをマークします。ファイル21が変更されていないので、i-node 21は**マークされません**。
    - **フェーズ2（図4-26(b)）**: このフェーズでは、配下に変更されたファイル/ディレクトリを持たないディレクトリのマークを外します。ファイル21がマークされていないため、親ディレクトリ16の配下には変更されたものがなくなります。しかし、ディレクトリ16は、さらにその親であるディレクトリ6の配下にあるファイル17（変更あり）へのパス上にあるため、**マークされたまま**です。（**解答集の「(b)では21はマークされない」は自明ですが、このフェーズでの本質的な変化は、もし21がディレクトリ16配下の唯一の変更点だった場合にディレクトリ16のマークが外れる可能性がある点です。この例ではファイル17もあるため変化はありません**）。
    - **フェーズ3（図4-26(c)）**: ダンプ対象のディレクトリをダンプします。ファイル21はディレクトリではないので、このフェーズには影響しません。
    - **フェーズ4（図4-26(d)）**: ダンプ対象のファイルをダンプします。ファイル21は変更されていないため、ダンプ対象ではなく、ビットマップ上でも**マークされません**。

30. 各UNIXファイルの最初の部分をそのiノードと同じディスクブロックに保持することが提案されています。これにはどのような利点がありますか。

- **[解答訳]** 多くのUNIXファイルは短いです。もしファイル全体がi-nodeと同じブロックに収まるなら、ファイルを読み込むのに必要なディスクアクセスは2回ではなく1回で済みます。より長いファイルでも、1回少ないディスクアクセスで済むという利点があります。

- **[より詳しい解説]** これは、**ディスクアームの動きを減らす**ための性能最適化のアイデアです [4.4.4]。

    - **通常のファイル読み込み**: 通常、短いファイルを読み込むには最低でも2回のディスクアクセスが必要です。
        1. i-nodeを読み込むためのアクセス。
        2. i-node内のアドレスを使って、最初のデータブロックを読み込むためのアクセス。
    - **提案手法の利点**: i-nodeとファイルの先頭部分を同じディスクブロックに格納すれば、**1回のディスクアク**

セスでi-nodeとファイルの先頭データの両方を取得できます。
    - **効果**: 多くのファイルは非常に小さいため（数KB以下）、ファイル全体がi-nodeと同じブロックに収まる可能性があります。その場合、ディスクアクセスは半分になります。ファイルがi-nodeと同じブロックに収まらない場合でも、最初のデータブロックはi-nodeと一緒に読み込めるため、ディスクアクセスを1回減らすことができます。これはシステムの性能を大幅に向上させる可能性があります。

31. 図4-27を考えます。特定のブロック番号に対して、両方のリストのカウンタが2の値を持つことは可能ですか。この問題はどのように修正すべきですか。

- **[解答訳]** それは起こるべきではありませんが、バグのせいで起こる可能性があります。これは、あるブロックが2つのファイルに存在し、かつフリーリストにも2回現れることを意味します。このエラーを修正する最初のステップは、フリーリストから両方のコピーを削除することです。次に、フリーブロックを取得し、問題のブロックの内容をそこにコピーします。最後に、そのブロックがファイルの一方で出現している箇所を、新しく取得したブロックのコピーを指すように変更します。これでシステムは再び一貫性を保ちます。

- **[より詳しい解説]** ファイルシステムチェッカー (fsck) が遭遇する可能性のある、極めて深刻な矛盾状態です [4.4.3]。

    - **カウンタが2になる意味**:
        - **使用中リストのカウンタが2**: あるブロックが2つの異なるファイルに属していることを意味します（図4-27(d)のケース）。これは非常に危険で、一方のファイルを変更すると他方のファイルも壊れる可能性があります。
        - **空きリストのカウンタが2**: あるブロックがフリーリストに2回登録されていることを意味します（図4-27(c)のケース）。これは、そのブロックが2回割り当てられる可能性があり、データ破壊につながります。
    - **両方が2になる**: これは、あるブロックが2つのファイルに属し、かつフリーリストにも2回登録されているという、矛盾の極みです。
    - **修正手順**:
        1. まず、そのブロックをフリーリストから完全に削除します。
        2. 新しい空きブロックを1つ割り当てます。
        3. 問題のブロックの内容を、新しく割り当てたブロックにコピーします。
        4. 2つのファイルのうち一方のファイルが、元のブロックではなく、この新しいコピーを指すようにi-nodeまたはFATを修正します。 これで、各ブロックは1つのファイルに属するか、フリーリストに1回だけ存在するかのどちらかになり、ファイルシステムの一貫性が回復します。

32. ファイルシステムの性能はキャッシュのヒット率に依存します。…ヒット率がhの場合にリクエストを満たすのに必要な平均時間の式を求めなさい。…

- **[解答訳]** 必要な時間は `h + 40 * (1 - h)` です。プロットはただの直線です。

- **[より詳しい解説]** これは、キャッシュの有効性を示すための**実効メモリアクセス時間 (EAT: Effective Memory Access Time)** の計算です。

    - **式**: `EAT = (ヒット率 * ヒット時のコスト) + (ミス率 * ミス時のコスト)`
    - **与えられた値**:
        - ヒット率 = `h`
        - ミス率 = `1 - h`
        - ヒット時のコスト（キャッシュから読み出し） = 1 ms
        - ミス時のコスト（ディスクから読み出し） = 40 ms
    - **計算**:
        - `EAT = (h * 1) + ((1 - h) * 40)`
        - `EAT = h + 40 - 40h`
        - `EAT = 40 - 39h` この式をプロットすると、h=0のときEAT=40ms、h=1のときEAT=1msとなる右下がりの

直線になります。これは、キャッシュヒット率がわずかに向上するだけで、平均アクセス時間が劇的に改善されることを示しています。

33. コンピュータに接続された外部USBハードドライブには、ライトスルーキャッシュとブロックキャッシュのどちらがより適していますか。

- **[解答訳]** この場合、ライトスルーキャッシュの方が適しています。なぜなら、更新されたファイルがユーザーが誤ってハードドライブを取り外す前にディスク上に確実に存在するようにするためです。
- **[より詳しい解説]** これは、データの永続性と性能のトレードオフに関する問題です。
    - **ライトスルーキャッシュ (Write-through cache)**: データを書き込む際、キャッシュとディスクの両方に**同時に**書き込みます。書き込みが完了するのは、ディスクへの書き込みが終わった後です。これにより、キャッシュとディスクの内容は常に一致します。
    - **ブロックキャッシュ (Write-back cache)**: データを書き込む際、まずキャッシュにだけ書き込みます。ディスクへの実際の書き込みは、後で（例えば、キャッシュブロックが追い出される時やsyncコマンド実行時など）行われます。
    - **USBドライブの場合**: ユーザーはいつでもUSBドライブを抜き取る可能性があります。もしブロックキャッシュ（ライトバック）を使っていると、キャッシュに書き込んだだけでまだディスクに書き込まれていないデータが、ドライブが抜かれた瞬間に失われてしまいます。ライトスルーキャッシュであれば、書き込み操作が完了した時点でデータはディスク上にあることが保証されるため、より安全です。

34. 学生の記録がファイルに保存されているアプリケーションを考えます。…「ブロック先行読み込み（block read-ahead）」技術はここで役立ちますか。

- **[解答訳]** ブロック先行読み込み技術は、ブロックをその使用に先駆けてシーケンシャルに読み込むことで性能を向上させます。このアプリケーションでは、レコードはおそらくシーケンシャルにはアクセスされないでしょう。なぜなら、ユーザーは任意の学生IDをいつでも入力できるからです。したがって、先行読み込み技術はあまり役立ちません。
- **[より詳しい解説]** 先行読み込み（Read-ahead）は、**シーケンシャルアクセス**を前提とした最適化手法です [4.4.4]。システムは、ファイルが順に読み取られていることを検知すると、次に要求されるであろうブロックを予測して事前にディスクから読み込み、キャッシュに入れておきます。 問題のアプリケーションは、学生IDに基づいてレコードを**ランダムに**アクセスします。ある学生の次に、全く関係のないIDの学生が検索される可能性が高いです。このようなランダムなアクセスパターンでは、次にどのブロックが必要になるかを予測することは不可能です。したがって、先行読み込みは効果がなく、むしろ無駄なディスクI/Oを発生させ、キャッシュ内の有用なブロックを追い出してしまう可能性があるため、性能を低下させることさえあります。

35. ブロック14から23までの10個のデータブロックを持つディスクを考えます。…f1とf2に割り当てられているデータブロックは何ですか。

- **[解答訳]** f1に割り当てられているブロックは、22, 19, 15, 17, 21です。f2に割り当てられているブロックは、16, 23, 14, 18, 20です。
- **[より詳しい解説]** これはFAT（File Allocation Table）のチェインをたどる問題です [4.3.2]。FATは、各エントリがファイルの次のブロックを指すリンクリストとして機能します。-1はファイルの終端を示します。
    - **ファイル f1**:
        1. ディレクトリによると、開始ブロックは **22** です。
        2. FATテーブルのインデックス22を見ると、値は19です。次のブロックは **19** です。
        3. インデックス19の値は15です。次のブロックは **15** です。
        4. インデックス15の値は17です。次のブロックは **17** です。
        5. インデックス17の値は21です。次のブロックは **21** です。
        6. インデックス21の値は-1なので、ここでファイルは終了です。

        - したがって、f1のブロック列は: 22，19，15，17，21
    - **ファイル f2**:
        1. ディレクトリによると、開始ブロックは **16** です。
        2. FATテーブルのインデックス16を見ると、値は23です。次のブロックは **23** です。
        3. インデックス23の値は14です。次のブロックは **14** です。
        4. インデックス14の値は18です。次のブロックは **18** です。
        5. インデックス18の値は20です。次のブロックは **20** です。
        6. インデックス20の値は-1なので、ここでファイルは終了です。
        - したがって、f2のブロック列は: 16，23，14，18，20

36. 図4-21の背後にある考えを考えますが、今度は平均シーク時間が6ミリ秒、回転数が15,000rpm、トラックあたり1,048,576バイトのディスクについてです。ブロックサイズがそれぞれ1KB、2KB、4KBの場合のデータレートは何ですか。

- **[解答訳]** 15,000rpmでは、ディスクは1周に4ミリ秒かかります。kバイトを読み込むための平均アクセス時間は、6 + 2 + (k/1,048,576) * 4 ミリ秒です。1KB, 2KB, 4KBのブロックに対するアクセス時間は、それぞれ約6.0039ms, 6.0078ms, 6.0156msです。これらのレートはそれぞれ約170.556 KB/sec, 340.890 KB/sec, 680.896 KB/sec となります。

- **[より詳しい解説]**

    1. **基本パラメータの計算**:
        - 平均シーク時間: 6 ms
        - 回転数: 15,000 rpm = 250 rps (revolutions per second)
        - 1回転あたりの時間: `1 / 250 = 0.004` 秒 = 4 ms
        - 平均回転遅延: 1回転の時間 / 2 = 2 ms
        - トラックあたりのバイト数: 1,048,576 バイト = 1 MB
        - 転送レート: `1 MB / 4 ms` = 250 MB/sec
    2. **ブロックごとの読み込み時間とデータレートの計算**:
        - **1KB (1024バイト) ブロック**:
            - 転送時間 = `1024 / (250 * 1024 * 1024) * 1000 ≈ 0.0039` ms
            - 合計時間 = 6(シーク) + 2(回転) + 0.0039(転送) ≈ 8.0039 ms
            - データレート = 1024バイト / 8.0039ms ≈ 127.9 KB/sec
        - **2KB (2048バイト) ブロック**:
            - 転送時間 ≈ 0.0078 ms
            - 合計時間 ≈ 8.0078 ms
            - データレート ≈ 255.7 KB/sec
        - **4KB (4096バイト) ブロック**:
            - 転送時間 ≈ 0.0156 ms
            - 合計時間 ≈ 8.0156 ms
            - データレート ≈ 511.0 KB/sec **注**: 解答集 は平均回転遅延を2msとして計算していますが、合計読み込み時間を `6 + 2 + ...` ではなく `8 + ...` としているため、計算結果が異なります。ここでは教科書の計算式 シーク + 回転遅延 + 転送時間 に基づいて再計算しました。いずれにせよ、ブロックサイズが大きくなるほど、シークと回転遅延のオーバーヘッドが相対的に小さくなり、実効データレートが向上することがわかります。

37. あるファイルシステムは4KBのディスクブロックを使用しています。ファイルサイズの中央値は1KBです。すべてのファイルが正確に1KBだった場合、ディスクスペースの何パーセントが無駄になりますか。…

- **[解答訳]** 全てのファイルが1KBの場合、各4KBブロックは1つのファイルと3KBの無駄なスペースを含みます。…これは75%の無駄なスペースにつながります。実際のファイルシステムでは、多くの大きなファイルと小さなファイルが存在します。大きなファイルはディスクをより効率的に利用します。例えば、32,769バイト

のファイルは9つのディスクブロックを使用し、スペース効率は約89%です。

- **[より詳しい解説]** これは**内部フラグメンテーション**の問題です [4.4.1]。
    - **無駄の計算**:
        - ブロックサイズ: 4KB
        - ファイルサイズ: 1KB
        - 1つのファイルを格納するために、1つの4KBブロックが割り当てられます。
        - ブロック内の使用領域: 1KB
        - ブロック内の未使用（無駄な）領域: `4KB - 1KB = 3KB`
        - 無駄な領域の割合 = 無駄な領域 / 割り当てられた総領域 = `3KB / 4KB = 0.75` = **75%**
    - **実際のファイルシステムでの考察**:
        - 実際のファイルシステムでは、図4-20 に示されるように、ファイルサイズは様々です。非常に大きなファイル（動画など）も多数存在します。
        - 大きなファイルの場合、内部フラグメンテーションの影響は相対的に小さくなります。例えば10MBのファイル（10240KB）を4KBブロックで格納すると、`10240 / 4 = 2560` ブロックをぴったり使い切るので無駄は0です。10MB+1KBのファイルなら、2561ブロックを使い、最後のブロックで3KBが無駄になりますが、ファイル全体に対する無駄の割合は `3KB / (10MB+1KB)` となり、非常に小さくなります。
        - したがって、様々なサイズのファイルが存在する実際のファイルシステムでは、全てのファイルが1KBであると仮定した場合よりも、**無駄になるスペースの割合は低くなる**と考えられます。

38. 4KBのディスクブロックサイズと4バイトのブロックポインタアドレス値が与えられた場合、10個のダイレクトアドレスと1個の間接ブロックを使用してアクセスできる最大のファイルサイズ（バイト単位）は何ですか。

- **[解答訳]** 間接ブロックは1024個のアドレスを保持できます。10個のダイレクトアドレスと合わせて、合計で1034個のアドレスがあります。各々が4KBのディスクブロックを指すので、最大のファイルは4,235,264バイトです。

- **[より詳しい解説]** i-nodeの構造から最大ファイルサイズを計算します [4.3.2]。
    1. **ダイレクトブロック**: i-node内に直接格納されている10個のアドレスポインタが指すデータ量。
        - `10アドレス * 4 KB/ブロック = 40 KB`
    2. **間接ブロック**: i-node内の1つのポインタが、**アドレスポインタのリストを格納したブロック**（間接ブロック）を指します。
        - 間接ブロックに格納できるアドレス数: `4 KB（ブロックサイズ）/ 4バイト（アドレスサイズ）= 1024 アドレス`
        - 間接ブロックが指すデータ量: `1024アドレス * 4 KB/ブロック = 4096 KB = 4 MB`
    3. **最大ファイルサイズ**:
        - `40 KB（ダイレクト）+ 4096 KB（間接）= 4136 KB`
        - バイト単位に変換: `4136 * 1024 = 4,235,264 バイト`

39. MS-DOSのファイルは、メモリ内のFAT-16テーブルのスペースを競合しなければなりません。…これは、すべてのファイルの合計長にどのような制約を課しますか。

- **[解答訳]** これは、全ファイルの合計長がディスクよりも大きくはなれないという制約を課します。これはあまり深刻な制約ではありません。もしファイルが集合的にディスクより大きかったら、それら全てをディスクに格納する場所はないでしょう。

- **[より詳しい解説]** この問題は、FAT-16の設計上の制約について考察させるものです [4.5.1]。
    - **FAT-16の仕組み**: FAT-16テーブルは、ディスク上の**全てのブロック**に対応するエントリを持っています。ファイルがk個のブロックを使用する場合、FATテーブル内のk個のエントリがそのファイルのために使用されます。

- **制約の本質**: FATテーブルのエントリ数はディスクの総ブロック数と同じです。したがって、「FATテーブルのスペースを競合する」というのは、実質的に「ディスク上のブロックを競合する」のと同じことです。ファイルがk個のブロックを占有すれば、FATテーブルのk個のエントリが使われますが、それはディスク上のk個のブロックが使われていることを意味します。
- **結論**: したがって、この制約は「**全ファイルの合計サイズがディスク容量を超えることはできない**」という自明な物理的制約以上のものを課してはいません。

40. あるUNIXファイルシステムは4KBのブロックと4バイトのディスクアドレスを持っています。iノードがそれぞれ10個のダイレクトエントリ、1個のシングル、ダブル、トリプル間接エントリを含む場合、最大のファイルサイズは何ですか。

- **[解答訳]** i-nodeは10個のポインタを保持します。単一間接ブロックは1024個のポインタを保持します。二重間接ブロックは1024^2個のポインタに相当します。三重間接ブロックは1024^3個のポインタに相当します。これらを合計すると、1,074,791,434ブロックとなり、これは約16.06GBです。 *(注：原文解答の16.06GBは計算が誤っています。1074Mブロック*×4KB/ブロック ≒ 4TBが正しいです）*

- **[より詳しい解説]** これはUNIX V7のi-node構造 を一般化した問題です。

  1. **1ブロックあたりのポインタ数**: `4KB / 4バイト = 1024` ポインタ
  2. **ダイレクトブロック**: `10ポインタ * 4KB/ブロック = 40 KB`
  3. **単一間接ブロック**: `1024ポインタ * 4KB/ブロック = 4 MB`
  4. **二重間接ブロック**: `1024 * 1024ポインタ * 4KB/ブロック = 4 GB`
  5. **三重間接ブロック**: `1024 * 1024 * 1024ポインタ * 4KB/ブロック = 4 TB`
  6. **最大ファイルサイズ**: これらを全て合計すると、ほぼ **4TB** となります（ダイレクト、単一、二重間接ブロックの寄与は4TBに比べて無視できるほど小さい）。

41. パス名 /usr/ast/courses/os/handout.t を持つファイルのiノードを取得するために、いくつのディスク操作が必要ですか。…

- **[解答訳]** 以下のディスク読み込みが必要です: /のディレクトリ、/usrのi-node、/usrのディレクトリ、/usr/astのi-node、/usr/astのディレクトリ、/usr/ast/coursesのi-node、/usr/ast/coursesのディレクトリ、/usr/ast/courses/osのi-node、/usr/ast/courses/osのディレクトリ、/usr/ast/courses/os/handout.tのi-node。 合計で10回のディスク読み込みが必要です。

- **[より詳しい解説]** これはパス名解決のプロセスをトレースする問題です [4.5.2, 1175-1177]。

  - **前提**:
    - ルート(/)のi-nodeはメモリにある。
    - 他のi-nodeやディレクトリはディスクにある。
    - 各ディレクトリは1ブロックに収まる。
  - **ステップ・バイ・ステップ**:
    1. /のi-nodeはメモリにある → ディスクアクセス 0回
    2. /のディレクトリブロックを読む（usrを探すため）→ 1回
    3. /usrのi-nodeを読む → 1回
    4. /usrのディレクトリブロックを読む（astを探すため）→ 1回
    5. /usr/astのi-nodeを読む → 1回
    6. /usr/astのディレクトリブロックを読む（coursesを探すため）→ 1回
    7. /usr/ast/coursesのi-nodeを読む → 1回
    8. /usr/ast/coursesのディレクトリブロックを読む（osを探すため）→ 1回
    9. /usr/ast/courses/osのi-nodeを読む → 1回
    10. /usr/ast/courses/osのディレクトリブロックを読む（handout.tを探すため）→ 1回
    11. /usr/ast/courses/os/handout.tのi-nodeを読む → 1回 合計で**10回**のディスク操作が必要です（パスの各要素（usr, ast, courses, os, handout.t）に対して、親ディレクトリの読み込みと自身のi-node

の読み込みが必要。ルートディレクトリはi-nodeがメモリにあるので1回少なくて済むが、ルートディレクトリ自体の読み込みは必要）。

42. 多くのUNIXシステムでは、iノードはディスクの先頭に保持されます。代替の設計は、ファイルが作成されるときにiノードを割り当て、iノードをファイルの最初のブロックの先頭に置くことです。この代替案の長所と短所を議論しなさい。

- **[解答訳]** 長所: 1) 未使用のi-nodeにディスクスペースが無駄にならない。2) i-nodeが尽きることがない。3) i-nodeと最初のデータを1回の操作で読み込めるため、ディスクの動きが少なくなる。 短所: 1) ディレクトリントリが16ビットのi-node番号の代わりに32ビットのディスクアドレスを必要とする。2) データを含まないファイル（空ファイル、デバイスファイル）でもブロック全体が使用される。3) ファイルシステムの整合性チェックが遅くなる。4) ファイルサイズがブロックサイズにぴったり合うように注意深く設計されたファイルが、i-nodeのせいでブロックサイズに収まらなくなり、性能が損なわれる。

- **[より詳しい解説]** これは、ファイルシステムのデータ構造の物理的配置に関する設計上のトレードオフを問う問題です [4.4.4, 4.5.2]。

    - **長所（Pros）**:
        1. **i-nodeの動的確保**: `mkfs`時に固定数のi-nodeを確保する必要がなく、ディスクスペースが残っている限りi-nodeが不足することはない。
        2. **性能向上（局所性）**: 小さいファイルを読み込む際、i-nodeとデータの先頭が同じブロックにあるため、ディスクシークが1回で済む。これは大きな性能向上につながる。
    - **短所（Cons）**:
        1. **ディレクトリントリの肥大化**: i-node番号（例：32ビット）の代わりに、より大きなディスクブロックアドレス（例：64ビット）を格納する必要がある。
        2. **空ファイル等でのスペース浪費**: 空ファイルやスペシャルファイルでも、i-nodeを格納するためだけに1ブロック全体を消費してしまう。
        3. **fsckの低速化**: i-nodeがディスク全体に分散するため、ファイルシステムチェッカーが全てのi-nodeをスキャンするのが非常に遅くなる。従来の方式ではi-nodeは一箇所にまとまっているため、高速にスキャンできる。
        4. **アライメント問題**: ブロックサイズぴったりに作られたファイル（例：データベースのページ）が、i-nodeの分だけはみ出してしまい、性能が低下する。

43. ファイルのバイトを逆順にするプログラムを書きなさい。最後のバイトが最初になり、最初のバイトが最後になるようにします。これは任意に長いファイルで動作しなければなりませんが、合理的に効率的にするように試みなさい。

- **[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]** 任意に長いファイルで動作させるためには、ファイル全体を一度にメモリに読み込む方法は使えません。効率的なアルゴリズムは以下のようになります。

    1. 2つのファイルポインタを用意します。一つはファイルの先頭 (`p_start`)、もう一つはファイルの末尾 (`p_end`)を指します。
    2. ループを開始します。`p_start` が `p_end` の前にいる間、ループを続けます。
    3. `p_start` の位置から1バイト読み込み（`byte_start`）、`p_end` の位置から1バイト読み込みます（`byte_end`）。
    4. `p_start` の位置に `byte_end` を書き込み、`p_end` の位置に `byte_start` を書き込みます。
    5. `p_start` を1バイト進め、`p_end` を1バイト戻します。
    6. ループを繰り返します。 この方法は、ファイル内での直接書き換え（in-place）を行うため、追加のディスク容量を必要としません。ただし、処理の途中でクラッシュするとファイルが破損するリスクがあります。より安全な方法は、別の出力ファイルに逆順で書き出すことです。その場合は、入力ファイルの末尾から1バイトずつ読み、出力ファイルの先頭から書き込んでいきます。

44. 与えられたディレクトリから始まり、その時点からファイルツリーを降りていき、見つけたすべてのファイルのサイズを記録するプログラムを書きなさい。すべてが終わったら、パラメータとして指定されたビン幅を使用してファイルサイズのヒストグラムを印刷すべきです。

- **[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]** このプログラムは、再帰的なディレクトリトラバーサルを用いて実装するのが一般的です。

    1. **再帰関数 traverse(path) を作成します。**
    2. traverse 関数内では、与えられた path を opendir で開きます。
    3. readdir を使ってディレクトリ内のエントリを一つずつ読み込みます。
    4. 各エントリについて：
        - stat システムコールを使って、ファイルの種類とサイズを取得します。
        - もしエントリがファイルであれば、そのサイズを取得し、ヒストグラムの対応するビン（bin_index = file_size / bin_width）のカウントを1増やします。
        - もしエントリがディレクトリ（かつ . や .. でない）であれば、新しいパスを構築し、traverse を再帰的に呼び出します。
    5. ディレクトリ内のすべてのエントリを処理したら closedir します。
    6. メインプログラムは、最初のディレクトリパスで traverse を呼び出し、処理が終わったらヒストグラムの結果を表示します。

45. UNIXファイルシステム内のすべてのディレクトリをスキャンし、ハードリンクカウントが2以上のすべてのiノードを見つけて特定するプログラムを書きなさい。そのようなファイルごとに、そのファイルを指すすべてのファイル名をまとめてリストアップしなさい。

- **[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]** この問題は、i-node番号とファイルパスのマッピングを管理する必要があり、やや複雑です。効率的な実装方法は以下の通りです。

    1. **i-nodeからパスリストへのマップを作成**: map<inode_number, list<path_string>> のようなハッシュマップ（または連想配列）を用意します。
    2. **ファイルシステムを再帰的に走査**: 問題44と同様に、ルートディレクトリからファイルシステム全体を再帰的に探索します。
    3. **マップの構築**: 各ファイル（またはディレクトリ）を見つけるたびに、そのi-node番号とフルパスを取得します。そして、ハッシュマップ map[inode_number].add(path_string) のように、i-node番号をキーとしてパスをリストに追加します。
    4. **結果のフィルタリングと表示**: 走査が完了した後、ハッシュマップを走査します。各エントリについて、パスのリストのサイズが2以上であれば、そのi-node番号と関連するすべてのパス名のリストを表示します。 この方法では、ファイルシステムの走査は1回で済みます。

46. UNIXのlsプログラムの新しいバージョンを書きなさい。このバージョンは、引数として1つ以上のディレクトリ名を取り、各ディレクトリについてそのディレクトリ内のすべてのファイルを1行に1ファイルずつリストアップします。各フィールドは、そのタイプに応じて合理的な方法でフォーマットされるべきです。ディスクアドレスがある場合は、最初のディスクアドレスのみをリストアップしなさい。

- **[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]** この演習は、ディレクトリの読み取りとファイル属性の取得・表示を組み合わせるものです。

    1. プログラムはコマンドライン引数（ディレクトリ名）をループで処理します。
    2. 各ディレクトリ名に対して opendir を呼び出します。
    3. readdir を使ってディレクトリ内のエントリを一つずつループで取得します。

4. 取得した各エントリ（ファイル名）に対して stat（または lstat でシンボリックリンク自体を扱う）を呼び出し、struct stat 構造体に属性情報を取得します。
5. struct stat 構造体の各フィールドを読み取り、整形して表示します。
    - st_mode: ファイルの種類（ディレクトリ、ファイルなど）とパーミッション（rwx）を表示します。
    - st_nlink: リンク数を表示します。
    - st_uid, st_gid: 所有者とグループIDを取得し、/etc/passwd や /etc/group を参照してユーザー名・グループ名に変換して表示します。
    - st_size: ファイルサイズを表示します。
    - st_mtime: 最終更新日時を人間が読める形式に変換して表示します。
    - ファイル名を表示します。
6. closedir でディレクトリを閉じます。 i-node構造体から最初のディスクアドレスを取得する部分は、実際のファイルシステムの実装に依存するため、標準のstat構造体からは直接取得できません。これは、より低レベルのファイルシステム操作が必要になることを示唆しています。

47. アプリケーションレベルのバッファサイズが読み取り時間に与える影響を測定するプログラムを実装しなさい…

- **[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]** この演習は、アプリケーションのバッファサイズとI/O性能の関係を実験的に検証するものです。

    1. **実装の方針**:
        - まず、数GBの巨大なテストファイルを作成します。
        - バッファサイズをパラメータとして（例：64バイトから4KBまで2のべき乗で変化させ）、ファイル全体を読み込むループを作成します。
        - 各バッファサイズについて、高精度タイマー（例：UNIXの gettimeofday）を使い、ファイルの読み込み開始から終了までの総時間を測定します。
        - 測定結果から、総書き込み時間と、writeシステムコール1回あたりの平均時間を計算し、バッファサイズとの関係をプロットします。
    2. **期待される結果と分析**:
        - **バッファサイズが小さい場合**: read システムコールの呼び出し回数が非常に多くなります。システムコールにはユーザーモードとカーネルモードの切り替えというオーバーヘッドがあるため、総読み込み時間は長くなります。
        - **バッファサイズが大きい場合**: システムコールの呼び出し回数が減り、オーバーヘッドが減少します。また、一度に多くのデータを転送することで、ディスクI/Oがより効率的になる可能性があります。
        - **結論**: 一般的に、バッファサイズがある程度大きくなるまでは性能が向上し、OSの内部バッファやディスクのブロックサイズに近い値で性能が飽和する傾向が見られるはずです。バッファサイズは、システムコールのオーバーヘッドとディスク転送効率のトレードオフに影響を与えるため、全体の書き込み時間に**大きな違いをもたらします**。

48. ディスクに保存された単一の通常ファイルに完全に含まれるシミュレートされたファイルシステムを実装しなさい…

- **[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]** この演習は、ファイルシステム全体の概念を理解するために、そのミニチュア版を実装させるものです。FUSE (Filesystem in Userspace) のようなライブラリを使うとより本格的になりますが、ここでは単一ファイル内でのシミュレーションを考えます。

    1. **データ構造の設計**:
        - ホストOS上の単一の大きなファイルを、仮想的なディスクとして扱います。
        - このファイル内に、ファイルシステムの構成要素をすべて配置します。例えば、

    - **スーパーブロック**: ファイルシステムの全体情報（ブロックサイズ、i-node数など）をファイルの先頭に置きます。
    - **i-nodeテーブル**: ファイルのメタデータを格納するi-nodeの配列を配置します。
    - **フリーブロック管理**: ビットマップまたはフリーリストを配置します。
    - **データブロック領域**: ファイルの実際のデータを格納する領域を確保します。
2. **アルゴリズムの選択**:
    - **ファイル割り当て**: i-node方式（インデックス割り当て）が柔軟性が高く、実装しやすいでしょう。i-nodeはダイレクトブロックポインタと間接ブロックポインタを持つことができます。
    - **空きブロック管理**: ビットマップが実装しやすいでしょう。
3. **コマンドの実装**:
    - `mkdir`: ディレクトリエントリ（ファイル名とi-node番号のペア）を持つ新しいファイル（i-nodeのモードをディレクトリに設定）を作成します。
    - `create/open`: 新しいi-nodeを割り当て、ディレクトリにエントリを追加します。オープン時には、ファイルディスクリプタに相当するものをプログラム内で管理します。
    - `read/write`: ファイルディスクリプタからi-nodeをたどり、データブロックにアクセスします。アクセスはホストOSの `seek` と `read/write` を使って、単一ファイル内の対応するオフセットに対して行います。
    - `ls`: ディレクトリファイルを読み込み、ファイル名とi-node番号のリストを表示します。 この演習を通して、ファイルシステムがどのように抽象化され、ディスク上のデータ構造を管理しているかを深く理解することができます。