第3章 メモリ管理 演習問題の解答と解説

1. IBM 360は、2KBのブロックをロックするために、各ブロックに4ビットのキーを割り当て、CPUがすべてのメモリアクセス時にそのキーをPSW内の4ビットキーと比較する方式を採用していました。本文で言及されていないこの方式の欠点を2つ挙げなさい。

- **[解答訳]** 第一に、比較を行うための特別なハードウェアが必要であり、そのハードウェアは高速でなければなりません。なぜなら、すべてのメモリアクセスで比較が行われるからです。第二に、4ビットのキーでは、OSを含めて16個のプログラムしかメモリに同時に存在できません。

- **[より詳しい解説]** 教科書3.1節で触れられているIBM 360のロックアンドキー方式には、本文で指摘された「静的再配置が必要」という欠点以外にも、以下のような問題点があります。

    1. **粒度（Granularity）の問題**: 保護の単位が2KBブロックであるため、それより細かい単位での保護設定ができません。例えば、あるプロセスが1KBしかメモリを必要としない場合でも2KBが割り当てられ、残りが無駄になる可能性があります。また、同じ2KBブロック内にあるデータの一部だけを読み取り専用にするといった柔軟な制御も不可能です。
    2. **同時実行可能なプロセス数の制限**: 4ビットのキーでは、0から15までの16通りの保護レベルしか設定できません。通常、OSがキーの一つを使用するため、ユーザープロセス用に残されるのは15個です。これは、同時にメモリ上で実行できる（それぞれが異なる保護を必要とする）プロセス数が最大15個に制限されることを意味し、現代の多重プログラミング環境では不十分です。

2. 図3-3では、ベースレジスタとリミットレジスタが同じ値16,384を含んでいます。これは単なる偶然ですか、それとも常に同じ値になるのでしょうか。もし偶然だとしたら、なぜこの例では同じ値になっているのですか。

- **[解答訳]** これは偶然です。ベースレジスタは16,384ですが、これはプログラムがアドレス16,384にロードされたからです。どこにでもロードされた可能性があります。リミットレジスタは16,384ですが、これはプログラムの長さが16,384バイトだからです。どのような長さでもあり得ました。ロードアドレスがプログラムの長さにちょうど一致したのは、全くの偶然です。

- **[より詳しい解説]** 教科書3.2.1節の図3-3で示されているように、ベースレジスタとリミットレジスタは異なる目的で使用されます。

    - **ベースレジスタ**: プロセスが物理メモリのどの番地からロードされているかを示す**開始アドレス**を保持します。この値は、OSがメモリの空き状況に応じて決定します。
    - **リミットレジスタ**: プロセスのアドレス空間の**長さ（サイズ）**を保持します。これは、プロセスが自身のメモリ領域を超えてアクセスするのを防ぐための保護に使われます。 この問題の例では、たまたま16,384バイトのプログラムが物理アドレス16,384番地にロードされたため、両方のレジスタの値が一致していますが、これは全くの偶然です。例えば、同じプログラムがアドレス0にロードされていれば、ベースレジスタは0、リミットレジスタは16,384となります。

3. スワッピングシステムは、コンパクションによってホール（空き領域）を解消します。多くのホールと多くのデータセグメントがランダムに分布しており、32ビットのメモリワードを読み書きするのに4ナノ秒かかると仮定した場合、4GB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]** ほぼメモリ全体をコピーする必要があり、各ワードは読み込まれてから別の場所に書き戻されます。4バイトを読むのに4ナノ秒かかるので、1バイトを読むのに1ナノ秒、書き込むのにさらに1ナノ秒、合計で1バイトあたり2ナノ秒のコンパクション時間が必要です。これは1秒あたり5億バイトの速度です。4GB（2^32バイト、約4.295×10^9バイト）をコピーするには、2^{32} / 500,000,000秒、つまり約8.59秒かかります。（訳注：原文解答では859ミリ秒となっていますが、計算上は約8.59秒が正しいと思われます）

- **[より詳しい解説]** メモリコンパクションは、教科書図3-4で示されているように、メモリ上の断片化したホールを一つにまとめる操作です。この問題では、その処理時間を計算します。

- **1バイトあたりのコピー時間**: 32ビット（4バイト）の読み書きに4ナノ秒かかります。これは、1バイトあたり1ナノ秒で読み取り、1ナノ秒で書き込むことを意味します。したがって、1バイトの移動には合計2ナノ秒かかります。
- **コピー速度**: 1秒間にコピーできるバイト数は 1秒 / (2ナノ秒/バイト) = 1 / (2 * 10^-9) = 5 * 10^8 バイト/秒、つまり5億バイト/秒です。
- **総時間**: 4GBは 4 * 2^30 = 4,294,967,296 バイトです。これをコピー速度で割ると、4,294,967,296 / 500,000,000 ≈ 8.59 秒となります。 解答集では859ミリ秒とされていますが、これは計算ミスと考えられます。教科書本体の例（16GBを16秒でコンパクション）とも比較すると、4GBに約8.6秒というのは妥当な計算結果です。いずれにせよ、コンパクションは非常に時間のかかる操作であることがわかります。

4. あるスワッピングシステムにおいて、メモリが次のサイズのホールで構成されているとします…。連続するセグメント要求…に対して、…ではどのホールが選択されますか。

- **[解答訳]** ファーストフィットは20MB、10MB、18MBのホールを取ります。ベストフィットは12MB、10MB、9MBのホールを取ります。ワーストフィットは20MB、18MB、15MBのホールを取ります。ネクストフィットは20MB、18MB、9MBのホールを取ります。

- **[より詳しい解説]** 教科書3.2.3節で説明されているメモリ割り当てアルゴリズムを適用します。

  - **ファーストフィット**: ホールリストを先頭から探し、最初に見つかった十分な大きさのホールを使用します。
    1. 12MB要求: 10MB(×) → 4MB(×) → **20MB(○)**。残りは8MBのホールになる。
    2. 10MB要求: **10MB(○)**。ホールはなくなる。
    3. 18MB要求: 4MB(×) → 8MB(×) → **18MB(○)**。ホールはなくなる。
  - **ベストフィット**: 全てのホールを調べ、要求サイズに最も近い（最小の）十分な大きさのホールを使用します。
    1. 12MB要求: 20, 18, 12, 15MBが候補。最小は**12MB(○)**。
    2. 10MB要求: **10MB(○)**。
    3. 9MB要求: **9MB(○)**。
  - **ワーストフィット**: 全てのホールを調べ、最大のホールを使用します。
    1. 12MB要求: **20MB(○)**。
    2. 18MB要求: **18MB(○)**。
    3. 15MB要求: **15MB(○)**。
  - **ネクストフィット**: ファーストフィットと同様ですが、前回見つけた場所から探索を開始します。
    1. 12MB要求: **20MB(○)**。ポインタは18MBのホールを指す。
    2. 10MB要求: 18MBから探索開始。**18MB(○)**。ポインタは7MBを指す。
    3. 9MB要求: 7MBから探索開始。7MB(×) → **9MB(○)**。

5. 物理アドレスと仮想アドレスの違いは何ですか。

- **[解答訳]** 物理メモリは物理アドレスを使用します。これらはメモリチップがバス上で応答する番号です。仮想アドレスは、プロセスのアドレス空間を指す論理アドレスです。したがって、32ビットワードのマシンは、4GBより多いメモリがあってもなくても、最大4GBまでの仮想アドレスを生成できます。

- **[より詳しい解説]** これはメモリ管理の基本的な概念です。

  - **物理アドレス (Physical Address)**: メモリハードウェア（RAMチップ）が直接認識するアドレスです。CPUがメモリバスにこのアドレスを乗せると、対応する物理的なメモリセルがアクセスされます。
  - **仮想アドレス (Virtual Address)**: プログラムが生成するアドレスです。これは、各プロセスに与えられた独立した論理的なアドレス空間内のアドレスを指します。OSとMMU（メモリ管理ユニット）が連携して、この仮想アドレスを対応する物理アドレスに変換します。 この分離により、各プロセスは他のプロセスとは独立した広大なアドレス空間を持つことができ、実際の物理メモリのサイズよりも大きなプログラムを実行すること（仮想メモリ）や、プロセスの保護・再配置が容易になります。

6. 4KBページと8KBページのそれぞれの場合について、次の10進数の仮想アドレスに対する仮想ページ番号とオフセットを計算しなさい：20000, 32768, 60000。

- **[解答訳]** 4KBページの場合の（ページ、オフセット）のペアは、(4, 3616), (8, 0), (14, 2656)です。8KBページの場合は、(2, 3616), (4, 0), (7, 2656)です。
- **[より詳しい解説]** 仮想ページ番号とオフセットは、以下の計算で求められます。
    - ページ番号 = `floor(仮想アドレス / ページサイズ)`
    - オフセット = 仮想アドレス % ページサイズ
    - **4KB (4096バイト) ページの場合**:
        - 20000: 20000 / 4096 = 4 余り 3616 → ページ4, オフセット3616
        - 32768: 32768 / 4096 = 8 余り 0 → ページ8, オフセット0
        - 60000: 60000 / 4096 = 14 余り 2656 → ページ14, オフセット2656
    - **8KB (8192バイト) ページの場合**:
        - 20000: 20000 / 8192 = 2 余り 3616 → ページ2, オフセット3616
        - 32768: 32768 / 8192 = 4 余り 0 → ページ4, オフセット0
        - 60000: 60000 / 8192 = 7 余り 2656 → ページ7, オフセット2656

7. 図3-9のページテーブルを使って、次の各仮想アドレスに対応する物理アドレスを答えなさい：(a) 20, (b) 4100, (c) 8300。

- **[解答訳]**
    a. 8212, (b) 4100, (c) 24684
- **[より詳しい解説]** 図3-9 は4KB（4096バイト）ページを使用しています。物理アドレスは (ページフレーム番号 * ページサイズ) + オフセット で計算します。
    - **(a) 仮想アドレス 20**:
        - ページ番号 = `floor(20 / 4096) = 0`
        - オフセット = `20 % 4096 = 20`
        - 図3-9から、仮想ページ0はページフレーム2にマップされています。
        - 物理アドレス = `2 * 4096 + 20 = 8192 + 20 = 8212`
    - **(b) 仮想アドレス 4100**:
        - ページ番号 = `floor(4100 / 4096) = 1`
        - オフセット = `4100 % 4096 = 4`
        - 図3-9から、仮想ページ1はページフレーム1にマップされています。
        - 物理アドレス = `1 * 4096 + 4 = 4096 + 4 = 4100` **(訂正：図3-9では仮想ページ1はページフレーム1ではなく、マップされていません。解答集の答えは図と矛盾しています。もし仮想ページ1がページフレーム1にマップされていると仮定すれば、この計算になります)**。
    - **(c) 仮想アドレス 8300**:
        - ページ番号 = `floor(8300 / 4096) = 2`
        - オフセット = `8300 % 4096 = 108`
        - 図3-9から、仮想ページ2はページフレーム6にマップされています。
        - 物理アドレス = `6 * 4096 + 108 = 24576 + 108 = 24684`

8. Intel 8086プロセッサはMMUを持っておらず、仮想メモリをサポートしていませんでした。それにもかかわらず、いくつかの企業は未変更の8086 CPUを搭載し、ページングを行うシステムを販売していました。彼らがそれをどのように実現したか、根拠のある推測をしなさい。

- **[解答訳]** 彼らはMMUを構築し、それを8086とバスの間に挿入しました。したがって、8086からのすべての物理アドレスはMMUに入力され、仮想アドレスとして扱われました。MMUはそれらを物理アドレスにマッピングし、バスに送りました。
- **[より詳しい解説]** この問題は、MMUの論理的な位置を考察させるものです。教科書図3-8 に示されているよう

に、MMUはCPUとメモリバスの間に位置し、アドレス変換を行います。Intel 8086自体にMMUが内蔵されていなくても、**外付けのMMUチップ**をマザーボード上に実装し、CPUが生成するアドレスをすべてこのMMUを経由させることで、ページングを実現できます。

1. 8086 CPUは、物理アドレスであると信じてアドレスをバスに出力します。
2. このアドレスは、メモリに直接行くのではなく、マザーボード上のカスタムMMUチップに捕捉されます。
3. MMUは、CPUから来たアドレスを「仮想アドレス」と解釈し、自身のページテーブル（これもマザーボード上の専用メモリに保持）を使って「物理アドレス」に変換します。
4. 変換された物理アドレスが、実際のメモリバスに送られ、RAMがアクセスされます。 ページフォールトが発生した場合、MMUはCPUに割り込みをかけ、OSがディスクからページをロードする処理を行います。 このようにして、CPUを変更することなく仮想メモリ機能を追加することが可能です。

9. ページング方式の仮想メモリが機能するために、どのようなハードウェアのサポートが必要ですか。

- **[解答訳]** 仮想ページを物理ページに再マップできるMMUが必要です。また、マップされていないページが参照された場合には、OSにトラップしてページをフェッチできるようにする必要があります。

- **[より詳しい解説]** ページングを実装するためには、以下の2つの主要なハードウェアサポートが不可欠です。

  1. **MMU (Memory Management Unit)**: CPU内に（または密接に連携して）配置され、プログラムが生成する仮想アドレスを物理アドレスに高速に変換する役割を担います。これには、ページテーブルを保持または参照する機能が含まれます。
  2. **ページフォールト機構**: プログラムがアクセスしようとした仮想ページが物理メモリ上に存在しない（ページテーブルエントリが無効になっている）場合、MMUはCPUに**トラップ（例外）**を発生させる必要があります。このトラップにより、OSのページフォールトハンドラが起動され、ディスクから必要なページをメモリにロードする処理が行われます。

10. コピーオンライトはサーバーシステムで使われる興味深いアイデアです。スマートフォンでも意味がありますか。

- **[解答訳]** iPhone、Android、Windows Phoneはすべて多重プログラミングをサポートしており、複数のプロセスがサポートされています。もしあるプロセスがフォークし、親と子の間でページが共有されるなら、コピーオンライトは間違いなく意味があります。スマートフォンはサーバーより小さいですが、論理的にはそれほど違いはありません。

- **[より詳しい解説]** コピーオンライト（Copy-on-Write, COW）は、forkシステムコールなどを高速化するための重要な最適化手法です。forkが呼ばれると、OSは親プロセスのメモリ空間を子プロセスに物理的にコピーするのではなく、ページテーブルだけをコピーし、両方のプロセスが同じ物理ページを共有するようにします。ただし、これらの共有ページは読み取り専用にマークされます。 親または子のどちらかが共有ページに書き込もうとすると、保護違反のトラップが発生します。OSはこのトラップを捕捉し、そのページだけを物理的にコピーして、書き込みを行おうとしたプロセスに専用のコピーを与え、ページを書き込み可能に設定します。 スマートフォンOS（Android, iOSなど）もLinuxやUNIXベースであり、プロセス生成にforkを利用します。アプリの起動時などにプロセスが生成されるため、コピーオンライトは**プロセスの生成を高速化し、メモリ消費を抑える**上で非常に有効です。したがって、スマートフォンでも大いに意味があります。

11. 次のCプログラムを考えます： … (a) 4KBページと64エントリTLBを持つマシンで、内部ループごとにTLBミスを起こすMとNの値は？ (b) ループが何度も繰り返されたら答えは変わるか？

- **[解答訳]**
  a. Mは少なくとも4096である必要があり、Xへのすべてのアクセスで TLBミスが保証されます。NはXがTLBをスラッシングするのに十分な大きさ、つまりXが256KBを超える必要があります。
  b. MはTLBミスを保証するために少なくとも4,096であるべきです。しかし、Nは64Kより大きい必要はありません。Nは64で十分です。
- **[より詳しい解説]** この問題は、TLBとページングの動作、特に「スラッシング」の理解を問うものです。

- **前提**:
  - ページサイズ: 4KB = 4096バイト
  - TLBエントリ数: 64
  - intのサイズは4バイトと仮定します。
- **(a) 1回のループで毎回TLBミスを起こす条件**:
  - **Mの値**: X[i]とX[i+step]が常に異なるページにあるようにする必要があります。stepはMなので、M * sizeof(int)がページサイズ以上であれば、アクセスごとに新しいページにジャンプします。つまりM * 4 >= 4096、したがって **M >= 1024** が必要です。（**注：解答集のM>=4096は誤り。stepはMであり、X[i]とX[i+M]へのアクセスなので、M*sizeof(int)が4096以上、つまりM>=1024が正しい**）
  - **Nの値**: ループが一巡する間に、TLBが保持できる64ページよりも多くのページにアクセスする必要があります。64ページは 64 * 4096 = 262144 バイト（256KB）に相当します。配列 X がこれより大きければ、ループの後半でアクセスするページのTLBエントリは、ループ前半で使われたエントリを追い出してしまいます。したがって、**N * sizeof(int) > 262144**、つまり **N > 65536** が必要です。
- **(b) ループが何度も繰り返される場合**:
  - M >= 1024 の条件は同じです。
  - ループが繰り返される場合、Nの値はTLBエントリ数より大きければ十分です。64ページ以上にアクセスすると、TLBは必ずスラッシングを起こします。したがって、N/M > 64 であれば、ループが繰り返されるうちにTLBの内容は完全に洗い替えられます。例えば **N = 65 * M** であれば十分です。（解答集のN=64は不正確で、N/M > 64がより正確な条件です）

12. ページストレージとして利用可能でなければならないディスク容量は、…に関連しています。最悪の場合のディスク容量要件の式を求めなさい。

- **[解答訳]** すべてのプロセスの仮想アドレス空間の合計はnvです。このうち、rバイトはRAMに保持できます。したがって、最悪の場合のディスク容量要件はnv - rです。この量は非現実的です。
- **[より詳しい解説]** この問題は、スワップ領域（バッキングストア）の最悪のシナリオを考えさせるものです。
  - n: プロセスの最大数
  - v: 1プロセスあたりの仮想アドレス空間のバイト数
  - r: RAMのバイト数 最悪のケースは、**すべてのプロセスが同時に、自身の仮想アドレス空間を最大限まで使用する**状況です。
  - 全プロセスが必要とする総仮想メモリ: n * v
  - そのうちRAMに保持できる量: r
  - したがって、残りをディスク（スワップ領域）に保持する必要があります。必要なディスク容量は **nv - r** となります。 この量は、現実にはほとんどありえません。なぜなら、ほとんどのプロセスは自身の仮想アドレス空間のごく一部しか実際には使用しないためです。このため、多くのOSは、実際に必要になったときにだけスワップ領域を割り当てる「オンデマンド」方式を採用しています。

13. 命令の実行に1ナノ秒かかり、ページフォールトが追加でnナノ秒かかるとします。k命令ごとにページフォールトが発生する場合の実効命令時間の式を求めなさい。

- **[解答訳]** k命令ごとにページフォールトが発生すると、平均して1命令あたり n/k ナノ秒のオーバーヘッドが追加されます。したがって、平均命令時間は 1 + n/k ナノ秒となります。
- **[より詳しい解説]** 実効命令時間（または実効メモリアクセス時間）は、各イベントの発生確率とそのコストの加重平均で計算されます。
  - 通常の命令実行時間: 1 nsec
  - ページフォールトの追加コスト: n nsec
  - ページフォールトの発生確率: 1/k
  - ページフォールトが発生しない確率: (k-1)/k 実効命令時間 = (フォールトなしの時間 × 発生しない確率) + (フォールトありの時間 × 発生する確率) = (1) * ((k-1)/k) + (1 + n) * (1/k) = (k-1)/k + (1+n)/k =

(k-1+1+n)/k = (k+n)/k = 1 + n/k したがって、実効命令時間は **1 + n/k** ナノ秒となります。

14. あるマシンは32ビットのアドレス空間と8KBのページを持っています。…CPU時間の何パーセントがページテーブルのロードに費やされますか。

- **[解答訳]** ページテーブルには 2^32 / 2^13 = 524,288個のエントリがあります。ページテーブルをロードするには52ミリ秒かかります。プロセスが100ミリ秒実行される場合、このうち52ミリ秒がページテーブルのロードに、48ミリ秒が実行に使われます。したがって、時間の52%がページテーブルのロードに費やされます。

- **[より詳しい解説]** この問題は、コンテキストスイッチ時のページテーブルロードのオーバーヘッドを計算するものです。

    1. **ページテーブルのエントリ数**:
        - アドレス空間: 32ビット = 2^32 バイト
        - ページサイズ: 8KB = 2^13 バイト
        - エントリ数 = アドレス空間サイズ / ページサイズ = 2^32 / 2^13 = 2^19 = 524,288
    2. **ページテーブルのロード時間**:
        - 1ワードのロード時間: 100 nsec = 0.1 µsec
        - 総ロード時間 = エントリ数 * 1ワードあたりの時間 = 524,288 * 100 nsec
        - = 52,428,800 nsec = 52.4288 msec （約52ミリ秒）
    3. **CPU時間の割合**:
        - 1プロセスの総時間: 100 msec
        - ページテーブルロードに費やす時間: 52 msec
        - 割合 = (ロード時間 / 総時間) * 100 = (52 / 100) * 100 = 52% したがって、CPU時間の**52%**がページテーブルのロードというオーバーヘッドに使われます。これは、ページテーブル全体をハードウェアにロードする方式がいかに非効率であるかを示しています。

15. あるマシンが48ビットの仮想アドレスと32ビットの物理アドレスを持っているとします。(a) 4KBページの場合、単一レベルページテーブルにはいくつのエントリが必要か。(b) 32エントリのTLBがある場合、…TLBはどれくらい効果的か。

- **[解答訳]**
    a. ページごとに1つのエントリが必要なので、2^36 = 16 x 1024 x 1024エントリが必要です。ページ番号フィールドには 36 = 48 - 12 ビットあるからです。
    b. 命令アドレスは100% TLBにヒットします。データページは、プログラムが次のデータページに進むまで100%ヒットします。4KBページには1,024個のlong integerが収まるので、1,024データ参照ごとに1回のTLBミスと1回の余分なメモリアクセスが発生します。
- **[より詳しい解説]**
    a. **ページテーブルのエントリ数**:
    - 仮想アドレス空間: 48ビット
    - ページサイズ: 4KB = 2^12 バイト
    - オフセット部のビット数: 12ビット
    - 仮想ページ番号のビット数: 48 - 12 = 36 ビット
    - したがって、必要なエントリ数は 2^36 個です。これは約687億エントリに相当し、単一レベルページテーブルが非現実的であることがわかります。
    b. **TLBの効果**:
    - **命令ページ**: プログラムの命令は1ページに収まるので、一度そのページがTLBにロードされれば、以降の命令フェッチはすべてTLBヒットになります。
    - **データページ**: long integerを4バイトと仮定すると、1ページ（4096バイト）には 4096 / 4 = 1024 個の要素が収まります。プログラムは配列を順次読み取るため、1024回データにアクセスする間は同じデータページを参照し、TLBヒットが続きます。しかし、1025回目のアクセスでは新しいページに移るため、ここで**TLBミス**が発生します。

- **結論**: 1024回のデータアクセスのうち1回だけがTLBミスとなるため、TLBは非常に効果的です。ヒット率は 1023/1024 ≈ 99.9% となります。これは、プログラムの**参照の局所性**がTLBの性能に大きく寄与する良い例です。

16. ある仮想メモリシステムについて…実効アドレス変換時間はどれくらいですか。

- **[解答訳]** ヒットの確率はTLBで0.99、ページテーブルで0.0099、ページフォールトで0.0001です。実効アドレス変換時間（ナノ秒）は、0.99 x 1 + 0.0099 x 100 + 0.0001 x 6 x 10^6 となり、約602クロックサイクルです。ページフォールトが10,000回に1回しか発生しなくても、ページ置換時間が支配的であるため、実効アドレス変換時間は非常に高くなることに注意してください。

- **[より詳しい解説]** 実効アクセス時間（EAT）は、各イベントのコストにその発生確率を掛けて合計することで求められます。

  - TLBヒット:
    - 確率: 0.99
    - コスト: 1 nsec
  - TLBミス、ページテーブルヒット（ページはメモリに存在）:
    - 確率: 0.9999 (ページがメモリにある確率) - 0.99 (TLBヒットの確率) = 0.0099
    - コスト: 100 nsec (ページテーブル検索) + 1 nsec (TLBアクセス失敗分) ≈ 100 nsec
  - TLBミス、ページフォールト:
    - 確率: 0.0001
    - コスト: 6 ms = 6,000,000 nsec EAT = (0.99 * 1) + (0.0099 * 100) + (0.0001 * 6,000,000) EAT = 0.99 + 0.99 + 600 = 601.98 nsec したがって、実効アドレス変換時間は約**602ナノ秒**（または602クロックサイクル）となります。解答が指摘するように、ごくわずかな確率で発生するページフォールトが、その莫大なコスト（ディスクアクセス）ゆえに全体の性能を大きく左右することがわかります。

17. あるマシンが38ビットの仮想アドレスと32ビットの物理アドレスを持っているとします。(b) 2階層ページテーブル、16KBページ、4バイトエントリの場合、…ビット割り当てはどうすべきか。

- **[解答訳]**
  a. 多階層ページテーブルは、単一レベルページテーブルに比べて、必要な実際のページテーブルページの数を削減します。これは階層構造によるものです。実際、命令とデータの局所性が高いプログラムでは、トップレベルのページテーブル（1ページ）、命令ページ（1ページ）、データページ（1ページ）だけが必要になります。
  b. 各フィールドに12ビットずつ割り当てるべきです。オフセットフィールドは16KBをアドレッシングするために14ビット必要です。これにより、ページフィールドには24ビットが残ります。各エントリは4バイトなので、1ページには2^12個のページテーブルエントリが収まり、1ページをインデックスするには12ビットが必要です。したがって、各ページフィールドに12ビットを割り当てると、38ビットアドレス空間全体をアドレッシングできます。
- **[より詳しい解説]**
  b. **ビット割り当ての計算**:
  1. **オフセット**: ページサイズが16KB = 2^14 バイトなので、オフセットには **14ビット** が必要です。
  2. **ページ番号部**: 仮想アドレス全体は38ビットなので、ページ番号部には 38 - 14 = 24 ビットが残ります。これを2階層ページテーブル（PT1とPT2）に分割します。
  3. **1ページあたりのエントリ数**: 1ページは16KB = 2^14 バイト。1エントリは4バイト = 2^2 バイト。したがって、1ページには 2^14 / 2^2 = 2^12 個のエントリが収まります。
  4. **PT1とPT2のビット数**: 1つの（下位レベルの）ページテーブルを指し示すためには、2^12 個のエントリの中から1つを選ぶので、**12ビット** のインデックスが必要です。
  5. **結論**: ページ番号部の24ビットを、トップレベル（PT1）とセカンドレベル（PT2）にそれぞれ **12ビット** ずつ割り当てるのが適切です。

    - 仮想アドレス = PT1（12ビット）+ PT2（12ビット）+ オフセット（14ビット）
    - 合計ビット数: 12 + 12 + 14 = 38 ビット。これで仮想アドレス空間全体をカバーできます。

18. 3.3.4節では、Pentium Proが…依然として4GBのメモリしかアドレッシングできなかったと述べています。ページテーブルエントリが64ビットであるのに、どのようにしてこの記述が真実となりうるのか説明しなさい。

- **[解答訳]** 仮想アドレスは（PT1、PT2、PT3、オフセット）から（PT1、PT2、PT3、PT4、オフセット）へと変更されました。しかし、仮想アドレスは依然として32ビットしか使用していませんでした。仮想アドレスのビット構成が（10, 10, 12）から（2, 9, 9, 12）へと変更されました。

- **[より詳しい解説]** 教科書3.3.4節で言及されているPentium ProのPAE (Physical Address Extension) モードに関する問題です。このモードでは、ページテーブルエントリ（PTE）が64ビットに拡張されましたが、CPUが生成する**仮想アドレスは32ビットのまま**でした。

    - **物理アドレスの拡張**: 64ビットPTEの主な目的は、ページフレーム番号フィールドを拡張し、4GB（2^32バイト）を超える物理メモリをアドレッシングすることでした。
    - **仮想アドレスの制限**: プロセスが生成する仮想アドレスは32ビットのままであったため、1つのプロセスが見ることができるアドレス空間は依然として4GBに制限されていました。
    - **ページテーブル構造の変更**: 32ビット仮想アドレスを（2, 9, 9, 12）のフィールドに分割する新しい3階層ページテーブルが導入されました。この構造でも、ページングされるアドレス空間の合計は 2 + 9 + 9 + 12 = 32 ビット分です。 つまり、**システム全体としては4GB以上の物理RAMを搭載・利用できましたが、個々のプロセスのアドレス空間は4GBのままであった**ため、この記述は真実となります。

19. 32ビットアドレスを持つコンピュータが2階層のページテーブルを使用しています。…ページの大きさはどれくらいで、アドレス空間にはいくつのページがありますか。

- **[解答訳]** 仮想ページ番号には20ビットが使われており、残りの12ビットがオフセットに使われます。これにより、4KBのページが得られます。仮想ページに20ビットあるため、2^20個のページが存在します。

- **[より詳しい解説]** 2階層ページテーブルのフィールド分割からページサイズとページ数を求めます。

    - トップレベルページテーブルフィールド: 9ビット
    - セカンドレベルページテーブルフィールド: 11ビット
    - **ページ番号部の合計ビット数**: 9 + 11 = 20 ビット
    - **オフセット部のビット数**: 32ビット（総アドレス長）- 20ビット（ページ番号部）= 12 ビット
    - **ページサイズ**: オフセットが12ビットなので、ページサイズは 2^12 = 4096 バイト、つまり**4KB**です。
    - **アドレス空間のページ数**: ページ番号部が20ビットなので、総ページ数は 2^20 = 1,048,576 ページです。

20. 32ビットの仮想アドレスと4KBのページを持つコンピュータがあります。…ページテーブルエントリはいくつ必要ですか。

- **[解答訳]** 単一レベルのページテーブルの場合、2^32 / 2^12 = 1M個のページが必要です。したがって、ページテーブルには1M個のエントリが必要です。2レベルページングの場合、メインのページテーブルには1K個のエントリがあり、それぞれが2番目のページテーブルを指します。これらのうち2つだけが使用されます。したがって、合計で3つのページテーブルエントリが必要です：トップレベルテーブルに1つ、2つの下位レベルテーブルにそれぞれ1つずつです。

- **[より詳しい解説]**

    - **単一レベルページング**:
        - アドレス空間: 32ビット
        - ページサイズ: 4KB = 2^12 バイト
        - 必要なページ数 = 2^32 / 2^12 = 2^20 = 1,048,576 (1M) ページ
        - したがって、ページテーブルには**1M個のエントリ**が必要です。

- **2レベルページング (各10ビット)**:
  - 仮想アドレスは (PT1: 10ビット, PT2: 10ビット, オフセット: 12ビット) に分割されます。
  - プログラムとデータは最下位ページ（ページ0）に収まります。これはPT1=0, PT2=0のページです。
  - スタックは最上位ページに収まります。32ビットアドレス空間の最上位ページは、ページ番号が 2^20 - 1 です。これは16進数で FFFFF であり、PT1=3FF(16進)=1023, PT2=3FF(16進)=1023に対応します。
  - したがって、トップレベルページテーブル（1024エントリ）では、インデックス0とインデックス1023の**2つのエントリ**だけが有効な下位ページテーブルを指します。他のエントリは未使用です。
  - PT1=0が指す下位ページテーブルでは、PT2=0に対応するエントリが1つだけ使用されます。
  - PT1=1023が指す下位ページテーブルでは、PT2=1023に対応するエントリが1つだけ使用されます。
  - 合計で必要なページテーブルページ（エントリではない）は、トップレベル1つ、下位レベル2つの**3ページ**です。

21. 512バイトのページを持つコンピュータでのプログラム断片の実行トレースを以下に示します。…このプログラムによって生成されるページ参照列を答えなさい。

- **[解答訳]** コードと参照列は以下の通りです。 LOAD 6144,R0 1(I), 12(D) PUSH R0 2(I), 15(D) CALL 5120 2(I), 15(D) JEQ 5152 10(I) (I)は命令参照、(D)はデータ参照を示します。

- **[より詳しい解説]** ページサイズは512バイトです。各アドレスがどのページに属するかを計算します。 ページ番号 = floor(アドレス / 512)

  1. Load word 6144 into register 0:
     - 命令アドレス: 1020 → ページ floor(1020/512) = 1。**参照: ページ1 (命令)**
     - データアドレス: 6144 → ページ floor(6144/512) = 12。**参照: ページ12 (データ)**
  2. Push register 0 onto the stack:
     - 命令アドレス: 1024 (1020+4) → ページ floor(1024/512) = 2。**参照: ページ2 (命令)**
     - スタックアドレス: 8188 (8192-4) → ページ floor(8188/512) = 15。**参照: ページ15 (データ)**
  3. Call a procedure at 5120, stacking the return address:
     - 命令アドレス: 1028 (1024+4) → ページ floor(1028/512) = 2。**参照: ページ2 (命令)**
     - スタックアドレス (リターンアドレスのpush): 8184 (8188-4) → ページ floor(8184/512) = 15。**参照: ページ15 (データ)**
     - ジャンプ先アドレス: 5120 → ページ floor(5120/512) = 10。これは次の命令フェッチなので、ここで参照列に含めます。
  4. Subtract the immediate constant 16 from the stack pointer (プロシージャ内の最初の命令):
     - 命令アドレス: 5120 → ページ 10。**参照: ページ10 (命令)**
  5. Compare the actual parameter to the immediate constant 4:
     - 命令アドレス: 5124 (5120+4) → ページ 10。**参照: ページ10 (命令)**
  6. Jump if equal to 5152:
     - 命令アドレス: 5128 (5124+4) → ページ 10。**参照: ページ10 (命令)** 重複を除いたページ参照列は **1, 12, 2, 15, 10** となります。解答集の参照列は命令ごとのページ番号を示しているようです。

22. アドレス空間に1024ページを持つプロセスがあるコンピュータは…平均オーバーヘッドを2ナノ秒に削減するには、どれくらいのヒット率が必要ですか。

- **[解答訳]** 実効命令時間は 1h + 5(1 - h) です。ここで h はヒット率です。この式を2に等しいとおいて h について解くと、h は少なくとも0.75でなければならないことがわかります。

- **[より詳しい解説]** 実効オーバーヘッドを計算する式を立てます。

  - TLBヒット時のオーバーヘッド: 1 nsec
  - TLBミス時のオーバーヘッド: 1 nsec (TLB検索失敗) + 5 nsec (ページテーブル参照) = 6 nsec
  - ヒット率を h とします。ミス率は 1-h です。 平均オーバーヘッド = (ヒット時のコスト * ヒット率) +

(ミス時のコスト * ミス率) 2 = (1 * h) + (6 * (1-h)) 2 = h + 6 - 6h 5h = 4 h = 0.8 したがって、ヒット率は**80%**必要です。 **注：解答集の計算 1h + 5(1 - h) は、ミス時のコストを5nsec（ページテーブル参照のみ）としています。この場合、 2 = h + 5 - 5h となり 4h=3 で h=0.75 となります。どちらの解釈が正しいかは問題文からは断定しにくいですが、通常TLBミス時には両方のコストがかかると考えられます。**

23. TLBに必要な連想メモリデバイスは、ハードウェアでどのように実装できますか。また、そのような設計が拡張性にどのような影響を与えますか。

- **[解答訳]** 連想メモリは、基本的に複数のレジスタの内容をキーと同時に比較します。各レジスタには、レジスタの内容と検索キーの各ビットを比較するコンパレータのセットが必要です。このようなデバイスを実装するために必要なゲート（またはトランジスタ）の数はレジスタの数に線形に比例するため、設計を拡張するのは線形に高価になります。

- **[より詳しい解説]** 連想メモリ（Content-Addressable Memory, CAM）は、通常のRAMのようにアドレスを指定してデータを読み出すのではなく、データを指定してそれが格納されている場所（アドレス）を見つけ出します。

    - **実装**: TLBでは、仮想ページ番号が検索キーになります。TLBの各エントリは、仮想ページ番号を保持するフィールドと、それに対応する物理ページフレーム番号などを保持するフィールドを持っています。ハードウェアは、与えられた仮想ページ番号を**すべてのエントリの仮想ページ番号フィールドと同時に（並列に）比較**します。一致が見つかれば、対応する物理ページフレーム番号が即座に出力されます。この並列比較を実現するために、各エントリに専用の比較回路が必要です。
    - **拡張性**: この並列比較回路のため、エントリ数を増やすと、回路規模（トランジスタ数）と消費電力が線形に増加します。これが、TLBのエントリ数が通常、64や128といった比較的小さな数に限定される理由です。大規模な連想メモリはコストが高く、実装が困難になります。

24. 48ビットの仮想アドレスと32ビットの物理アドレスを持つマシンがあります。ページは8KBです。単一レベルの線形ページテーブルには、いくつのエントリが必要ですか。

- **[解答訳]** 8KBのページと48ビットの仮想アドレス空間では、仮想ページ数は 2^48 / 2^13 であり、2^35（約340億）です。

- **[より詳しい解説]**

    - 仮想アドレス空間: 48ビット
    - ページサイズ: 8KB = 2^13 バイト
    - オフセット部のビット数: 13ビット
    - 仮想ページ番号のビット数: 48 - 13 = 35 ビット
    - したがって、単一レベルページテーブルには **2^35個のエントリ**が必要です。これは、34,359,738,368個という天文学的な数であり、単一レベルページテーブルが現実的でないことを示しています。

25. 8KBのページ、256KBの主記憶、64GBの仮想アドレス空間を持つコンピュータが、…逆引きページテーブルを使用しています。…ハッシュテーブルはどれくらいの大きさであるべきですか。

- **[解答訳]** 主記憶には 2^18 / 2^13 = 32,768ページあります。32Kエントリのハッシュテーブルは、平均チェーン長が1になります。1未満にするには、次のサイズ、65,536エントリに上げる必要があります。32,768エントリを65,536テーブルスロットに広げると、平均チェーン長は0.5になり、高速な検索が保証されます。

- **[より詳しい解説]** 逆引きページテーブルは、物理ページフレームごとエントリを持ちます。ハッシュテーブルは、この逆引きページテーブルを高速に検索するために使用されます。

    1. **物理ページフレーム数**:
        - 主記憶サイズ: 256KB = 2^18 バイト

    - ページサイズ: 8KB = 2^13 バイト
    - ページフレーム数 = 2^18 / 2^13 = 2^5 = 32 フレーム **注：解答集の計算 2^18 / 2^13 = 32,768 は誤りです。おそらく主記憶は256MB=2^28バイトの誤記でしょう。以下、256MBとして計算します。**
    - 256MB = 2^28 バイト
    - ページフレーム数 = 2^28 / 2^13 = 2^15 = 32,768 フレーム
2. **ハッシュテーブルのサイズ**:
    - ハッシュテーブルの負荷率（エントリ数/スロット数）が、平均チェーン長になります。
    - 平均チェーン長を1未満にするには、スロット数がエントリ数（=ページフレーム数）より大きい必要があります。
    - ページフレーム数は32,768です。
    - ハッシュテーブルのサイズは2のべき乗であるという仮定なので、32,768より大きい最小の2のべき乗は **65,536** です。
    - この場合、負荷率は 32768 / 65536 = 0.5 となり、平均チェーン長が1未満という条件を満たします。

26. あるコンパイラ設計コースの学生が、最適ページ置換アルゴリズムを実装するために使用できるページ参照のリストを生成するコンパイラを作成するプロジェクトを教授に提案しました。これは可能ですか。なぜ可能、あるいは不可能ですか。実行時のページング効率を改善するためにできることはありますか。

- **[解答訳]** これは、プログラムの実行過程がコンパイル時に完全に予測可能という、一般的でない、あまり有用でないケースを除いては、おそらく不可能です。もしコンパイラが、プロシージャの呼び出し箇所の情報をコード内に集めることができれば、この情報はリンク時にプロシージャを呼び出しコードに近い場所に再配置するために利用できるかもしれません。これにより、プロシージャが呼び出し元のコードと同じページにある可能性が高まります。もちろん、これはプログラム内の多くの場所から呼ばれるプロシージャにはあまり役立ちません。

- **[より詳しい解説]** この問題は、最適ページ置換アルゴリズムの実現可能性とその代替案について考察させるものです。

    - **最適アルゴリズムの実現不可能性**: 最適ページ置換アルゴリズムは、「将来最も長く参照されないページ」を追い出します。これを実装するには、プログラムの**将来のメモリアクセスパターン全体**を正確に予測する必要があります。しかし、コンパイル時には、入力データ、ユーザーの操作、条件分岐の結果などが不明なため、実行時の正確な参照順序を予測することは不可能です。したがって、この学生の提案は基本的には実現不可能です。
    - **コンパイラによるページング効率の改善**: コンパイラが将来を予測することはできませんが、**参照の局所性**を高めるための最適化を行うことは可能です。
        - **コードの再配置**: よく一緒に呼び出される関数や、ループ内でアクセスされるデータブロックなどを、物理的に近いメモリアドレスに配置するようコンパイラやリンカが調整することで、それらが同じページに収まる確率が高まります。これにより、ページフォールトの発生率を下げることができます。
        - **プリフェッチ命令**: 一部のアーキテクチャでは、コンパイラが「このデータはもうすぐ必要になる」というヒントをCPUに与えるプリフェッチ命令をコードに挿入できます。これにより、データが必要になる前にキャッシュやメモリにロードされ、メモリアクセスの遅延を隠蔽できます。

27. ある仮想ページ参照ストリームが、長いページ参照シーケンスの繰り返しと、それに続く時折のランダムなページ参照を含んでいるとします。… (a) なぜ標準的な置換アルゴリズム（LRU, FIFO, clock）はこのワークロードを効果的に扱えないのですか。(b) このプログラムに500ページフレームが割り当てられた場合、…はるかに優れた性能を発揮するアプローチを説明しなさい。

- **[解答訳]**
    a. シーケンス長よりも小さいページ割り当てに対して、これらのアルゴリズムはすべてページフォールトを起こします。

    b. 500フレームが割り当てられた場合、ページ0から498を固定フレームにマップし、残りの1フレームを他のページに使用します。
- **[より詳しい解説]** この問題は、特定の参照パターンに対する既存アルゴリズムの弱点と、それを克服するための特化した戦略を考えさせるものです。
    - **(a) 標準アルゴリズムの弱点**: 参照シーケンス 0，1，...，511 は512ページから成ります。もし利用可能なページフレーム数が512未満（例えば511）の場合、LRUやFIFOのようなアルゴリズムは、シーケンスの最後の方のページ（例：ページ511）をロードするために、シーケンスの先頭のページ（例：ページ0）を追い出さざるを得ません。そして、次の繰り返しでページ0が参照されると、再びページフォールトが発生します。このように、シーケンスを一周するたびにすべてのページが入れ替わり、参照ごとにページフォールトが発生する**スラッシング**状態に陥ります。
    - **(b) より良いアプローチ**: この参照パターンの特徴は、**大部分のページが規則的に、少数のページが不規則に**アクセスされる点です。この知識を活用します。
        - プログラムに500フレームが割り当てられているので、繰り返しアクセスされるシーケンスのうち、最初の499ページ（ページ0から498）を**メモリにロック（ピン留め）**します。これにより、これらのページは置換対象から外れ、常にメモリに存在することになります。
        - 残りの1フレームを、シーケンスの残り（ページ499〜511）とランダムな参照（ページ431, 332など）のための「犠牲」フレームとして使用します。
        - この戦略により、規則的なシーケンス部分でのページフォールトはほぼなくなり、フォールトは不規則な参照部分に限定されます。これにより、ページフォールト率を劇的に下げることができます。これは、アプリケーションの動作特性をOSが知っている（または推測できる）場合に、いかに性能を改善できるかを示す好例です。

28. 4つのページフレームと8つのページでFIFOページ置換が使用される場合、参照列 0172327103 で、…何回のページフォールトが発生しますか。次に、この問題をLRUについて繰り返しなさい。

- **[解答訳]** FIFOの場合のページフレームは以下のようになります： x0172333300 xx017222233 xxx01777722 xxxx0111177 FIFOでは6回のページフォールトが発生します。LRUの場合のページフレームは以下のようになります： x0172327103 xx017232710 xxx01773271 xxxx0111327 LRUでは7回のページフォールトが発生します。

- **[より詳しい解説]**

    - **FIFO (First-In, First-Out)**: 最も古くメモリに入ったページを追い出します。 | 参照 | フレーム0 | フレーム1 | フレーム2 | フレーム3 | フォールト | |—|—|—|—|—|—|—| | **0** | 0 | | | | * | | **1** | 0 | 1 | | | * | | **7** | 0 | 1 | 7 | | * | | **2** | 0 | 1 | 7 | 2 | * | | **3** | **3** | 1 | 7 | 2 | * (0を置換) | | **2** | 3 | 1 | 7 | 2 | | | **7** | 3 | 1 | 7 | 2 | | | **1** | 3 | 1 | 7 | 2 | | | **0** | 3 | **0** | 7 | 2 | * (1を置換) | | **3** | 3 | 0 | 7 | 2 | | **合計6ページフォールト** (解答集のトレースとは若干異なりますが、結果は一致します)

    - **LRU (Least Recently Used)**: 最も長い間参照されていないページを追い出します。 | 参照 | フレーム0 | フレーム1 | フレーム2 | フレーム3 | フォールト | |—|—|—|—|—|—|—| | **0** | 0 | | | | * | | **1** | 0 | 1 | | | * | | **7** | 0 | 1 | 7 | | * | | **2** | 0 | 1 | 7 | 2 | * | | **3** | **3** | 1 | 7 | 2 | * (0を置換) | | **2** | 3 | 1 | 7 | 2 | | | **7** | 3 | 1 | 7 | 2 | | | **1** | 3 | 1 | 7 | 2 | | | **0** | **0** | 1 | 7 | 2 | * (3を置換) | | **3** | 0 | 1 | **3** | 2 | * (7を置換) | **合計7ページフォールト**

    この例は、直感に反してFIFOがLRUより性能が良い場合があることを示す有名な例（ベラディの異常）です。

29. 図3-15(b)のページシーケンスを考えます。ページBからAまでのRビットがそれぞれ 11011011 であるとします。セカンドチャンスはどのページを削除しますか。

- **[解答訳]** 0ビットを持つ最初のページが選択されます。この場合、Dです。

- **[より詳しい解説]** セカンドチャンスアルゴリズムは、FIFOを改良したものです。

    1. FIFOキューの先頭（最も古いページ）を調べます。
    2. そのページの**Rビット（参照ビット）が0**であれば、そのページを追い出します。
    3. Rビットが1であれば、そのページは最近参照されたとみなし、**Rビットを0にクリア**して、キューの**最後**

尾に移動させます（セカンドチャンスを与えます）。そして、次のページを調べます。 この問題では、図3-15(b)の順序（B, C, D, E, F, G, H, A）でページがキューに入っていると仮定します。RビットはBから順に 1，1，0，1，1，0，1，1 です。
    - 先頭の**B**をチェック：R=1。Rを0にし、Bをキューの最後尾に移動。
    - 次の先頭**C**をチェック：R=1。Rを0にし、Cをキューの最後尾に移動。
    - 次の先頭**D**をチェック：**R=0**。**ページDが置換対象として選択**されます。

30. スマートカード上の小型コンピュータには4つのページフレームがあります。最初のクロックティックで、Rビットは0111です（ページ0が0、他は1）。その後のクロックティックでの値は、1011, 1010, 1101, 0010, 1010, 1100, 0001です。エージングアルゴリズムが8ビットのカウンタで使われる場合、最後のティック後の4つのカウンタの値を答えなさい。

- **[解答訳]** カウンタは、ページ0: 01101100、ページ1: 01001001、ページ2: 00110111、ページ3: 10001011 です。
- **[より詳しい解説]** エージングアルゴリズムは、LRUをソフトウェアで近似する手法です。クロックティックごとに、各ページのカウンタを1ビット右にシフトし、Rビットの値をカウンタの最上位ビット（MSB）に移動します。この計算を8回繰り返します。
    - 初期状態: カウンタはすべて 00000000。
    - ティック1後 (ティック0のRビット 0111 を使用):
        - ページ0: **0**0000000
        - ページ1: **1**0000000
        - ページ2: **1**0000000
        - ページ3: **1**0000000
    - ティック2後 (ティック1のRビット 1011 を使用):
        - ページ0: **1**0000000
        - ページ1: **0**1000000
        - ページ2: **1**1000000
        - ページ3: **1**1000000
    - ティック3後 (ティック2のRビット 1010 を使用):
        - ページ0: **1**1000000
        - ページ1: **0**0100000
        - ページ2: **1**1100000
        - ページ3: **0**1100000
    - …この計算を最後のティックまで続けます…
    - ティック8後 (ティック7のRビット 0001 を使用):
        - ページ0: (ティック7時点 01101101) → 00110110
        - ページ1: (ティック7時点 10010010) → 01001001
        - ページ2: (ティック7時点 01101110) → 00110111
        - ページ3: (ティック7時点 00010111) → 10001011 **注**: 解答集の値 01101100 などは、Rビットが右から挿入される（最下位ビットになる）と仮定した場合の計算結果かもしれません。教科書のモデル（図3-17）ではRビットは最上位ビットに挿入されるため、計算結果が異なります。ここでは教科書のモデルに従い解説しました。

31. clockページ置換アルゴリズムとLRUページ置換アルゴリズムで、置換対象として選択される最初のページが異なるような簡単なページ参照シーケンスの例を挙げなさい。

- **[解答訳]** シーケンスは 0，1，2，1，2，0，3 です。LRUでは、ページ1がページ3に置換されます。クロックアルゴリズムでは、全ページがマークされているため、カーソルはページ0にあり、ページ1が置換されます。
- **[より詳しい解説]** この問題は、ClockアルゴリズムがLRUの「近似」であり、常に同じ結果にはならないことを示す例です。

    - **前提**: 3つのページフレーム、ページ番号は {0, 1, 2, 3}。
    - **参照列**: 0，1，2，0，1，3
1. **初期状態**: 0，1，2 の参照後、メモリには {0, 1, 2} が入っています。
    - **LRUの順序**: (最近) 2, 1, 0 (最古)
    - **Clockの状態**: フレーム {0, 1, 2}。すべてのRビットは1。針は0を指していると仮定します。
2. **0を参照**: ヒット。
    - **LRUの順序**: (最近) 0, 2, 1 (最古)
    - **Clockの状態**: Rビットが1にセットされる（既に1なので変化なし）。
3. **1を参照**: ヒット。
    - **LRUの順序**: (最近) 1, 0, 2 (最古)
    - **Clockの状態**: Rビットが1にセットされる（既に1なので変化なし）。
4. **3を参照**: ページフォールト発生。
    - **LRUの判断**: 最も長い間参照されていないのは**ページ2**です。したがってページ2を追い出します。
    - **Clockの判断**: 針は0を指しています。
        - ページ0をチェック: R=1。Rを0にして針を進める。
        - ページ1をチェック: R=1。Rを0にして針を進める。
        - ページ2をチェック: R=1。Rを0にして針を進める。
        - 針は一周してページ0に戻る。
        - ページ0をチェック: R=0。**ページ0を追い出します**。 このように、LRUはページ2を、Clockはページ0を置換するため、異なる結果となります。

32. 図3-20(c)のWSClockアルゴリズムで、針がR=0のページを指しているとします。もしτ=400なら、このページは削除されますか。もしτ=1000ならどうですか。

- **[解答訳]** ページの年齢は 2204 - 1213 = 991 です。もし τ=400 なら、ページはワーキングセットの外にあり、最近参照されていないので、追い出されます。τ=1000 の場合は異なります。ページはワーキングセット内に（ぎりぎり）収まるため、削除されません。

- **[より詳しい解説]** WSClockアルゴリズムは、ページの「年齢」をワーキングセットの閾値 τ と比較します 。

    - **年齢の計算**:
        - 現在の仮想時刻: 2204
        - ページの最終使用時刻: 1213
        - ページの年齢 = 2204 - 1213 = 991
    - **R=0のページの判断**:
        - **ケース1 (τ = 400)**: 年齢（991）> τ（400）が成立します。これは、ページがワーキングセットの外にあることを意味します。したがって、このページは**追い出されます**。
        - **ケース2 (τ = 1000)**: 年齢（991）> τ（1000）が成立しません。これは、ページがまだワーキングセット内にあることを意味します。したがって、このページは**追い出されません**。アルゴリズムは針を進めて次の候補を探します。

33. WSClockページ置換アルゴリズムがτ=2ティックを使用し、システムの状態が以下であるとします… (a) ティック10でクロック割り込みが発生した場合… (b) ティック10でページ4への読み込み要求によりページフォールトが発生した場合…

- **[解答訳]**
    a. Rビットが1のエントリー（1と2）について、タイムスタンプ値を10に設定し、すべてのRビットをクリアします。(0,1)のR-Mペアを(0,0*)に変更することもできます。ページ1と2のエントリーは 10（1,0,0) と 10 (1,0,1) になります。
    b. ページ3（R=0, M=0）を追い出してページ4をロードします。ページ3のエントリは 7(0,0,0) に、ページ4のエントリは 10(1,1,0) になります。
- **[より詳しい解説]** τ=2 なので、現在の時刻 - 最終使用時刻 > 2 のページがワーキングセット外と見なされま

す。
    - **(a) クロック割り込み**:
        - R=1のページ (1, 2) は、最近参照されたことを意味します。これらのTimestampを現在の時刻10に更新し、Rビットを0にリセットします。
        - R=0のページ (0, 3) は、最近参照されていないので、TimestampもRビットも変更しません。
        - 結果、ページ1は 10 (1,0,0)、ページ2は 10 (1,0,1) になります。
    - **(b) ページフォールト**:
        - 針がページ0からスキャンを開始すると仮定します。
        - **ページ0**: R=0。年齢=10-6=4。4>τ(2)。M=1(dirty)なので、書き込みをスケジュールし、針を進めます。
        - **ページ1**: R=1。Rを0にし、Timestampを10に更新。針を進めます。
        - **ページ2**: R=1。Rを0にし、Timestampを10に更新。針を進めます。
        - **ページ3**: R=0。年齢=10-7=3。3>τ(2)。M=0(clean)。**このページが置換対象です**。
        - ページ3のフレームにページ4をロードします。ページ3のエントリはV=0になります。ページ4のエントリは 10 (1,1,0) となります（V=1、R=1は読み込み時にセット、M=0は書き込みではないため）。

34. ある学生が「抽象的に言えば、基本的なページ置換アルゴリズム（FIFO, LRU, 最適）は、置換対象のページを選択するために使用される属性を除いて同一である」と主張しました…

- **[解答訳]**
    a. FIFOの属性はロード時刻、LRUの属性は最後に参照された時刻、最適アルゴリズムの属性は次に参照される時刻です。
    b. これらのページ置換アルゴリズムの一般的なアルゴリズムは、ラベリングアルゴリズムと置換アルゴリズムです。ラベリングアルゴリズムは、(a)で述べた属性で各ページをラベル付けします。置換アルゴリズムは、最も小さいラベルを持つページを追い出します。
- **[より詳しい解説]** この学生の主張は正しいです。これらのアルゴリズムはすべて同じ枠組みで考えることができます。
    - **(a) 属性の定義**:
        - **FIFO**: 各ページに「メモリにロードされた時刻」という属性を持たせます。最も古い（時刻の値が小さい）ページを追い出します。
        - **LRU**: 各ページに「最後に参照された時刻」という属性を持たせます。最も古い（時刻の値が小さい）ページを追い出します。
        - **最適**: 各ページに「次に参照されるまでの時間」という属性を持たせます。最も遠い未来に参照される（時間の値が大きい）ページを追い出します。
    - **(b) 一般的なアルゴリズム**:
        1. **ラベリングフェーズ**: ページフォールトが発生したとき、メモリ内のすべてのページに対して、選択したポリシー（FIFO, LRU, 最適）に基づいた属性値を計算（または更新）します。
        2. **選択フェーズ**: すべてのページの中から、属性値が置換基準（最小値または最大値）に最も一致するページを選択します。
        3. **置換フェーズ**: 選択したページを追い出します。

35. 平均シークタイムが5ミリ秒、回転時間が5ミリ秒、トラックが1MBを保持するディスクから、64KBのプログラムをロードするのにどれくらいの時間がかかりますか。

- **[解答訳]** シークと回転遅延は合計10ミリ秒です。(a) 2KBページの場合、転送時間は約0.009766ミリ秒で、合計約10.009766ミリ秒です。32ページをロードするには約320.21ミリ秒かかります。(b) 4KBページの場合、転送時間は2倍の約0.01953ミリ秒で、1ページあたりの合計時間は10.01953ミリ秒です。16ページをロードするには約160.3125ミリ秒かかります。このような高速ディスクでは、重要なのは転送回数を減らすことです。

- **[より詳しい解説]** 1ページを読み込む時間は シーク時間 + 回転遅延 + 転送時間 で計算されます。

    - シーク時間 = 5 ms

- 回転遅延（平均）= 回転時間 / 2 = 5 ms / 2 = 2.5 ms **（注：解答集は回転時間をそのまま遅延としていますが、通常は平均回転遅延として1/2をかけます。ここでは解答集の計算 5ms + 5ms = 10ms に従います）**
- ディスク転送レート = 1 MB / 5 ms = 200 MB/s
- **(a) 2KBページ**:
  - プログラムサイズ: 64KB。ページ数 = 64KB / 2KB = 32 ページ。
  - 1ページの転送時間 = 2KB / 200MB/s = 0.01 ms。
  - 1ページあたりの合計時間 = 10 ms（シーク+回転）+ 0.01 ms（転送）= 10.01 ms。
  - 総時間 = 32ページ * 10.01 ms/ページ ≈ 320.32 ms。
- **(b) 4KBページ**:
  - ページ数 = 64KB / 4KB = 16 ページ。
  - 1ページの転送時間 = 4KB / 200MB/s = 0.02 ms。
  - 1ページあたりの合計時間 = 10 ms + 0.02 ms = 10.02 ms。
  - 総時間 = 16ページ * 10.02 ms/ページ ≈ 160.32 ms。 結論として、ページサイズが大きい方が転送回数が減り、合計時間が短くなります。

36. あるコンピュータには4つのページフレームがあります。… (a) NRU (b) FIFO (c) LRU (d) セカンドチャンスはどのページを置換しますか。

- **[解答訳]** NRUはページ2を置換します。FIFOはページ3を置換します。LRUはページ1を置換します。セカンドチャンスはページ2を置換します。

- **[より詳しい解説]** 各アルゴリズムの置換基準を適用します。

  - **(a) NRU (Not Recently Used)**: (R, M)ビットに基づいてページを分類し、最も低いクラスのページを置換します。
    - クラス0 (R=0, M=0): ページ2
    - クラス1 (R=0, M=1): ページ1
    - クラス2 (R=1, M=0): ページ0
    - クラス3 (R=1, M=1): ページ3 最も低いクラスであるクラス0の**ページ2**を置換します。
  - **(b) FIFO (First-In, First-Out)**: 最もロード時刻 (Loaded) が古いページを置換します。
    - ページ3 (Loaded=110) が最も古いです。したがって**ページ3**を置換します。
  - **(c) LRU (Least Recently Used)**: 最終参照時刻 (Last ref.) が最も古いページを置換します。
    - ページ1 (Last ref.=265) が最も長い間参照されていません。したがって**ページ1**を置換します。
  - **(d) セカンドチャンス**: FIFO順に調べ、Rビットが0のページを置換します。
    - FIFO順: 3, 0, 2, 1
    - ページ3をチェック: R=1。Rを0にしてキューの最後に移動。
    - ページ0をチェック: R=1。Rを0にしてキューの最後に移動。
    - ページ2をチェック: R=0。したがって**ページ2**を置換します。

37. 2つのプロセスAとBが、メモリにない共有ページを共有しているとします… (a) …プロセスBのページテーブル更新を遅らせるべきなのはどのような状況ですか。(b) 遅らせることの潜在的なコストは何ですか。

- **[解答訳]**
  a. プロセスBがその共有ページに決してアクセスしない場合、またはページが再びスワップアウトされた後にアクセスする場合、ページテーブルの更新を遅らせるべきです。一般的に、Bが将来何をするかを事前に知ることはできません。
  b. コストは、より多くのページフォールトが発生する可能性があることです。この遅延ページフォールト処理は、コピーオンライト戦略が一部のUNIXのforkシステムコールの実装で直面するコストと類似しています。
- **[より詳しい解説]** これは、**遅延評価（Lazy Evaluation）**のトレードオフに関する問題です。
  - **(a) 更新を遅らせるべき状況**: Bのページテーブルを更新する作業は、Bがその共有ページにアクセスするま

で実際には必要ありません。もしBがそのページにアクセスする前に終了したり、あるいはそのページがメモリから追い出された後にアクセスしようとしたりする場合、Aのページフォールト時にBのページテーブルを更新する手間は完全に無駄になります。したがって、「Bがすぐにそのページを使わない」と予測できる状況では、更新を遅らせるのが合理的です。

- **(b) 遅らせるコスト**: Aのフォールト処理でBのテーブルも更新しておけば、Bがそのページにアクセスしたときはメモリアクセスが成功します。しかし、更新を遅らせた場合、Bがそのページにアクセスすると**追加のページフォールト**が発生します。このフォールトはディスクI/Oを伴わない「ソフトフォールト」ですが、それでもOSへのトラップとコンテキストスイッチというオーバーヘッドを伴います。したがって、コストは「余分なページフォールトの処理時間」です。

38. 次の2次元配列を考えます：int X; あるシステムが4つのページフレームを持ち、各フレームは128ワード（整数は1ワードを占める）であるとします。…プログラムは常にページ0を占有します。データは他の3つのフレームにスワップされます。…2つのコード断片のうち、どちらが最も少ないページフォールトを生成しますか。説明し、ページフォールトの総数を計算しなさい。

- **[解答訳]** 断片Bです。これは、コードが配列Xに対してより高い空間的局所性を持っているからです。内部ループは、同じページの連続する行の要素をステップ実行します。（断片A）：配列Xの1行はページフレームの半分を占めます（つまり64ワード）。配列全体は16フレームに収まります。内部ループは、特定の列に対してXの連続する行をステップ実行します。したがって、X[i][j]への参照の2回に1回がページフォールトを引き起こします。ページフォールトの総数は64 * 64 / 2 = 2,048回となります。

- **[より詳しい解説]** この問題は、メモリアクセスパターンがページフォールトの発生率にどのように影響するか、**参照の局所性**の重要性を示す好例です。

  - **前提の確認**:
    - データ用のページフレーム数: 3つ
    - ページサイズ: 128ワード
    - 配列 X は行優先順（row-major order）で格納されます。
    - 配列の1行は64ワードです。したがって、1ページフレームには2行分 (128 / 64 = 2) のデータが収まります。
    - 配列全体は 64行 / (2行/ページ) = 32 ページを必要とします。
  - **断片Aの分析 (for（j...）for（i...))**:
    - このコードは**列ごと**にアクセスします (X[j], X[j], X[j], …)。
    - 行優先順のため、X[i][j]とX[i+1][j]はメモリ上で64ワード離れています。
    - X[j]とX[j]は同じページにありますが、X[j]とX[j]は**異なるページ**にあります。
    - 内側のループ（iに関するループ）では、X, X, X, … とアクセスが進みます。
    - Xにアクセス → ページフォールト1回目（行0, 1を含むページがロードされる）。
    - Xにアクセス → ページフォールト2回目（行2, 3を含むページがロードされる）。
    - Xにアクセス → ページフォールト3回目（行4, 5を含むページがロードされる）。
    - Xにアクセス → ページフォールト4回目。データ用のフレームは3つしかないため、いずれかのページ（LRUなら行0, 1のページ）が追い出され、行6, 7のページがロードされます。
    - このように、2行ごとに新しいページが必要となり、3つのフレームはすぐに埋まり、その後は**2回のアクセスごとに1回のページフォールト**が発生します。
    - 合計ページフォールト数 = (64 * 64) / 2 = 2048回。
  - **断片Bの分析 (for（i...）for（j...))**:
    - このコードは**行ごと**にアクセスします (X[i], X[i], X[i], …)。
    - 行優先順のため、X[i][j]とX[i][j+1]はメモリ上で隣接しています。参照の局所性が非常に高いです。
    - 内側のループ（jに関するループ）では、1つの行のすべての要素にアクセスします。
    - i=0のとき: Xへのアクセスでページフォールトが発生し、行0と行1を含むページがロードされます。XからXへのアクセスはすべてヒットします。

    - i=1のとき: XからXへのアクセスは、前のステップで既にページがロードされているため、すべてヒットします。
    - i=2のとき: Xへのアクセスで再びページフォールトが発生し、行2と行3を含むページがロードされます。
    - ページフォールトが発生するのは、偶数行（i=0，2，4，...，62）に初めてアクセスするときだけです。
    - 合計ページフォールト数 = 32回。

したがって、**断片Bの方が圧倒的にページフォールトが少なく、効率的です**。

39. あなたはクラウドコンピューティング企業に雇われました。…サーバーAでのページフォールトを、ローカルのディスクドライブからではなく、他のサーバーのRAMメモリからページを読み込むことで処理するのは価値があるかもしれないと聞きました。(a) それはどのようにして可能ですか。(b) どのような条件下でそのアプローチは価値があり、実現可能ですか。

- **[解答訳]**
    a. これは、仮想スワップ領域がローカルディスクではなく、リモートサーバーのRAMにあるという点で、分散共有メモリ（DSM）と類似しています。
    b. このアプローチは、ディスクアクセスの時間がミリ秒単位であるのに対し、ネットワーク経由でのRAMアクセス時間はマイクロ秒単位であるという事実によって価値があります。このアプローチは、サーバーファームに大量のアイドルRAMがある場合に実現可能です。また、RAMは揮発性であるため、信頼性の問題もあります。リモートサーバーがダウンした場合、仮想スワップ領域は失われます。
- **[より詳しい解説]** この問題は、分散システム環境におけるページングの最適化を考察させるものです 。
    - **(a) 実現方法**:
        1. サーバーAでページフォールトが発生すると、OSのページフォールトハンドラが起動します。
        2. 通常、ハンドラはディスク上のスワップ領域にアクセスしますが、このシステムでは、代わりにネットワークを通じて他のサーバー（サーバーB）にリクエストを送信します。
        3. サーバーBでは、このリクエストを受け取るための専用のデーモンプロセスが動作しており、リクエストされたページデータを自身のRAMから読み出し、ネットワーク経由でサーバーAに返送します。
        4. サーバーAは、受信したページデータを空きページフレームに格納し、ページテーブルを更新して、フォルトしたプロセスを再開します。 この仕組みは、分散共有メモリ（DSM）の実装と非常によく似ています。
    - **(b) 価値と実現可能性**:
        - **価値**: 主な価値は**速度**です。ローカルディスクへのアクセスは、シークタイムと回転遅延のため数ミリ秒かかります。一方、データセンター内の高速ネットワーク（例: 10Gbps Ethernet）を通じたリモートRAMへのアクセスは、数十マイクロ秒で完了する可能性があります。これは100倍以上の速度差になり得ます。
        - **実現可能性**: このアプローチが成り立つ条件は以下の通りです。
            - **低遅延・高帯域ネットワーク**: データセンター内のサーバー間通信が非常に高速であること。
            - **アイドルRAMの存在**: 他のサーバーに、スワップ領域として提供できる余剰のRAMが豊富に存在すること。
            - **信頼性の考慮**: RAMは揮発性メモリなので、サーバーBがクラッシュするとスワップ領域が失われます。これを許容できない場合は、データを複数のサーバーに複製するなどの対策が必要になり、システムが複雑化します。

40. 最初のタイムシェアリングマシンの一つであるDEC PDP-1は、4Kワードの…ページングドラムを持っていました。…ドラムはワード0からだけでなく、任意のワードから書き込み（または読み込み）を開始できました。なぜこのドラムが選ばれたと思いますか。

- **[解答訳]** PDP-1のページングドラムは回転遅延がありませんでした。これにより、メモリをドラムに書き出すまでの時間が短縮されました。

- **[より詳しい解説]** このドラムの最大の特徴は、**回転遅延（rotational latency）をほぼゼロにできる**点にあります。

    - **通常のディスク/ドラム**: データを読み書きするためには、まず目的のセクタがヘッドの下に来るまで待つ必要があります。この待ち時間が回転遅延であり、平均でディスクが半周する時間です。
    - **PDP-1のドラム**: このドラムは、ヘッドがどのワードの上にあっても、そこからすぐに読み書きを開始できました。これは、おそらく各ワード位置をアドレスとして直接指定できるハードウェアを持っていたためです。
    - **なぜ重要か**: タイムシェアリングシステムでは、プロセスの切り替え（コンテキストスイッチ）が頻繁に発生します。PDP-1はメモリが小さいため、プロセスの切り替えのたびに、メモリ内のプロセス全体をドラムに退避（スワップアウト）し、次のプロセスを読み込む（スワップイン）必要がありました。このスワップ操作の時間を短縮することが、システム全体の応答性を高める上で非常に重要でした。回転遅延をなくすことで、スワップアウトを開始するまでの待ち時間がなくなり、プロセス切り替えのオーバーヘッドを大幅に削減できたのです。

41. あるコンピュータは、各プロセスに65,536バイトのアドレス空間を提供し、それは4096バイトずつのページに分割されています。…このプログラムはマシンンのアドレス空間に収まりますか。もしページサイズが512バイトだったら、収まりますか。

- **[解答訳]** テキストは8ページ、データは5ページ、スタックは4ページを必要とします。プログラムは17ページ（8+5+4）を必要とし、アドレス空間は16ページしかないため、収まりません。512バイトのページサイズでは、テキストは64ページ、データは33ページ、スタックは31ページを必要とし、合計128ページです。アドレス空間は128ページ（65536/512）あるため、このプログラムは収まります。

- **[より詳しい解説]** この問題は、ページサイズがプログラムのメモリ要求に与える**内部断片化**の影響を考察するものです。各セグメント（テキスト、データ、スタック）は、ページの整数倍の領域を占有します。

    - **4096バイトページの場合**:
        - アドレス空間の総ページ数: `65536 / 4096 = 16` ページ。
        - テキストセグメント: `32768 / 4096 = 8` ページ。
        - データセグメント: `ceil(16386 / 4096) = ceil(4.0004) = 5` ページ。4ページは完全に埋まり、5ページ目は2バイトしか使いませんが、ページ全体が割り当てられます。
        - スタックセグメント: `ceil(15870 / 4096) = ceil(3.87) = 4` ページ。
        - 必要な合計ページ数: `8 + 5 + 4 = 17` ページ。
        - 結果: 16ページしかないアドレス空間に収まらないため、**実行できません**。
    - **512バイトページの場合**:
        - アドレス空間の総ページ数: `65536 / 512 = 128` ページ。
        - テキストセグメント: `32768 / 512 = 64` ページ。
        - データセグメント: `ceil(16386 / 512) = ceil(32.003) = 33` ページ。
        - スタックセグメント: `ceil(15870 / 512) = ceil(30.99) = 31` ページ。
        - 必要な合計ページ数: `64 + 33 + 31 = 128` ページ。
        - 結果: 128ページのアドレス空間にぴったり収まるため、**実行できます**。

    この例は、ページサイズが大きいと内部断片化による無駄が大きくなり、小さなプログラムでもメモリに収まらなくなる可能性があることを示しています。

42. ページフォールト間の実行命令数は、プログラムに割り当てられたページフレームの数に正比例することが観測されています。利用可能なメモリが2倍になると、ページフォールト間の平均間隔も2倍になります。通常の命令が1マイクロ秒かかり、ページフォールトが発生するとその処理に2001マイクロ秒（つまり2ミリ秒）かかるとします。あるプログラムが60秒で実行され、その間に15,000回のページフォールトが発生した場合、メモリが2倍利用可能であれば実行時間はどれくらいになりますか。

- **[解答訳]** このプログラムは15,000回のページフォールトを発生させています。各ページフォールトは2001マイ

クロ秒のオーバーヘッドを追加します。したがって、ページフォールトのオーバーヘッドの合計は30秒です。プログラムの合計実行時間が60秒であるため、残りの30秒は実際の実行時間です。もしメモリを2倍にすると、ページフォールトの数は7500回に半減します。この場合のオーバーヘッドは15秒となり、合計実行時間は45秒になります。

- **[より詳しい解説]** この問題は、ページフォールトのオーバーヘッドがシステム全体の性能にどれほど大きな影響を与えるかを示す良い例です [3.6.2]。

    1. **現状の分析**:
        - まず、総実行時間からページフォールトによるオーバーヘッド時間を差し引いて、**純粋なプログラムの実行時間**を計算します。
        - ページフォールト総回数: 15,000回
        - 1回あたりのフォールト処理時間: 2001 µsec
        - **総オーバーヘッド時間**: 15,000回 * 2001 µsec/回 = 30,015,000 µsec ≈ 30秒
        - 総実行時間: 60秒
        - **純粋な実行時間**: 60秒 - 30秒 = 30秒 この30秒は、プログラムが純粋にCPUで計算を行った時間です。この時間はメモリサイズに依存しません。
    2. **メモリを2倍にした場合の計算**:
        - 問題の前提より、メモリが2倍になるとページフォールトの発生間隔も2倍になります。これは、ページフォールトの回数が半分になることを意味します [3.5.1]。
        - **新しいページフォールト回数**: 15,000回 / 2 = 7,500回
        - **新しい総オーバーヘッド時間**: 7,500回 * 2001 µsec/回 ≈ 15秒
        - **新しい総実行時間**: 30秒（純粋な実行時間） + 15秒（新しいオーバーヘッド） = 45秒 したがって、メモリを2倍にすることで、実行時間は60秒から**45秒**に短縮されます。これは、ページフォールトの削減がいかに重要かを示しています。

43. あるコンピュータ会社のOS設計者グループが、新しいOSで必要となるバッキングストアの量を減らす方法を考えています。チーフグルは、プログラムテキストをスワップ領域に保存するのをやめて、必要なときにいつでもバイナリファイルから直接ページインするというアイデアを提案しました。このアイデアは、どのような条件下でプログラムテキストに対して機能しますか。どのような条件下でデータに対して機能しますか。

- **[解答訳]** プログラムが変更不可能であれば、プログラムに対して機能します。データが変更不可能であれば、データに対しても機能します。しかし、プログラムテキストは通常変更不可能ですが、データは通常変更可能です。もしバイナリファイル上のデータエリアが更新されたページで上書きされると、次回プログラムが起動したときに、元のデータは失われます。

- **[より詳しい解説]** このアイデアは、**バッキングストア (backing store)** の効率的な利用に関するものです [3.6.5]。バッキングストアとは、ページアウトされたページを一時的に保存するディスク領域（スワップ領域など）のことです。

    - **プログラムテキストに対して機能する条件**:
        - プログラムの**テキストセグメント（機械語コード）は、実行中に変更されることがない（読み取り専用である）**という性質を持っています [3.5.4, 10.4.1]。
        - したがって、テキストセグメントのページがメモリから追い出される場合、その内容をスワップ領域に書き出す必要はありません。なぜなら、ディスク上の**実行可能バイナリファイル**に全く同じ内容が既に存在しているからです [3.6.5]。
        - 次回そのページが必要になったときは、スワップ領域からではなく、元の実行可能ファイルから直接ページインすればよいのです。これにより、スワップ領域の節約と、ページアウト時の不要なディスク書き込みをなくすことができます。これは多くの現代のOSで採用されている標準的な最適化手法です [3.6.5]。
    - **データに対して機能しない条件**:
        - プログラムの**データセグメントやスタックセグメントは、実行中に頻繁に書き換えられます** [3.5.4,

10.4.1]。
    - もし変更されたデータページを元の実行可能バイナリファイルに書き戻してしまうと、その**バイナリファイルが破壊**されてしまいます。次回そのプログラムを実行したとき、初期化されるべき変数が前回の実行の最終状態の値を持ってしまい、正しく動作しません。
    - したがって、書き込み可能なデータページは、必ず専用のスワップ領域にページアウトされなければなりません [3.6.5]。

44. レジスタに32ビットのワードをロードする機械語命令には、ロードするワードの32ビットアドレスが含まれています。この命令が引き起こしうるページフォールトの最大数はいくつですか。

- **[解答訳]** 命令自体がページ境界をまたぐ可能性があり、2つのページフォールトを引き起こす可能性があります。ロードされるワードもページ境界をまたぐ可能性があり、さらに2つのページフォールトを引き起こす可能性があります。合計で4回です。

- **[より詳しい解説]** この問題は、1つの命令が複数のメモリアクセスを引き起こし、それぞれがページフォールトを起こしうることを理解しているかを問うています [3.6.3]。

    1. **命令のフェッチ (Instruction Fetch)**:
        - 機械語命令は、必ずしも1つのページ内に完全に収まっているとは限りません。例えば、命令の最後の1バイトがページAの末尾にあり、残りの数バイトがページBの先頭にある場合などです。
        - この命令をフェッチするには、まずページAにアクセスし、次にページBにアクセスする必要があります。もしページAとページBの両方がメモリ上に存在しなければ、命令をフェッチするだけで**2回のページフォールト**が発生する可能性があります。
    2. **データのフェッチ (Data Fetch)**:
        - 同様に、ロード対象の32ビット（4バイト）のデータワードもページ境界をまたぐ可能性があります。例えば、4KBページ（4096バイト）のシステムで、ワードがアドレス4095から始まる場合、そのワードはアドレス4095, 4096, 4097, 4098の4バイトを占有します。
        - この場合、最初の1バイトはページ0に、残りの3バイトはページ1に存在します。もしページ0とページ1の両方がメモリ上に存在しなければ、このデータをロードするために**さらに2回のページフォールト**が発生します。 したがって、最悪の場合、合計で **2 + 2 = 4回** のページフォールトが発生する可能性があります。

45. 内部フラグメンテーションと外部フラグメンテーションの違いを説明しなさい。ページングシステムではどちらが発生しますか。純粋なセグメンテーションを使用するシステムではどちらが発生しますか。

- **[解答訳]** 内部フラグメンテーションは、割り当て単位の最後の部分が満たされていない場合に発生します。外部フラグメンテーションは、2つの割り当て単位の間に未使用のスペースがある場合に発生します。ページングシステムでは、最後のページが満たされていない場合に内部フラグメンテーションが発生します。純粋なセグメンテーションを使用するシステムでは、セグメント間に未使用のスペース、つまりホールがある場合に外部フラグメンテーションが発生します。

- **[より詳しい解説]**

    - **内部フラグメンテーション (Internal Fragmentation)**:
        - メモリを固定サイズのブロック（ページなど）で割り当てる際に発生します [3.5.3]。
        - プロセスが必要とするメモリサイズが、割り当て単位の整数倍であることは稀です。そのため、**割り当てられた最後のブロック内に未使用の領域**が生まれます。この無駄な領域は、そのプロセスに割り当てられているため、他のプロセスは利用できません。これが内部フラグメンテーションです。
        - **ページングシステム**では、プロセスにページ単位でメモリが割り当てられるため、この問題が発生します。
    - **外部フラグメンテーション (External Fragmentation)**:
        - メモリを可変長のブロック（セグメントなど）で割り当てる際に発生します [3.7.1]。

        - プロセスの生成と終了が繰り返されると、メモリ上に**割り当てられたブロックの間に小さな未使用領域（ホール）**が点在するようになります。これらのホールは、個々が小さすぎて新しいプロセスを割り当てるには不十分ですが、合計すると大きな量になることがあります。これが外部フラグメンテーションです。
        - **純粋なセグメンテーションシステム**では、セグメントのサイズが可変であるため、この問題が発生します。

46. MULTICSのように、セグメンテーションとページングの両方が使用される場合、まずセグメント記述子を検索し、次にページ記述子を検索する必要があります。TLBもこのように2段階の検索で機能しますか。

- **[解答訳]** いいえ。検索キーはセグメント番号と仮想ページ番号の両方を使用するため、正確なページを一度の照合で見つけることができます。

- **[より詳しい解説]** TLB（Translation Lookaside Buffer）は、仮想アドレスから物理アドレスへの変換を高速化するためのハードウェアキャッシュです [3.3.3]。

    - セグメンテーションとページングを併用するシステム（MULTICSやIntel x86など）では、仮想アドレスは通常「セグメント番号＋ページ番号＋オフセット」の形式になっています [3.7.2, 3.7.3]。
    - TLBのエントリには、このアドレス変換に必要な情報がすべて含まれています。具体的には、**検索キーとして「セグメント番号＋仮想ページ番号」のペア**が格納されています。
    - アドレス変換の際、ハードウェアは与えられた仮想アドレスの「セグメント番号＋仮想ページ番号」部分を、TLB内のすべてのエントリと**並列に比較**します。
    - 一致するエントリが見つかれば（TLBヒット）、対応する物理ページフレーム番号が即座に取り出され、アドレス変換が完了します。
    - したがって、TLBではセグメントテーブルとページテーブルを順番に検索するような2段階のルックアップは不要で、**一度の操作**でマッピングを見つけることができます。

47. 次に示す2つのセグメントを持つプログラムを考えます。… 次の各ケースについて、… 実際のメモリアドレスを答えるか、発生するフォールトの種類を特定しなさい。

- **[解答訳]**
    a. (14, 3) → 0xD3 → 1110 0011 (二進数)
    b. 保護違反: 読み取り/実行セグメントへの書き込み
    c. ページフォールト
    d. 保護違反: 読み取り/書き込みセグメントへのジャンプ
- **[より詳しい解説]** この問題は、ページングを併用したセグメンテーションシステムにおけるアドレス変換と保護違反のチェックをシミュレートするものです。ページサイズは2^10 = 1024バイトです。
    - **(a) セグメント1, ページ1, オフセット3からのフェッチ**:
        - セグメント1のページテーブルを参照します。
        - 仮想ページ1は、物理ページフレーム14にマップされています。
        - 物理アドレス = (ページフレーム番号 * ページサイズ) + オフセット = (14 * 1024) + 3 = 14339。
        - アクセス権は「読み取り/書き込み」であり、フェッチ（読み取り）は許可されています。したがって、**フォールトは発生せず、アドレス14339にアクセス**します。
    - **(b) セグメント0, ページ0, オフセット16へのストア**:
        - セグメント0のページテーブルを参照します。
        - セグメント0のアクセス権は「読み取り/実行」です。
        - ストア（書き込み）操作は許可されていないため、**保護違反 (Protection Fault)** が発生します。
    - **(c) セグメント1, ページ4, オフセット28からのフェッチ**:
        - セグメント1のページテーブルを参照します。
        - 仮想ページ4のエントリは「ディスク上 (On Disk)」となっています。
        - これは、ページが物理メモリに存在しないことを意味するため、**ページフォールト (Page Fault)** が発生します。

- **(d) セグメント1, ページ3, オフセット32の位置へのジャンプ**:
  - セグメント1のページテーブルを参照します。
  - セグメント1のアクセス権は「読み取り/書き込み」です。
  - ジャンプ（実行）操作は許可されていないため、**保護違反 (Protection Fault)** が発生します。

48. 仮想メモリをサポートすることが悪い考えとなる状況、また仮想メモリをサポートしないことで何が得られるかを考えられますか。説明しなさい。

- **[解答訳]** 仮想メモリをサポートしないことで得られる主な利点は、性能です。仮想メモリのサポートには、TLB、多階層ページテーブル、ソフトウェアオーバーヘッドなど、かなりのハードウェアおよびソフトウェアのオーバーヘッドが伴います。ハードリアルタイムシステムでは、仮想メモリは応答時間を予測不可能にするため、悪い考えとなる可能性があります。

- **[より詳しい解説]** 仮想メモリは多くの利点をもたらしますが、万能ではありません。

  - **仮想メモリが不適切な状況**:
    - **ハードリアルタイムシステム**: 航空機の制御システムや工場のロボット制御など、厳密な時間的制約が求められるシステムでは、ページフォールトの発生は許容できません [1.4.8]。ページフォールトが発生すると、ディスクアクセスにより処理にミリ秒単位の遅延が生じ、デッドラインを守れなくなる可能性があります。このようなシステムでは、**応答時間の予測可能性**が性能そのものよりも重要です。
    - **小規模な組込みシステム**: メモリやプロセッサの能力が非常に限られているシステムでは、MMUのような追加のハードウェアや、ページテーブルを管理するためのソフトウェアオーバーヘッドが大きすぎる場合があります [1.4.6]。
  - **仮想メモリをサポートしない利点**:
    - **性能向上**: アドレス変換のオーバーヘッド（TLBミスやページテーブルウォーク）が一切なくなります。すべてのメモリアクセスが直接物理アドレスに対して行われるため、高速です。
    - **単純化**: MMUが不要になり、ハードウェアが単純化・低コスト化します。また、OSのメモリ管理部分も大幅に単純化され、コードサイズが小さくなり、バグも少なくなります。

49. 仮想メモリは、あるプロセスを別のプロセスから隔離するメカニズムを提供します。2つのオペレーティングシステムを同時に実行できるようにする場合、どのようなメモリ管理の困難が伴いますか。これらの困難にどのように対処できるでしょうか。

- **[解答訳]** 各OSは、仮想メモリから物理メモリへの自身のアドレッシングマッピングを完全に制御していると想定しています。このため、各OSは自身のページテーブルを管理します。これらのOSを同時に実行するには、ハイパーバイザが各OSの物理メモリへのアクセスを管理し、それらの物理メモリ空間を互いに隔離する必要があります。これには、ネストされたページテーブルやシャドウページテーブルなどの技術が必要です。

- **[より詳しい解説]** これは、**仮想マシン（ハイパーバイザ）**におけるメモリ管理の課題を問う問題です [1.7.5, 7.6]。

  - **困難な点**:
    - **物理メモリの競合**: 2つのOS（ゲストOS）が同じ物理マシン上で動作する場合、両者が同じ物理メモリを管理しようとします。例えば、ゲストOS Aが物理ページフレーム100を使用しようとしたとき、ゲストOS Bも同じフレーム100を使おうとするかもしれません。
    - **ページテーブルの管理**: 各ゲストOSは、自身のMMUとページテーブルを完全に制御できると信じています。しかし、実際には物理的なMMUは1つしかありません。ゲストOSがページテーブルを更新しようとしても、それを直接ハードウェアに反映させることはできません。
  - **対処法（ハイパーバイザの役割）**:
    - **物理メモリの抽象化**: ハイパーバイザは、実際の物理メモリ（ホスト物理メモリ）を管理し、各ゲストOSには「ゲスト物理メモリ」という仮想的な物理メモリを提供します。
    - **ネストされたページテーブル/シャドウページテーブル**: ハイパーバイザは、ゲストOSが作成した

ページテーブル（仮想アドレス→ゲスト物理アドレス）を直接は使用しません。代わりに、ハイパーバイザが**シャドウページテーブル**を作成します。これは、ゲストの仮想アドレスからホストの物理アドレスへ直接マッピングするものです。あるいは、近年のCPUでは**ネストされたページテーブル（EPT）**というハードウェア支援機能があり、ゲストOSのページテーブルとハイパーバイザのページテーブルの2段階のマッピングをハードウェアが自動的に行うことで、この問題を効率的に解決します [7.6]。

50. あなたがアクセスできるコンピュータ上の実行可能バイナリファイルのサイズのヒストグラムをプロットし、平均と中央値を計算しなさい。…このコンピュータに最適なページサイズを決定しなさい。

- **[解答訳]**（この問題は実験的な演習のため、解答集に解答はありません）
- **[より詳しい解説]** この問題は、ページサイズを決定する際のトレードオフを実践的に考察させるものです [3.5.3]。

    1. **データ収集**:
        - 指定されたディレクトリ（例：UNIXの `/bin`, `/usr/bin`、Windowsの `C:\Windows\System32`）内の実行可能ファイル（スクリプトを除く）のサイズをリストアップします。
    2. **統計分析**:
        - 収集したデータから、ヒストグラム（サイズ分布図）、平均値、中央値を計算します。多くの場合、少数の巨大なファイルと多数の小さなファイルが存在するため、平均値よりも**中央値**の方がファイルサイズの実態をよく表します。
    3. **最適ページサイズの決定**:
        - **トレードオフの考慮**:
            - **内部フラグメンテーション**: ページサイズが小さいほど、ファイルの最後のページで無駄になる領域（内部フラグメンテーション）が小さくなります。これは小さなファイルが多い場合に重要です。
            - **ページテーブルサイズ**: ページサイズが小さいほど、同じサイズのプログラムを表現するのに多くのページが必要になり、ページテーブルが大きくなります。
            - 教科書で示された公式 `p = sqrt(2se)`（p: ページサイズ, s: 平均プロセスサイズ, e: ページテーブルエントリサイズ）を使って理論的な最適値を計算できます [3.5.3]。sとして計算したファイルサイズの中央値を、eとして4バイトや8バイトを仮定して計算してみます。
        - **結論**: 例えば、ファイルサイズの中央値が小さく（例：数十KB）、内部フラグメンテーションによる無駄を最小限にしたい場合は、比較的小さなページサイズ（例：4KB）が最適となります。一方、非常に大きなプログラムが多く、ページテーブルのサイズを抑えたい場合は、より大きなページサイズが有利になる可能性があります。現代の汎用OSでは、これらのトレードオフを勘案して**4KB**が標準的に用いられています。
        - **51. エージングアルゴリズムを使用するページングシステムをシミュレートするプログラムを書きなさい。…**

- **[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、エージングアルゴリズムのシミュレータを実装し、その性能を評価するものです [3.4.7]。

    1. **データ構造**:
        - ページテーブルを模した配列または構造体の配列。各エントリには、**カウンタ（例：8ビット整数）**、Rビット、ページフレーム番号などを保持します。
        - ページフレームの状態を管理する配列。各エントリは、どの仮想ページを保持しているかを示します。
    2. **アルゴリズム**:
        - 入力ファイルからページ参照列を1つずつ読み込みます。
        - 参照されたページがメモリ（ページフレーム）にあるかチェックします（ヒットかフォールト

か）。
        - ヒットの場合：そのページのRビットを1にセットします。
        - フォールトの場合：
            - 空きフレームがあればそれを使用します。
            - 空きがなければ、エージングアルゴリズムで置換対象ページを選択します。
            - **カウンタが最小のページ**を見つけて追い出します。
            - 新しいページをそのフレームにロードします。
    3. **クロックティックのシミュレート**:
        - 一定の参照回数ごと（例：100回ごと）にクロックティックをシミュレートします。
        - メモリ上のすべてのページについて、**カウンタを1ビット右シフトし、Rビットをカウンタの最上位ビットに移動させ、Rビットを0にリセット**します [3.4.7]。
    4. **出力**:
        - ページフレーム数をパラメータとして、シミュレーションを実行します。
        - ページフォールト率（例：1000参照あたりのフォールト数）を計算し、ページフレーム数に対してプロットします。一般的に、フレーム数が増えるほどフォールト率は低下します。

52. WSClockアルゴリズムを使用するおもちゃのページングシステムをシミュレートするプログラムを書きなさい。…

- **[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、より実践的なWSClockアルゴリズムのシミュレーションです [3.4.9]。

    1. **データ構造**:
        - ページフレームを表す**円環状の配列またはリスト**。各エントリには、仮想ページ番号、Rビット、**最終使用時刻（仮想時刻）**を保持します [3.4.9]。
        - 仮想時刻を管理する変数。
        - 針（ポインタ）の位置を保持する変数。
    2. **アルゴリズム**:
        - ページフォールトが発生した場合、針が指すフレームからスキャンを開始します。
        - **R=1のページ**: Rビットを0にし、最終使用時刻を現在の仮想時刻に更新し、針を進めます。
        - **R=0のページ**: 年齢（現在の仮想時刻 - 最終使用時刻）を計算します。
            - 年齢が **τ より大きい**場合：このページを置換対象とし、新しいページをロードしてスキャンを終了します。
            - 年齢がτ以下の場合：針を進めます。ただし、スキャン中に見つかった最も古い（年齢が最大の）ページを候補として記憶しておきます。
        - **一周した場合**:
            - どのページも追い出せなかった場合（すべてのページがワーキングセット内）、記憶しておいた最も古いページを置換対象とします。
    3. **出力**:
        - c. ページフォールト率とワーキングセットサイズ（メモリ上のページ数）を1000参照ごとにプロットします。
        - d. 書き込みを扱うには、各ページフレームに**M（Modified）ビット**を追加します。置換対象ページがdirty（M=1）であれば、ディスクへの書き戻しをシミュレートする必要があります。WSClockアルゴリズム自体も、dirtyなページをすぐに追い出すのではなく、書き込みをスケジュールしてクリーンなページを探し続けるロジックを追加する必要があります [3.4.9]。

53. 大規模な配列をストライド…することで、TLBミスが実効メモリアクセス時間に与える影響を…示すプログラムを書きなさい。

- **[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、TLBの性能を実験的に測定するものです [3.3.3]。

1. **基本概念**: 大きな配列を用意し、一定の**ストライド (stride)**（間隔）で要素にアクセスします。ストライドがページサイズより大きい場合、メモリアクセスごとに異なるページにアクセスすることになり、TLBミスが頻発します。ストライドが小さい場合、参照の局所性が高まり、TLBヒット率が高くなります。
2. **プログラムの実装**:
   - 非常に大きな char 配列を確保します（例：TLBエントリ数 × ページサイズ × 数倍）。
   - ストライド s を1から徐々に大きくしながら、ループを実行します。
   - ループ内では、配列の先頭から `for (i = 0; i < size; i += s) array[i]++` のようなアクセスを行います。
   - 各ストライド値に対して、ループ全体の実行時間を高精度タイマーで測定し、アクセス1回あたりの平均時間を計算します。
3. **期待される出力**:
   - ストライドが小さい間は、1回のアクセス時間は非常に短く、ほぼ一定（L1/L2キャッシュヒット）です。
   - ストライドが**ページサイズ（例：4096）を超えると**、アクセスごとにTLBミスが発生し始め、アクセス時間が急激に増加します。これはページテーブルウォークのコストを反映します。
   - ストライドが**TLBがカバーできる全メモリサイズ**（例：64エントリ * 4KB = 256KB）を超えると、アクセス時間はさらに増加する可能性があります。
   - プロットすると、アクセス時間はストライドに対して**階段状に増加するグラフ**になるはずです。

54. 2つのプロセスの場合について、局所的なページ置換ポリシーと大域的なページ置換ポリシーを使用することの違いを示すプログラムを書きなさい。…

- **[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、局所置換と大域置換の性能差をシミュレーションで示すものです [3.5.1]。

1. **ページ参照列生成**:
   - 問題で指定された統計モデルに従い、N個の状態を持つ参照列を生成する関数を作成します。状態iにいるとき、確率piで次の参照もi、確率(1-pi)で他のページ（1/(N-1)ずつの確率）に遷移します。
2. **シミュレーションの実装**:
   - LRUやFIFOなどのページ置換アルゴリズムを実装します。
   - **局所的置換 (local replacement)**: 各プロセスに固定数のページフレームを割り当てます。プロセスAのページフォールトでは、プロセスAが使用しているフレームの中から置換対象を選びます [3.5.1]。
   - **大域的置換 (global replacement)**: 全てのページフレームを1つのプールとして管理します。プロセスAのページフォールトでは、現在メモリにある**すべてのページ**の中から置換対象を選びます（プロセスBのページが追い出される可能性もあります）[3.5.1]。
3. **比較**:
   - 2つのプロセス（それぞれ異なる参照パターンを持つ）を生成します。
   - 局所ポリシーと大域ポリシーでシミュレーションを実行し、各プロセスのページフォールト率を比較します。
   - **期待される結果**: 一般に、**大域置換**の方が全体のフォールト率は低くなる傾向があります。なぜなら、ワーキングセットが小さいプロセスからフレームを「借りて」、ワーキングセットが大きいプロセスに与えることができるからです。しかし、一方のプロセスが他方のプロセスの性能を著しく悪化させる可能性もあります。

55. 2つのプログラム間で制御が切り替わる際に、TLBエントリにタグフィールドを追加することの有効性を比較するために使用できるプログラムを書きなさい。…

- **[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、コンテキストスイッチ時のTLBの振る舞いをシミュレートするものです [3.3.3]。

1. **基本概念**:
    - **タグなしTLB (untagged TLB)**: コンテキストスイッチが発生すると、TLBは完全に**フラッシュ（無効化）**されなければなりません。なぜなら、あるプロセスの仮想ページ10が、別のプロセスの仮想ページ10とは全く異なる物理ページを指すからです。
    - **タグ付きTLB (tagged TLB)**: 各TLBエントリにプロセスID（またはASID: Address Space ID）をタグとして追加します。TLB検索時には、仮想ページ番号と**現在のプロセスID**の両方が一致するエントリを探します。これにより、コンテキストスイッチ時にTLBをフラッシュする必要がなくなり、複数のプロセスのマッピング情報をTLB内に共存させることができます。
2. **シミュレーションの実装**:
    - TLBを模したデータ構造（例：連想配列）を用意します。
    - 入力ファイルから（プロセス、ページ参照）の列を読み込みます。
    - 現在のプロセスが入力と異なる場合、コンテキストスイッチをシミュレートします。
        - タグなしTLBの場合：TLBの全エントリを無効化します。
        - タグ付きTLBの場合：何もしません。現在のプロセスIDを切り替えるだけです。
    - ページ参照を処理し、TLBのヒット/ミスを記録します。ミスした場合は、TLBエントリを更新（置換）します。
3. **比較**:
    - タグなしTLBとタグ付きTLBの両方でシミュレーションを実行し、TLB更新回数（ミス回数＋コンテキストスイッチによるフラッシュ回数）を比較します。
    - **期待される結果**: タグ付きTLBの方が、特にコンテキストスイッチが頻繁に発生する場合、TLB更新回数が大幅に少なくなり、性能が向上することを示します。