第2章 プロセスとスレッド 演習問題の解答と解説

1. 図2-2では、3つのプロセスの状態が示されています。理論上、3つの状態があれば、各状態から2つずつ、合計6つの遷移が考えられます。しかし、図には4つの遷移しか示されていません。示されていない遷移の一方または両方が発生するような状況はありますか。

- **[解答訳]** ブロック状態から実行状態への遷移は考えられます。あるプロセスがI/Oでブロックされており、そのI/Oが完了したとします。もしCPUがアイドル状態であれば、そのプロセスは準備完了状態を経ずに即座に実行状態へ移行する可能性があります。もう一方のありえない遷移、つまり準備完了状態からブロック状態への遷移は不可能です。準備完了状態のプロセスは、実行状態になるまで何もできません。I/Oのようなブロックを引き起こす可能性のある活動を行うためには、プロセスは実行されていなければなりません。

- **[より詳しい解説]** この問題は、教科書図2-2に示されているプロセスの3状態モデル（実行中、準備完了、ブロック）の理解を問うものです。

    - **ブロック → 実行中**: この遷移は可能です。あるプロセスがI/O完了を待ってブロック状態にあるとします。I/Oデバイスが割り込みを発生させ、その時点でCPUがアイドルであれば（つまり、他のどのプロセスも実行中でなければ）、OSのスケジューラはそのプロセスを準備完了キューに入れる代わりに、直接実行状態に遷移させることが最も効率的です。
    - **準備完了 → ブロック**: この遷移は不可能です。プロセスがブロック状態に陥るのは、I/Oの開始やセマフォの待機など、自らが何らかのシステムコールを実行した結果です。準備完了状態のプロセスは、定義上、CPUが割り当てられていないため、そのようなシステムコールを実行することはできません。したがって、準備完了状態から直接ブロック状態になることはありえません。

2. 割り込みではなく、ハードウェアでプロセスの切り替えを行う高度なコンピュータアーキテクチャを設計するとします。CPUはどのような情報を必要としますか。ハードウェアによるプロセス切り替えがどのように機能するか説明しなさい。

- **[解答訳]** レジスタに現在のプロセステーブルエントリへのポインタを保持することができます。I/Oが完了すると、CPUは現在のマシンの状態を現在のプロセステーブルエントリに保存します。その後、割り込みデバイスに対応する割り込みベクタへ行き、そこから別のプロセステーブルエントリ（サービスプロシージャ）へのポインタを取得します。そして、このプロセスが起動されます。

- **[より詳しい解説]** この問題は、OSの基本的な機能であるコンテキストスイッチをハードウェアで実装するアイデアを考察させるものです。ハードウェアによるプロセス切り替えを実現するためには、CPUは以下の情報を必要とします。

    - **現在のプロセステーブルエントリへのポインタ**: CPU内に専用レジスタを設け、現在実行中のプロセスのプロセステーブルエントリのアドレスを保持します。
    - **切り替え先のプロセステーブルエントリへのポインタ**: I/O完了などのイベントが発生した際に、次に実行すべきプロセス（通常は割り込みハンドラやデバイスドライバ）のプロセステーブルエントリのアドレスを知る必要があります。これは割り込みベクタのような仕組みで提供できます。 ハードウェアによるプロセス切り替えの動作は以下のようになります。

1. I/O完了などのイベントが発生します。
2. ハードウェアは、専用レジスタが指す現在のプロセステーブルエントリに、プログラムカウンタや汎用レジスタなど、現在のCPUの状態を自動的に保存します。
3. 次に、イベントに対応する割り込みベクタから、切り替え先のプロセスのプロセステーブルエントリのアドレスを読み出します。
4. そのアドレスから新しいプロセスの状態をCPUレジスタにロードし、実行を開始します。 このような機構は、ソフトウェアによるコンテキストスイッチよりも高速になる可能性がありますが、ハードウェアが複雑化し、柔軟性が失われるというトレードオフがあります。

3. 現在のすべてのコンピュータにおいて、割り込みハンドラは少なくともその一部がアセンブリ言語で書かれています。それはなぜですか。

- **[解答訳]** 一般的に、高級言語ではCPUハードウェアへの必要なアクセスができません。例えば、割り込みハンドラは、特定のデバイスへの割り込みサービスを有効化・無効化したり、プロセスのスタック領域内のデータを操作したりする必要があるかもしれません。また、割り込みサービスルーチンは、可能な限り高速に実行される必要があります。

- **[より詳しい解説]** 割り込みハンドラの最も低レベルな部分は、アセンブリ言語で書く必要があります。理由は以下の通りです。

    1. **レジスタの直接操作**: 割り込み発生直後、ハードウェアは一部のレジスタ（プログラムカウンタ、PSWなど）をスタックに退避しますが、残りの汎用レジスタはソフトウェアが保存しなければなりません。C言語などの高級言語には、すべてのCPUレジスタを直接読み書きする標準的な方法がありません。
    2. **スタックポインタの操作**: カーネル用のスタックに切り替えるため、スタックポインタを直接変更する必要がありますが、これも高級言語では困難です。
    3. **MMUやTLBの制御**: コンテキストをセットアップするために、メモリ管理ユニット（MMU）やTLB（Translation Lookaside Buffer）を操作する命令が必要になる場合がありますが、これらも特権命令であり、アセンブリ言語での記述が必要です。
    4. **性能**: 割り込み処理は可能な限り高速でなければなりません。アセンブリ言語を使うことで、きめ細かな最適化が可能になり、オーバーヘッドを最小限に抑えることができます。

4. 割り込みやシステムコールがOSに制御を移す際、割り込まれたプロセスのスタックとは別に、カーネル用のスタック領域が一般的に使用されます。それはなぜですか。

- **[解答訳]** カーネル用に別のスタックを使用する理由はいくつかあります。主な2つは次の通りです。第一に、出来の悪いユーザープログラムが十分なスタック領域を確保していないためにオペレーティングシステムがクラッシュするのを防ぎたいからです。第二に、カーネルがシステムコールからのリターン時にユーザープログラムのメモリ空間にスタックデータを残した場合、悪意のあるユーザーがこのデータを使って他のプロセスに関する情報を見つけ出すかもしれないからです。

- **[より詳しい解説]** ユーザーモードとカーネルモードでスタックを分離することは、OSの堅牢性とセキュリティにとって極めて重要です。

    1. **信頼性**: ユーザープロセスのスタックポインタが不正な値を指しているかもしれません（例えば、NULLポインタや書き込み禁止領域）。もしカーネルがそのままユーザースタックを使おうとすると、ページフォールトや保護違反を引き起こし、カーネル自身がクラッシュする可能性があります。カーネルは自身の管理下にある、常に有効なカーネルスタックを使うことで、この問題を回避します。
    2. **セキュリティ**: カーネルが処理を行う際に、カーネル内の他の関数を呼び出したり、一時的なデータをスタックに置いたりします。もしこれをユーザースタック上で行うと、カーネルがユーザーモードに戻った後、その情報（例：他のプロセスのデータ、カーネル内部のアドレス）がユーザープロセスから読み取られてしまう危険性があります。これは深刻な情報漏洩につながります。
    3. **再帰的なトラップの処理**: カーネルの実行中にページフォールトが発生するなど、トラップが入れ子になる可能性があります。ユーザースタックがページアウトされている場合、スタックへのアクセスでページフォールトが発生し、そのページフォールトハンドラがさらにスタックを使おうとすると、無限ループに陥る危険があります。カーネルスタックは常に物理メモリ上に確保（ピニング）することで、この問題を回避します。

5. あるコンピュータシステムには、主記憶に5つのプログラムを保持するのに十分な領域があります。これらのプログラムは、半分の時間はI/O待ちでアイドル状態です。CPU時間の何割が無駄になりますか。

- **[解答訳]** 5つのプロセスがすべてアイドル状態である確率は1/32なので、CPUのアイドル時間は1/32です。

- **[より詳しい解説]** この問題は、教科書2.1.7節で説明されているマルチプログラミングのCPU利用率モデル 1 -

p^n を使って解くことができます。

- nはプロセスの数で、ここでは5です。
- pはプロセスがI/O待ちである確率で、ここでは1/2 (50%) です。 CPUがアイドルになるのは、メモリ上にあるすべてのプロセスが同時にI/O待ち状態にある場合です。各プロセスが独立にI/O待ちになる確率はpなので、5つのプロセスがすべてI/O待ちになる確率は p^n = (1/2)^5 = 1/32 です。 したがって、CPUが無駄になる（アイドルになる）時間の割合は **1/32** となります。

6. あるコンピュータには4GBのRAMがあり、そのうちOSが512MBを占有しています。プロセスはすべて256MB（簡単のため）で、同じ特性を持つとします。99%のCPU利用率を目標とする場合、許容できる最大のI/O待ちの割合はどれくらいですか。

- **[解答訳]** メモリには14個のプロセスを格納するのに十分な領域があります。あるプロセスのI/O待ちの割合がpである場合、それらすべてがI/Oを待っている確率はp^14です。これを0.01に等しいとおくと、p^14 = 0.01という方程式が得られます。これを解くとp = 0.72となり、最大で72%のI/O待ちを持つプロセスまで許容できることになります。

- **[より詳しい解説]** これもCPU利用率モデル CPU利用率 = 1 - p^n を使います。

  1. **プロセス数 n の計算**:
     - ユーザーが利用可能なメモリ: 4 GB - 512 MB = 4096 MB - 512 MB = 3584 MB
     - 1プロセスあたりのメモリ: 256 MB
     - 同時にメモリに置けるプロセス数 n: 3584 MB / 256 MB = 14
  2. **I/O待ちの割合 p の計算**:
     - 目標CPU利用率: 0.99
     - CPUアイドル率: 1 - 0.99 = 0.01
     - CPUアイドル率は p^n なので、p^14 = 0.01 となります。
     - 両辺の対数をとるなどして p を求めると、p = (0.01)^(1/14) ≈ 0.72 となります。 したがって、許容できる最大のI/O待ちの割合は約**72%**です。

7. 2つのジョブがあり、それぞれ20分間のCPU時間を必要とするとします。これらが同時に開始された場合、順次実行すると最後のジョブが完了するまでにどれくらいの時間がかかりますか。並列実行した場合はどうですか。50%のI/O待ちを仮定しなさい。

- **[解答訳]** 各ジョブが50%のI/O待ちを持つ場合、競合がない状態で完了するには40分かかります。もし順次実行されるなら、2番目のジョブは最初のジョブが開始してから80分後に完了します。2つのジョブがある場合、おおよそのCPU利用率は 1 - 0.5^2 です。したがって、各ジョブは実時間の1分あたり0.375 CPU分を取得します。20分のCPU時間を蓄積するには、ジョブは20/0.375分、つまり約53.33分実行される必要があります。したがって、順次実行するとジョブは80分後に完了しますが、並列実行すると53.33分後に完了します。

- **[より詳しい解説]**

  - **順次実行の場合**:
    - 各ジョブは20分間のCPU時間を必要とし、I/O待ちの割合が50%です。これは、実行時間の50%がCPU処理、50%がI/O待ちであることを意味します。したがって、1つのジョブが完了するのに必要な実時間は 20分 / 0.5 = 40分 です。
    - 2つのジョブを順次実行すると、最初のジョブが40分、次のジョブが40分かかり、合計で **80分** 後に最後のジョブが完了します。
  - **並列実行の場合**:
    - 2つのプロセスがメモリにある場合(n=2)、CPU利用率は 1 - p^n = 1 - 0.5^2 = 1 - 0.25 = 0.75 となります。
    - これは、CPUが75%の時間稼働していることを意味します。2つのプロセスがCPUを公平に分け合うと仮定すると、各プロセスはCPU時間の 0.75 / 2 = 0.375 を得ることになります。

    - 各ジョブは20分間のCPU時間を必要とするため、完了までに必要な実時間は 20分 / 0.375 = 53.33分 となります。 並列実行することで、I/O待ちの時間を他のジョブのCPU処理に充てることができ、全体の完了時間が短縮されることがわかります。

8. 多重プログラミング度が6（つまり、同時に6つのプログラムがメモリにある）のシステムを考えます。各プロセスがその時間の40%をI/O待ちに費やすと仮定します。CPU利用率はどうなりますか。

- **[解答訳]** すべてのプロセスがI/Oを待っている確率は 0.4^6 で、これは0.004096です。したがって、CPU利用率は 1 - 0.004096 = 0.995904 となります。
- **[より詳しい解説]** ここでも CPU利用率 = 1 - p^n の公式を使います。
    - 多重プログラミング度 n = 6
    - I/O待ちの割合 p = 0.40 (40%) CPUアイドル率（すべてのプロセスがI/O待ちである確率）は p^n = 0.4^6 = 0.004096 です。 したがって、CPU利用率は 1 - 0.004096 = 0.995904、つまり約**99.6%**となります。

9. インターネットから2GBの大きなファイルをダウンロードしようとしているとします。そのファイルは複数のミラーサーバーから利用可能で、各サーバーはファイルの一部を提供できます。リクエストではファイルの開始バイトと終了バイトを指定すると仮定します。ダウンロード時間を改善するためにスレッドをどのように利用できるか説明しなさい。

- **[解答訳]** クライアントプロセスは、別々のスレッドを作成することができます。各スレッドは、ミラーサーバーの1つからファイルの異なる部分を取得します。これにより、ダウンタイムを短縮できます。もちろん、すべてのスレッドが共有する単一のネットワークリンクが存在します。このリンクは、スレッド数が非常に多くなるとボトルネックになる可能性があります。
- **[より詳しい解説]** この問題は、スレッドの並列性を利用した性能向上の典型例です。
    1. **タスクの分割**: 2GBのファイルを複数のチャンク（区間）に分割します。例えば、10個のミラーサーバーがあるなら、それぞれに200MBずつ割り当てることができます。
    2. **スレッドの生成**: 各チャンクをダウンロードするために、クライアントアプリケーション内で別々のスレッドを生成します。
    3. **並列ダウンロード**: 各スレッドは、割り当てられたチャンクを、異なるミラーサーバーから同時にダウンロードします。
    4. **ファイルの結合**: すべてのスレッドがダウンロードを完了したら、ダウンロードしたチャンクを結合して元の2GBのファイルを復元します。 この方法の利点は、単一のサーバーからシーケンシャルにダウンロードするのではなく、複数のサーバーの帯域幅を同時に利用できるため、全体のダウンロード時間を大幅に短縮できることです。ただし、解答が指摘するように、クライアント側のネットワーク接続（例：家庭のインターネット回線）の帯域幅が、全サーバーからの合計データレートよりも狭い場合は、そこがボトルネックとなり、スレッドを増やしても効果は頭打ちになります。

10. 本文では、図2-11(a)のモデルはメモリ内にキャッシュを使用するファイルサーバーには適していないと述べました。それはなぜですか。各プロセスが独自のキャッシュを持つことは可能ですか。

- **[解答訳]** ファイルシステムの一貫性を保つことは困難、あるいは不可能でしょう。クライアントプロセスがサーバープロセス1にファイルを更新するリクエストを送ったとします。このプロセスはメモリ内のキャッシュエントリを更新します。その後すぐに、別のクライアントプロセスがサーバー2にそのファイルを読み込むリクエストを送ったとします。残念ながら、もしそのファイルがそこでもキャッシュされている場合、サーバー2は無邪気に古いデータを返してしまいます。もし最初のプロセスがキャッシュした後にディスクに書き込み、サーバー2が読み込むたびにディスクをチェックしてキャッシュされたコピーが最新か確認するなら、システムは動作可能ですが、これではキャッシュシステムが避けようとしているディスクアクセスがすべて発生してしまいます。
- **[より詳しい解説]** この問題は、共有リソース（この場合はファイルキャッシュ）へのアクセスにおける一貫性

の問題、つまり競合状態を考察させるものです。 図2-11(a)は、それぞれが独立したアドレス空間を持つ複数のプロセスを示しています。もし、各サーバープロセスが独自のファイルキャッシュを自身のアドレス空間内に持つと、以下のような問題が発生します。

- **キャッシュの一貫性問題**: プロセスAがファイルFを更新し、その内容を自身のキャッシュに反映させたとします。その後、プロセスBがファイルFを読み込もうとした場合、プロセスBのキャッシュには古い内容が残っているかもしれません。ディスクから読み直さない限り、プロセスBは古いデータを取得してしまい、データの一貫性が失われます。
- **ディスクアクセスの増加**: この一貫性問題を解決するには、プロセスAがキャッシュを更新するたびにディスクに書き戻し（ライトスルーキャッシュ）、プロセスBが読み込む前に必ずディスクの内容を確認する必要があります。しかし、これではキャッシュによる性能向上のメリットが失われてしまいます。 各プロセスが独自のキャッシュを持つこと自体は可能ですが、それらの一貫性を保つための同期メカニズム（例えば、キャッシュコヒーレンシプロトコル）が必要になり、実装が非常に複雑になります。一方、図2-11(b)のように、複数のスレッドが単一のプロセス内で動作する場合、すべてのスレッドが同じアドレス空間、したがって同じキャッシュを共有するため、この問題は発生しません（ただし、キャッシュ内のデータ構造への同時アクセスを防ぐための同期は別途必要です）。

11. マルチスレッドのプロセスがforkした場合、子プロセスが親のすべてのスレッドのコピーを取得すると問題が発生します。元のスレッドの1つがキーボード入力を待ってブロックしていたとします。今や、各プロセスに1つずつ、2つのスレッドがキーボード入力を待っていることになります。この問題はシングルスレッドのプロセスでも発生しますか。

- **[解答訳]** いいえ。もしシングルスレッドのプロセスがキーボードでブロックされている場合、それはforkできません。

- **[より詳しい解説]** この問題は、マルチスレッド環境におけるforkの複雑さを指摘するものです。

  - **マルチスレッドプロセスの場合**: 親プロセスに複数のスレッドが存在し、そのうちの1つ（スレッドA）がreadシステムコールでキーボード入力を待ってブロックしているとします。このとき、別のスレッド（スレッドB）がforkを呼び出すと、子プロセスが生成されます。もし子プロセスが親のすべてのスレッドを複製する仕様だと、子プロセス内にもキーボード入力を待つスレッドA’が生まれます。結果として、親プロセスのスレッドAと子プロセスのスレッドA’の両方がキーボード入力を待つことになり、次に入力された行がどちらに渡されるべきかという曖昧さが生じます。
  - **シングルスレッドプロセスの場合**: プロセスにはスレッドが1つしかありません。もしそのスレッドがキーボード入力でブロックされている場合、そのプロセスはブロック状態にあり、CPUが割り当てられていないため、forkシステムコールを実行すること自体ができません。forkを呼び出すためには、プロセスが実行状態でなければなりません。したがって、この問題はシングルスレッドのプロセスでは発生しません。

12. 図2-8には、マルチスレッドのWebサーバーが示されています。ファイルから読み取る唯一の方法が通常のブロッキングreadシステムコールである場合、そのWebサーバーにはユーザーレベルスレッドとカーネルレベルスレッドのどちらが使われていると考えられますか。その理由も述べなさい。

- **[解答訳]** ワーカースレッドは、ディスクからWebページを読み込む必要があるときにブロックします。もしユーザーレベルスレッドが使われている場合、このアクションはプロセス全体をブロックさせ、マルチスレッドの価値を破壊してしまいます。したがって、一部のスレッドがブロックしても他のスレッドに影響を与えないように、カーネルスレッドが使われることが不可欠です。

- **[より詳しい解説]** この問題は、ユーザーレベルスレッドとカーネルレベルスレッドの決定的な違い、すなわち**ブロッキングシステムコールの扱い**を問うています。

  - **ユーザーレベルスレッドの場合**: カーネルはスレッドの存在を認識していません。カーネルが見ているのは、単一のスレッドを持つプロセスだけです。したがって、プロセス内のいずれかのユーザーレベルスレッドがブロッキングreadシステムコールを発行すると、カーネルはそのプロセス全体をブロック状態に

します。これにより、他のすべてのユーザーレベルスレッドも実行できなくなり、スレッド化による並行処理のメリットが失われます。
    - **カーネルレベルスレッドの場合**: カーネルが各スレッドを個別の実行単位として認識し、スケジューリングします。したがって、あるカーネルレベルスレッドがreadでブロックされても、カーネルは同じプロセス内の他の準備完了状態にあるカーネルレベルスレッドをスケジュールして実行させることができます。Webサーバーのように、I/O処理で頻繁にブロックが発生するアプリケーションでは、I/Oの待ち時間を他のリクエストの処理でオーバーラップさせることが性能向上の鍵です。したがって、**カーネルレベルスレッド**が使われていると考えるのが妥当です。

13. 本文では、マルチスレッドのWebサーバーについて説明し、それがシングルスレッドのサーバーや有限状態機械のサーバーより優れている理由を示しました。シングルスレッドのサーバーの方が優れている状況はありますか。例を挙げなさい。

- **[解答訳]** はい。もしサーバーが完全にCPUバウンドであるなら、複数のスレッドを持つ必要はありません。それは不必要な複雑さを加えるだけです。例えば、100万人のエリアの電話帳案内番号（555-1212のような）を考えます。もし各（名前、電話番号）のレコードが、例えば64文字であるなら、データベース全体は64メガバイトを占め、高速なルックアップを提供するためにサーバーのメモリに簡単に保持できます。

- **[より詳しい解説]** スレッド化の主な利点は、CPU処理とI/O処理をオーバーラップさせることによる性能向上です。したがって、この利点が得られない状況では、シングルスレッドの方がシンプルで優れている可能性があります。

    - **CPUバウンドなタスク**: 解答の例のように、すべてのデータがメモリ上にあり、ディスクI/Oが全く発生しないサーバーでは、処理は完全にCPUバウンドになります。この場合、リクエストを処理中にブロックすることがないため、マルチスレッド化しても性能は向上しません。むしろ、スレッド間の同期やコンテキストスイッチのためのオーバーヘッドが生じるため、性能がわずかに低下する可能性があります。
    - **シンプルな逐次処理**: 非常に単純で、各リクエストが極めて短時間で完了するようなサーバーでは、シングルスレッドでリクエストを順番に処理しても十分な性能が得られる場合があります。この場合も、マルチスレッド化による複雑さを導入するメリットはありません。

14. 図2-12では、レジスタセットはプロセスごとの項目ではなく、スレッドごとの項目としてリストされています。それはなぜですか。マシンにはレジスタセットが1つしかないのに。

- **[解答訳]** スレッドが停止すると、そのレジスタには値が入っています。それらは保存されなければなりません、プロセスが停止したときにレジスタを保存する必要があるのと同様です。マルチプログラミングスレッドは、マルチプログラミングプロセスと何ら変わりません。したがって、各スレッドは自身のレジスタ保存領域を必要とします。

- **[より詳しい解説]** この問題は、スレッドが独立した実行の単位であることの含意を問うています。 物理的には、CPUのレジスタセットは1つしかありません。しかし、OSが複数のスレッドを（擬似）並列に実行するとき、あるスレッドから別のスレッドへCPUを切り替える（コンテキストスイッチ）必要があります。このとき、中断されるスレッドが計算途中で保持していたレジスタの値（プログラムカウンタ、スタックポインタ、汎用レジスタなど）をどこかに保存しなければ、次にそのスレッドが実行を再開したときに正しく処理を続けられません。 したがって、**各スレッドは、自身のレジスタの値を保存するための論理的な保存領域を必要とします**。この保存領域が、図2-12で「スレッドごとの項目」としてリストされている「レジスタセット」です。これは、スレッドがアクティブでないときにその状態を保持する、スレッド制御ブロックの一部です。

15. なぜスレッドは`thread_yield`を呼び出して自発的にCPUを明け渡すことがあるのでしょうか。定期的なクロック割り込みがないため、CPUを二度と取り戻せないかもしれないのに。

- **[解答訳]** 一つのプロセス内のスレッドは互いに協力します。それらは互いに敵対的ではありません。もしyielding（CPUを譲ること）がアプリケーションのためになるのであれば、スレッドはyieldします。結局のところ、通常は同じプログラマがすべてのスレッドのコードを書いているのです。

- **[より詳しい解説]** この質問は、特にユーザーレベルスレッドの実装における協調的スケジューリングの概念に関連しています。

    - **協調動作**: プロセス内のスレッドは、共通の目標（アプリケーションのタスク）を達成するために協力して動作するように設計されています。あるスレッドが重要な処理を終え、他のスレッドがすぐに実行されるべきであるとプログラマが判断した場合、thread_yieldを呼び出して能動的に他のスレッドにCPUを譲ることができます。
    - **デッドラインや応答性**: あるスレッドが長時間CPUを占有すると、他のスレッド（例えば、ユーザーインターフェースを処理するスレッド）の応答性が悪化する可能性があります。thread_yieldを適切に呼び出すことで、アプリケーション全体の応答性を保つことができます。
    - **クロック割り込みの不在**: ユーザーレベルスレッドでは、カーネルからのクロック割り込みによる強制的なプリエンプション（横取り）がありません。そのため、あるスレッドがCPUを独占し続けると、他のスレッドは全く実行できなくなります。thread_yieldは、この状況を避けるためのプログラマによる協調的なメカニズムです。「CPUを二度と取り戻せないかもしれない」という懸念は理論的にはありますが、協力的に設計されたスレッドパッケージでは、ランタイムシステムが公平なスケジューリングを保証するため、実際には問題になりません。

16. スレッドがクロック割り込みによってプリエンプト（横取り）されることはありますか。もしあるなら、どのような状況ですか。もしないなら、なぜですか。

- **[解答訳]** ユーザーレベルスレッドは、プロセス全体のクォンタムが使い果たされない限り、クロックによってプリエンプトされることはありません（ただし、透過的なクロック割り込みは発生する可能性があります）。カーネルレベルスレッドは個別にプリエンプトされることがあります。後者の場合、もしスレッドが実行時間が長すぎると、クロックは現在のプロセス、したがって現在のスレッドを割り込みます。カーネルは、もしそう望むなら、同じプロセスから別のスレッドを次に実行するように自由に選択できます。

- **[より詳しい解説]** この問題の答えは、スレッドがユーザーレベルで実装されているか、カーネルレベルで実装されているかによって異なります。

    - **ユーザーレベルスレッド**: カーネルはスレッドの存在を知らず、プロセス単位でスケジューリングします。クロック割り込みが発生すると、カーネルは現在実行中のプロセスの残りクォンタムを確認します。もしクォンタムが残っていれば、同じプロセス（したがって、同じユーザーレベルスレッド）の実行を再開します。もしクォンタムが切れていれば、カーネルはプロセスを切り替えます。したがって、**個々のユーザーレベルスレッドがクロック割り込みで直接プリエンプトされることはありません**。プリエンプションはプロセス単位で行われます。
    - **カーネルレベルスレッド**: カーネルが各スレッドを個別のスケジューリング単位として管理します。クロック割り込みが発生すると、カーネルは現在実行中のスレッドのクォンタムが切れたかどうかを確認します。もし切れていれば、カーネルはスケジューリングを行い、別のスレッド（同じプロセスの別のスレッド、または別のプロセスのスレッド）にCPUを割り当てることができます。したがって、**カーネルレベルスレッドはクロック割り込みによってプリエンプトされます**。

17. シングルスレッドのファイルサーバーとマルチスレッドのサーバーを使ってファイルを読み取る場合を比較します。データの要求を受け取り、それをディスパッチし、必要な処理を行うのに12ミリ秒かかります（必要なデータがブロックキャッシュにある場合）。ディスク操作が必要な場合（1/3の確率で発生）、追加で75ミリ秒が必要で、その間スレッドはスリープします。サーバーがシングルスレッドの場合、1秒あたり何件のリクエストを処理できますか。マルチスレッドの場合はどうですか。

- **[解答訳]** シングルスレッドの場合、キャッシュヒットには12ミリ秒、キャッシュミスには87ミリ秒かかります。加重平均は 2/3 × 12 + 1/3 × 87 です。したがって、平均リクエストには37ミリ秒かかり、サーバーは1秒あたり約27件のリクエストを処理できます。マルチスレッドサーバーの場合、ディスク待ち時間はすべてオーバーラップされるため、すべてのリクエストは12ミリ秒かかり、サーバーは1秒あたり83と1/3件のリクエストを処理できます。

- **[より詳しい解説]** この問題は、I/Oバウンドなタスクにおけるマルチスレッドの性能向上効果を定量的に示すものです。
    - **シングルスレッドサーバー**:
        - リクエストは1つずつ順番に処理されます。
        - キャッシュヒットの場合の処理時間: 12 ms
        - キャッシュミスの場合の処理時間: 12 ms (ディスパッチ) + 75 ms (ディスクI/O) = 87 ms
        - キャッシュミスの確率は1/3、ヒットの確率は2/3です。
        - 平均処理時間: (2/3) * 12 ms + (1/3) * 87 ms = 8 ms + 29 ms = 37 ms
        - 1秒あたりの処理可能リクエスト数: 1000 ms / 37 ms ≈ 27.03 件
    - **マルチスレッドサーバー**:
        - ディスクI/Oでブロックするスレッドがあっても、他のスレッドがCPUを使って別のリクエストを処理できます。つまり、**75msのディスク待ち時間を他のリクエストの処理で完全に隠蔽（オーバーラップ）できる**と仮定します。
        - したがって、サーバーの性能はディスクアクセスではなく、CPU処理時間によって決まります。
        - 各リクエストのCPU処理時間は12msです。
        - 1秒あたりの処理可能リクエスト数: 1000 ms / 12 ms ≈ 83.33 件 マルチスレッド化により、サーバーの処理能力は約3倍に向上することがわかります。

18. ユーザースペースでスレッドを実装する最大の利点は何ですか。最大の欠点は何ですか。

- **[解答訳]** 最大の利点は効率です。スレッドを切り替えるのにカーネルへのトラップは必要ありません。最大の欠点は、もし1つのスレッドがブロックすると、プロセス全体がブロックしてしまうことです。

- **[より詳しい解説]** 教科書2.2.4節と2.2.5節で詳述されている内容です。
    - **最大の利点: 効率性・パフォーマンス**
        - **高速なコンテキストスイッチ**: スレッドの生成、終了、切り替えがすべてユーザー空間のライブラリ関数として実行されるため、モードスイッチ（ユーザーモード↔カーネルモード）のオーバーヘッドが発生しません。これはカーネルコールよりも桁違いに高速です。
        - **独自のスケジューリング**: 各プロセスが自身に最適化されたスケジューリングアルゴリズムを持つことができます。
    - **最大の欠点: プロセス全体のブロッキング**
        - **ブロッキングシステムコール**: あるユーザーレベルスレッドがブロッキングI/O（例：read）を発行すると、カーネルはプロセス全体をブロックさせます。これにより、他の実行可能なスレッドも停止してしまいます。
        - **ページフォールト**: 同様に、あるスレッドがページフォールトを起こすと、カーネルはプロセス全体をブロックさせ、他のスレッドの実行が妨げられます。

19. 図2-15では、スレッドの生成とスレッドによって出力されるメッセージがランダムな順序で入り混じっています。これを「スレッド1生成、スレッド1メッセージ出力、スレッド1終了、スレッド2生成、スレッド2メッセージ出力、スレッド2終了」という厳密な順序に強制する方法はありますか。もしあるなら、どうすればよいですか。もしないなら、なぜですか。

- **[解答訳]** はい、できます。各pthread_create呼び出しの後に、メインプログラムはpthread_joinを呼び出して、たった今作成したスレッドが終了するのを待ってから、次のスレッドを作成することができます。

- **[より詳しい解説]** 図2-15のプログラムでは、メインスレッドがループ内で次々とスレッドを生成し、すぐに次のループに進みます。生成されたスレッドとメインスレッドは並行して実行されるため、どのスレッドがいつ実行されるかはスケジューラに依存し、実行順序は非決定的になります。 この順序を強制するには、**同期**が必要です。解答にあるように、pthread_join関数がその役割を果たします。pthread_joinは、指定したスレッドが終了するまで呼び出し元のスレッドをブロックさせる関数です。 修正後のメインプログラムのループは以下のようになります。 c　　for(i=0; i < NUMBER_OF_THREADS; i++) {　　printf("Main here. Creating

```
thread %d\n", i);         status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
if (status != 0) { /* エラー処理 */ }         pthread_join(threads[i], NULL); // この行を追加     }
```

このように変更すると、メインスレッドはスレッドiを生成した後、そのスレッドiがpthread_exitで終了するまでpthread_joinで待機します。そして初めて次のループ（スレッドi+1の生成）に進むため、厳密な順序が保証されます。ただし、これではスレッドを並列実行する意味がなくなってしまいます。

20. スレッドにおけるグローバル変数の議論で、変数自身ではなく、変数へのポインタを格納するためにcreate_globalというプロシージャを使用しました。これは不可欠ですか、それともプロシージャは値そのものでもうまく機能しますか。

- **[解答訳]** ポインタは本当に必要です。なぜなら、グローバル変数のサイズが不明だからです。それは文字かもしれないし、浮動小数点数の配列かもしれません。もし値が格納されるなら、create_globalにサイズを渡さなければならないでしょうが、それは問題ありません。しかし、set_globalの第2引数の型、そしてread_globalの値の型はどうなるでしょうか？

- **[より詳しい解説]** この問題は、型安全性の観点からスレッドごとのグローバル変数の実装を考察させるものです。 C言語のような静的型付け言語では、関数は特定の型の引数を受け取り、特定の型の値を返します。もしread_globalが「値そのもの」を返すとしたら、その関数の戻り値の型は何であるべきでしょうか？ あるスレッドはintを、別のスレッドはdoubleを、また別のスレッドは構造体を格納したいかもしれません。これらすべての型を扱える単一のread_global関数を（型安全な方法で）実装することは困難です。 一方、**ポインタ（void *）** を使うことで、この問題を回避できます。

    - set_globalは、格納したいデータの型に関わらず、そのデータへのポインタ（void *）を引数として受け取ることができます。
    - read_globalは、格納されているポインタ（void *）を返すことができます。 呼び出し元のスレッドは、自分がどのような型のデータを格納したかを知っているので、read_globalから返されたvoid *ポインタを正しい型にキャストして使用することができます。このように、ポインタを使うことで、任意の型のデータを汎用的に扱うことが可能になります。

21. スレッドが完全にユーザースペースで実装され、ランタイムシステムが1秒に1回クロック割り込みを受け取るシステムを考えます。あるスレッドがランタイムシステム内で実行中にクロック割り込みが発生した場合、どのような問題が起こり得ますか。それを解決する方法を提案できますか。

- **[解答訳]** ランタイムシステムがまさにスレッドをブロックまたはアンブロックしようとしていて、スケジューリングキューを操作している最中に、クロック割り込みハンドラがスレッド切り替えの時期かどうかを確認するためにそれらのキューを検査し始めると、問題が発生する可能性があります。なぜなら、キューが矛盾した状態にあるかもしれないからです。一つの解決策は、ランタイムシステムに入るときにフラグを立てることです。クロックハンドラはこのフラグを見て、自身のフラグを立ててからリターンします。ランタイムシステムが終了するときにクロックフラグをチェックし、クロック割り込みが発生したことを見て、クロックハンドラを実行します。

- **[より詳しい解説]** この問題は、ユーザーレベルのスレッドランタイムシステムと、それをプリエンプトする可能性がある（しかし協調的ではない）クロック割り込みとの間の競合状態を指摘しています。

    - **問題点**: ユーザーレベルスレッドのランタイムシステムは、スレッドテーブルや準備完了キューといった共有データ構造を操作します。もし、これらのデータ構造を更新している途中で（つまり、データ構造が一時的に矛盾した状態にあるときに）クロック割り込みが発生し、割り込みハンドラが同じデータ構造を読み書きしようとすると、データが破壊されたり、システムがクラッシュしたりする可能性があります。これは典型的な**競合状態**です。
    - **解決策**: 解答で提案されている方法は、一種の**割り込みの遅延実行**です。
        1. ランタイムシステムは、自身のクリティカルリージョンに入る前に、グローバルな「ランタイム実行中」フラグを立てます。
        2. クロック割り込みハンドラは、まずこのフラグをチェックします。もしフラグが立っていれば、

キューの操作は行わず、「割り込み発生済み」フラグを立ててすぐにリターンします。
3. ランタイムシステムは、クリティカルリージョンを抜けた後、「ランタイム実行中」フラグをクリアする前に、「割り込み発生済み」フラグをチェックします。もし立っていれば、保留されていたクロック割り込み処理（スケジューリングなど）を実行し、「割り込み発生済み」フラグをクリアします。 この方法により、ランタイムシステムのクリティカルな処理が割り込みによって中断されるのを防ぎ、安全性を確保できます。

22. あるOSには、ファイル、パイプ、またはデバイスから安全に読み取れるかを事前に確認するためのselectシステムコールのようなものがないとします。しかし、ブロックされたシステムコールを中断させるアラームクロックを設定することはできます。これらの条件下で、ユーザースペースにスレッドパッケージを実装することは可能ですか。議論しなさい。

- **[解答訳]** はい、可能ですが非効率的です。スレッドがシステムコールを行いたいときは、まずアラームタイマーをセットしてから、コールを実行します。もしコールがブロックすると、タイマーがスレッドパッケージに制御を戻します。もちろん、ほとんどの場合、コールはブロックしないので、タイマーをクリアしなければなりません。したがって、ブロックする可能性のあるシステムコールは、それぞれ3つのシステムコールとして実行されることになります。もしタイマーが早すぎると、様々な問題が発生します。これはスレッドパッケージを構築する上で魅力的な方法ではありません。

- **[より詳しい解説]** この問題は、selectのようなノンブロッキングI/Oの確認手段がない場合に、ユーザーレベルスレッドパッケージがブロッキングシステムコールの問題をどう回避できるかを考察させるものです。 解答のアイデアは、ブロッキングシステムコールを**タイムアウト付き**で呼び出すことで、擬似的なノンブロッキング動作を実現しようとするものです。

  1. **アラーム設定**: スレッドがreadのようなブロッキングする可能性のあるシステムコールを呼び出す直前に、ライブラリのラッパー関数がalarmシステムコールで短いタイマーを設定します。
  2. **システムコール実行**: readを呼び出します。
     - **成功した場合**: readはデータを受け取り、すぐにリターンします。ラッパー関数は設定したアラームをキャンセルし、ユーザーコードに制御を戻します。
     - **ブロックした場合**: readでブロックされます。しかし、設定したタイマーが時間切れになると、alarmがシグナルを発生させ、readシステムコールは中断されてエラー（EINTR）でリターンします。シグナルハンドラは、スレッドライブラリに制御を戻します。
  3. **スレッド切り替え**: スレッドライブラリは、readがブロックしたと判断し、別の準備完了スレッドに切り替えます。 この方法は理論的には可能ですが、解答が指摘するように、**非常に非効率**です。ブロックしない場合でもalarmとcancelのオーバーヘッドが発生し、ブロックする場合にはシグナル処理のオーバーヘッドが加わります。また、適切なタイムアウト値の設定も難しく、実用的な実装とは言えません。

23. turn変数を使ったビジーウェイトの解決策（図2-23）は、2つのプロセスが共有メモリを持つマルチプロセッサ（つまり、共通のメモリを共有する2つのCPU）で実行された場合でも機能しますか。

- **[解答訳]** はい、動作します。ただし、もちろんビジーウェイトであることに変わりはありません。

- **[より詳しい解説]** 図2-23の厳密な交互実行アルゴリズムは、turnという単一の共有変数を読み書きすることで成り立っています。マルチプロセッサ環境でこれが機能するためには、あるCPUによるturn変数への書き込みが、他のCPUから即座に（あるいは一貫した順序で）観測可能でなければなりません。現代のキャッシュコヒーレントなマルチプロセッサでは、この条件は満たされます。

  - プロセス0がturn = 1;を実行すると、この変更は最終的に共有メモリに書き込まれ、プロセス1が動作しているCPUのキャッシュも更新されます（あるいは無効化され、再読み込みされます）。
  - これにより、プロセス1はwhile (turn != 1)のループを抜け、自身のクリティカルリージョンに入ることができます。 したがって、アルゴリズムの正当性は保たれます。しかし、このアルゴリズムが持つ「一方が非クリティカルリージョンにいても、もう一方がブロックされる」という問題点と、「CPU時間を浪費するビジーウェイト」という問題点は、マルチプロセッサ環境でも解決されません。

24. 図2-24に示されているピーターソンの相互排除問題の解決策は、プロセスのスケジューリングがプリエンプティブ（横取り有り）の場合に機能しますか。ノンプレンプティブ（横取り無し）の場合はどうですか。

- **[解答訳]** プリエンプティブスケジューリングでは確かに動作します。実際、このケースのために設計されています。ノンプレンプティブスケジューリングの場合、失敗する可能性があります。turnが最初は0で、プロセス1が最初に実行されるケースを考えます。プロセス1は無限にループし、CPUを解放することはありません。

- **[より詳しい解説]** ピーターソンのアルゴリズム（図2-24）は、ソフトウェアだけで相互排除を実現する優れた解法です。

    - **プリエンプティブスケジューリング**: このアルゴリズムは、interested配列への書き込みとturn変数への書き込みの間や、whileループの条件チェックの途中でプロセスが切り替えられても、正しく相互排除を保証するように設計されています。したがって、プリエンプティブ環境で問題なく動作します。
    - **ノンプレンプティブスケジューリング**: この環境では、プロセスは自発的にCPUを解放する（ブロックするか終了する）まで実行を続けます。解答が指摘するように、もしプロセス1が最初に実行され、interestedがFALSEである場合（初期状態）、プロセス1はenter_region内のwhile (turn == 1 && interested == TRUE)ループには入りません。しかし、もしプロセス0がinterested=TRUEとし、turn=0とした後にプロセス1がスケジュールされると、プロセス1はinterested=TRUE, turn=1とし、whileループに入ります。ここで、turn==1かつinterested==TRUEなので、プロセス1はループし続けます。ノンプレンプティブなので、プロセス0にCPUが切り替わることがなく、プロセス1は無限ループに陥ります。**訂正**: 解答のシナリオは少し違います。turn=0の初期状態でプロセス1が最初に実行されると、interested=TRUE, turn=1と設定し、whileループの条件 (turn == 1 && interested == TRUE) を評価します。interestedはFALSEなので、プロセス1はクリティカルリージョンに入ります。問題は、プロセス0がenter_regionを呼び出した後、whileループで待っている間に、プロセス1がCPUを解放しない場合です。 **解答のシナリオの再解釈**: turn=0の初期状態でプロセス1が実行を開始。interested = TRUE; turn = 1; を実行。whileループの条件はinterestedがFALSEなので偽となり、プロセス1はクリティカルリージョンへ。その後、非クリティカルリージョンへ。再びenter_regionを呼び出す。interested = TRUE; turn = 1;を実行。whileループへ。ここでプロセス0がまだenter_regionを呼び出していなければ、プロセス1は再びクリティカルリージョンへ。このシナリオでは問題ない。 **解答が意図する本当の問題**: turn=0の状態でプロセス0がenter_regionを呼び、interested = TRUE;とturn = 0;を実行。whileループの条件は偽なので、クリティカルリージョンへ。ここでノンプレンプティブなのでCPUを離さない。プロセス1は実行機会を得られず飢餓状態になる。これはアルゴリズムの失敗というより、ノンプレンプティブスケジューリングの問題そのものです。 **解答の指摘する「プロセス1が最初に実行され、ループし続ける」というシナリオ**: これが起こるのは、turnの初期値が1の場合です。turn=1、interested=FALSEの状態でプロセス1が実行開始すると、whileループの条件は偽なのでクリティカルリージョンに入れます。問題は、プロセス0が実行され、interested=TRUEとした後で、プロセス1がwhile (turn == 1 && interested == TRUE)ループに入った場合です。この場合、プロセス1はinterestedがFALSEになるのを待ち続けますが、ノンプレンプティブなのでプロセス0が実行されてleave_regionを呼び出すことはなく、無限ループになります。

25. 2.3.4節で議論された優先度逆転問題は、ユーザーレベルスレッドで発生する可能性はありますか。それはなぜですか、あるいはなぜ発生しないのですか。

- **[解答訳]** 優先度逆転問題は、低優先度のプロセスがクリティカルリージョンにいて、突然、高優先度のプロセスが準備完了状態になりスケジュールされた場合に発生します。もし高優先度のプロセスがビジーウェイトを使うと、永遠にループし続けます。ユーザーレベルスレッドでは、低優先度のスレッドが突然プリエンプトされて高優先度のスレッドが実行されることはありえません。プリエンプションは存在しません。

- **[より詳しい解説]** 優先度逆転問題は、高優先度プロセスH、中優先度プロセスM、低優先度プロセスLが存在し、Lがクリティカルリージョン（リソースRをロック）に入っているときにHが準備完了状態になることで発生します。HはLよりも優先度が高いため、Lをプリエンプトして実行を開始します。HがリソースRを要求すると、Lが保持しているためブロックされます。ここで、スケジューラは次に実行可能な最高優先度のプロセス、つまりMを実行します。結果として、HはMの実行が完了するまで待たされることになり、実質的にMよりも低

い優先度で動作してしまいます。

    - **ユーザーレベルスレッド**: ユーザーレベルスレッドでは、通常、同じプロセス内のスレッド間でのプリエンプションは存在しません。スレッドの切り替えは、スレッドが自発的に`thread_yield`を呼び出すか、ブロッキングI/O（のラッパー関数）を呼び出すかした場合にのみ発生します。したがって、低優先度のスレッドLがクリティカルリージョンを実行中に、高優先度のスレッドHが準備完了状態になったとしても、HがLを**強制的にプリエンプトすることはありません**。Lは自身の処理を続け、クリティカルリージョンを抜けることができます。したがって、教科書で説明された形の優先度逆転問題はユーザーレベルスレッドでは発生しません。

26. 2.3.4節で、高優先度プロセスHと低優先度プロセスLが無限ループに陥る状況を説明しました。優先度スケジューリングの代わりにラウンドロビンスケジューリングが使われた場合も、同じ問題が発生しますか。議論しなさい。

- **[解答訳]** ラウンドロビンスケジューリングでは動作します。遅かれ早かれLが実行され、最終的にはクリティカルリージョンを抜けます。重要なのは、優先度スケジューリングではLは全く実行されないのに対し、ラウンドロビンでは定期的に通常のタイムスライスが与えられるため、クリティカルリージョンを抜ける機会があるということです。

- **[より詳しい解説]** 問題の状況は、Lがクリティカルリージョンにいる間に、より優先度の高いHが準備完了になり、ビジーウェイトでLが保持するロックを待つ、というものです。

    - **優先度スケジューリング**: Hが準備完了である限り、Lは決してスケジュールされません。そのため、Lはクリティカルリージョンを抜けてロックを解放することができず、Hは永遠にビジーウェイトを続けることになります。
    - **ラウンドロビンスケジューリング**: この方式では、すべての準備完了プロセスが順番にタイムクォンタムを与えられます。HがビジーウェイトでCPUを消費していても、そのクォンタムが尽きれば、次にキューにいるプロセス（いずれLの番が来る）にCPUが切り替わります。Lがスケジュールされれば、クリティカルリージョン内の処理を進め、いずれロックを解放することができます。ロックが解放されれば、次にHが実行されたときにループを抜けることができます。したがって、ラウンドロビンではこの問題は発生しません。

27. スレッドを持つシステムでは、ユーザーレベルスレッドが使われる場合、スタックはスレッドごとに1つですか、それともプロセスごとに1つですか。カーネルレベルスレッドが使われる場合はどうですか。説明しなさい。

- **[解答訳]** 各スレッドは自身でプロシージャを呼び出すので、ローカル変数やリターンアドレスなどを保持するために独自のスタックが必要です。これはユーザーレベルスレッドでもカーネルレベルスレッドでも同様です。

- **[より詳しい解説]** スタックは、プロシージャ（関数）呼び出しの履歴、ローカル変数、引数、リターンアドレスなどを保持するために不可欠なデータ構造です。

    - **ユーザーレベルスレッド**: 各スレッドは独立した実行の流れを持ち、異なるプロシージャを呼び出す可能性があります。例えば、スレッドAが`foo()`を呼び出し、スレッドBが`bar()`を呼び出しているかもしれません。それぞれの実行履歴は異なるため、**各スレッドは自身専用のスタックをユーザー空間に持つ必要があります**。これらのスタックは、スレッドライブラリによって管理されます。
    - **カーネルレベルスレッド**: カーネルレベルスレッドも同様に、独立した実行の流れです。したがって、**各スレッドは自身専用のスタックを持つ必要があります**。ユーザーモードで実行中はユーザースタックを、カーネルモードで実行中（システムコールなど）はカーネルスタックを使用します。この場合、各スレッドはユーザー/カーネルの両方のスタックを持つことになります。

28. コンピュータを開発する際、通常はまず1命令ずつ実行するプログラムによってシミュレートされます。マルチプロセッサでさえ、このように厳密に逐次的にシミュレートされます。このような同時イベントがない状況で、競合状態が発生することは可能ですか。

- **[解答訳]** はい。シミュレートされるコンピュータはマルチプログラミングされているかもしれません。例えば、プロセスAが実行中に、共有変数を読み出したとします。その後、シミュレートされたクロックティックが発生し、プロセスBが実行されます。Bも同じ変数を読み出します。そして、変数に1を加えます。プロセスAが実行されるとき、もしそれも変数に1を加えるなら、競合状態が発生します。

- **[より詳しい解説]** 競合状態は、物理的に同時にイベントが発生すること（真の並列性）が原因ではなく、**複数の実行フローが共有リソースにアクセスする処理の途中で、予期せず実行が中断・再開されること（インターリーブ）** が原因で発生します。 シミュレータがマルチプログラミングOSをシミュレートする場合、以下のようになります。

  1. シミュレータはプロセスAの命令を1つずつ実行します。
  2. プロセスAが共有変数Xをレジスタに読み込みます。
  3. 次にシミュレータは「クロック割り込み」イベントをシミュレートし、OSのスケジューラを呼び出します。
  4. スケジューラはプロセスBに切り替えることを決定します。
  5. シミュレータはプロセスBの命令を実行し始めます。
  6. プロセスBも共有変数Xをレジスタに読み込み、値を更新してメモリに書き戻します。
  7. 再び「クロック割り込み」でプロセスAに切り替わります。
  8. プロセスAは、自分が以前に読み込んだ古いXの値を元に計算を続け、メモリに書き戻します。これにより、プロセスBの更新が失われます。 このように、物理的な同時実行がなくても、実行のインターリーブさえあれば競合状態は発生します。

29. 生産者消費者問題は、1つの共有バッファに書き込む（または読み取る）複数の生産者と消費者がいるシステムに拡張できます。各生産者と消費者が自身のスレッドで実行されると仮定します。図2-28で示されたセマフォを使った解決策は、このシステムで機能しますか。

- **[解答訳]** はい、そのまま機能します。ある時点で、1つの生産者（消費者）だけが、バッファに項目を追加（削除）できます。

- **[より詳しい解説]** 図2-28の解決策は、3つのセマフォ `mutex`, `empty`, `full` を使用しています。

  - **`mutex`**: このバイナリセマフォは、バッファへのアクセス自体を保護するクリティカルリージョンを制御します。複数の生産者が同時に`insert_item`を呼び出したり、複数の消費者が同時に`remove_item`を呼び出したりするのを防ぎます。一度に1つのスレッドしかバッファを操作できないため、複数の生産者・消費者がいても問題ありません。
  - **`empty`と`full`**: これらはカウンティングセマフォで、それぞれ空きスロットの数と満杯スロットの数を管理します。生産者は`down(&empty)`で空きスロットを1つ消費し、`up(&full)`で満杯スロットを1つ増やします。消費者はその逆です。これらのセマフォは、生産者や消費者が何人いるかには依存せず、単にリソースの数をカウントしているだけなので、複数の生産者・消費者がいても正しく動作します。 したがって、この解法は複数の生産者と複数の消費者の場合にも拡張可能です。

30. 2つのプロセスP0とP1が関わる相互排除問題に対する以下の解決策を考えます。変数turnは0に初期化されていると仮定します。プロセスP0のコードは以下の通りです。 … この解決策が、正しい相互排除の解決策に必要なすべての条件を満たしているか判断しなさい。

- **[解答訳]** この解決策は相互排除を満たします。なぜなら、turnが0のときP0はクリティカルセクションを実行できるがP1はできず、その逆も同様だからです。しかし、これは2つのプロセスが厳密に交互に実行されることを要求しており、望ましくありません。もしP1が何かを生産してバッファに入れ、P0がそれを取り出す場合を考えます。P0はバッファが空であることを見つけてブロックすることがあります。また、この解決策は、プロセスがクリティカルセクションの外でブロックされることを要求します。例えば、P1が何かを生産してバッファに入れ、その後、非クリティカルセクションをループし続けると、P0はいつまでも待たされることになります。

- **[より詳しい解説]** このアルゴリズムは、図2-23で示された「厳密な交互実行」のアルゴリズムと同じです。相互排除に必要な4つの条件について評価します。

    1. **相互排除**: 満たされています。turnは0か1のどちらかしか取り得ず、turnが0の間はP1はループで待機し、turnが1の間はP0がループで待機するため、同時にクリティカルリージョンに入ることはありません。
    2. **速度やCPU数に関する仮定なし**: この条件は満たされています。
    3. **非クリティカルリージョンで実行中のプロセスが他をブロックしない**: **満たされていません**。これがこのアルゴリズムの致命的な欠陥です。例えば、P0がクリティカルリージョンを抜けてturnを1にした後、P0が自身の非クリティカルリージョンにいる間に、P1がクリティカルリージョンを実行し、turnを0に戻したとします。この後、P0が再びクリティカルリージョンに入ろうとしても、turnが0であるため入れません。P1が非クリティカルリージョンで何をしているかに関わらず、P1が再びクリティカルリージョンに入ってturnを1にするまで、P0はブロックされます。
    4. **永久に待たされない**: この条件も**満たされていません**。上記のシナリオで、もしP1がクリティカルリージョンに入らずにループしたり、終了したりした場合、P0は永久に待たされることになります。 結論として、この解法は相互排除は実現しますが、進行（progress）と有界待機（bounded waiting）の条件を満たさないため、正しい解決策とは言えません。

31. 割り込みを無効にできるOSは、どのようにセマフォを実装できますか。

- **[解答訳]** セマフォ操作を行うには、オペレーティングシステムはまず割り込みを無効にします。次に、セマフォの値を読み取ります。もしdown操作を行っていてセマフォがゼロに等しい場合、呼び出し元のプロセスを、そのセマフォに関連付けられたブロックされたプロセスのリストに置きます。もしup操作を行っている場合、そのセマフォでブロックされているプロセスがあるか確認する必要があります。もし1つ以上のプロセスがブロックされているなら、そのうちの1つをブロックされたプロセスのリストから削除し、実行可能にします。これらすべての操作が完了したら、割り込みを再び有効にすることができます。

- **[より詳しい解説]** セマフォ操作（upとdown）は、**アトミック（不可分）** でなければなりません。つまり、値のチェック、更新、プロセスのスリープ/ウェイクアップといった一連の処理が、他のプロセスに中断されることなく完了する必要があります。シングルプロセッサシステムでは、これを実現する最も簡単な方法が割り込みの禁止です。

    1. **割り込み禁止**: upまたはdownのシステムコールハンドラの冒頭で、CPUの割り込みを無効にする命令を実行します。
    2. **セマフォ操作**:
        - downの場合: セマフォ値をチェックし、正であればデクリメントします。ゼロであれば、現在のプロセスをセマフォの待機キューに追加し、プロセスの状態をブロックに変更します。
        - upの場合: セマフォ値をインクリメントします。もし待機キューにプロセスがあれば、そのうちの1つを取り出し、状態を準備完了に変更します。
    3. **割り込み有効化**: すべての処理が終わったら、CPUの割り込みを有効に戻し、スケジューラを呼び出して次のプロセスを実行します。 割り込みを禁止している間は、クロック割り込みも発生しないため、プロセスの切り替えが起こりません。これにより、セマフォ操作のアトミック性が保証されます。ただし、この方法はマルチプロセッサ環境では機能しません。なぜなら、1つのCPUで割り込みを禁止しても、他のCPUはセマフォにアクセスできるからです。

32. バイナリセマフォと通常の機械語命令だけを使って、カウンティングセマフォ（任意の値を保持できるセマフォ）をどのように実装できるか示しなさい。

- **[解答訳]** 各カウンティングセマフォには、2つのバイナリセマフォ、M（相互排除用）とB（ブロッキング用）が関連付けられています。また、各カウンティングセマフォには、upの回数からdownの回数を引いた数を保持するカウンタと、そのセマフォでブロックされているプロセスのリストがあります。downを実装するには、プロセスはまずMに対してdownを実行して排他アクセス権を得ます。次にカウンタをデクリメントします。もしカウンタが0以上であれば、Mに対してupを実行して終了します。もしMが負であれば、プロセスはブロックされ

たプロセスのリストに置かれます。その後、Mに対してup、Bに対してdownを実行してプロセスをブロックします。upを実装するには、まずMをdownして相互排除権を得てから、カウンタをインクリメントします。もしカウンタが0より大きいなら、誰もブロックされていなかったので、Mをupするだけで完了です。しかし、カウンタが負または0になった場合は、リストからプロセスを削除する必要があります。最後に、その順でBとMに対してupを実行します。

- **[より詳しい解説]** カウンティングセマフォSを、値を持つ整数value、相互排除用のバイナリセマフォM（初期値1）、プロセスをブロックさせるためのバイナリセマフォB（初期値0）を使って実装します。

    - **down(S)の実装**:
        1. down(M) // クリティカルリージョンに入る
        2. value = value - 1
        3. もし value < 0 ならば:
            a. up(M) // 他のスレッドが操作できるようにMを解放
            b. down(B) // 自身をブロックさせる
        4. そうでなければ:
            a. up(M) // クリティカルリージョンを出る
    - **up(S)の実装**:
        1. down(M) // クリティカルリージョンに入る
        2. value = value + 1
        3. もし value <= 0 ならば:
            a. up(B) // 待機しているプロセスを1つ起こす
        4. up(M) // クリティカルリージョンを出る down操作でvalueが負になった場合、それは自身が待機状態に入ることを意味します。up操作では、valueが0以下（つまり、待機しているプロセスが存在する）の場合にのみ、up(B)で待機プロセスを1つ解放します。Mによって、valueの操作と、それに伴うBの操作がアトミックに行われることが保証されます。

33. システムに2つのプロセスしかない場合、それらを同期させるためにバリアを使う意味はありますか。それはなぜですか、あるいはなぜですか。

- **[解答訳]** もしプログラムがフェーズで動作し、どちらのプロセスも、両方が現在のフェーズを終えるまで次のフェーズに進むことができないのであれば、バリアを使うことは非常に理にかなっています。

- **[より詳しい解説]** バリアは、複数のプロセス（またはスレッド）が特定の同期ポイントに到達するまで、すべてのプロセスを待機させる同期プリミティブです。 2つのプロセスしかない場合でも、バリアは有効です。例えば、2つのプロセスが並列計算を行っており、計算がいくつかのステップ（フェーズ）に分かれているとします。

    - **フェーズ1**: プロセスAとプロセスBがそれぞれ異なるデータを処理する。
    - **同期ポイント**: 次のフェーズに進む前に、両方のプロセスがフェーズ1の計算を完了している必要がある。例えば、プロセスAがプロセスBの計算結果を必要とする場合など。
    - **フェーズ2**: 両者がフェーズ1を完了した後、次の計算ステップに進む。 このような状況では、各フェーズの終わりにバリアを設けることで、2つのプロセスの歩調を合わせることができます。バリアがなければ、一方のプロセスが他方の完了を待つために、セマフォや条件変数など、別の同期メカニズムを実装する必要があります。バリアは、このような「待ち合わせ」のシナリオをよりシンプルに記述するためのツールです。

34. 同じプロセス内の2つのスレッドは、スレッドがカーネルによって実装されている場合、カーネルセマフォを使って同期できますか。ユーザースペースで実装されている場合はどうですか。他のプロセスのスレッドはそのセマフォにアクセスできないと仮定します。あなたの答えを議論しなさい。

- **[解答訳]** カーネルスレッドでは、スレッドはセマフォでブロックでき、カーネルは同じプロセス内の別のスレッドを実行できます。したがって、セマフォを使うのに問題はありません。ユーザーレベルスレッドでは、1

つのスレッドがセマフォでブロックすると、カーネルはそのプロセス全体がブロックされたと考え、二度と実行しません。したがって、プロセスは失敗します。

- **[より詳しい解説]** この問題は、問12や問16と同様に、ユーザーレベルスレッドとカーネルレベルスレッドでのブロッキングの扱いの違いを問うています。

    - **カーネルレベルスレッド**: カーネルは各スレッドを個別に認識しています。したがって、あるスレッドがカーネルセマフォ（システムコール）でブロックした場合、カーネルは単にそのスレッドの状態をブロックに変更し、同じプロセス内の他の実行可能なスレッドをスケジュールすることができます。よって、**問題なく同期できます**。
    - **ユーザーレベルスレッド**: カーネルはスレッドの存在を知りません。もしユーザーレベルスレッドがカーネルセマフォでブロックするシステムコールを発行すると、カーネルはそのプロセス全体をブロック状態にします。これにより、他のすべてのユーザーレベルスレッドも実行できなくなり、セマフォをupしてくれるはずの他のスレッドも動けなくなってしまいます。結果として、プロセスはデッドロックに陥ります。したがって、**カーネルセマフォを直接使うことはできません**。ユーザーレベルスレッドが同期を行う場合は、ユーザー空間で実装された、ブロックしない（あるいはランタイムシステムに制御を戻す）独自の同期プリミティブ（ミューテックスなど）を使用する必要があります。

35. モニタ内の同期は、条件変数とwaitおよびsignalという2つの特別な操作を使用します。より一般的な同期形式は、任意のブール述語をパラメータとして持つ単一のプリミティブwaituntilを持つことです。…この方式は明らかにHoareやBrinch Hansenのものより一般的ですが、使われていません。なぜですか。（ヒント：実装について考えなさい。）

- **[解答訳]** 実装が非常に高価になります。あるプロセスが待機している述語に含まれる変数が変更されるたびに、ランタイムシステムは述語を再評価して、プロセスをアンブロックできるかどうかを確認する必要があります。HoareとBrinch Hansenのモニタでは、プロセスはsignalプリミティブでのみ起動できます。

- **[より詳しい解説]** `waituntil(Predicate)` というプリミティブは、プログラマにとっては非常に便利です。複雑な条件（例：`x < 0 or y + z < n`）を直接記述でき、どのスレッドがいつsignalを呼び出すべきかを気にする必要がなくなります。しかし、これを実装するOSやランタイムシステムの立場からすると、大きな困難が伴います。

    - **状態の常時監視**: `Predicate`には、モニタ内の任意の変数が含まれる可能性があります。システムは、この述語に含まれる**すべての変数**がいつ変更されたかを常に監視しなければなりません。
    - **高コストな再評価**: いずれかの変数が変更されるたびに、システムは待機中のすべてのプロセスの**すべての述語**を再評価する必要があります。この評価は、プロセスのクリティカルリージョン内で安全に行われなければならず、非常にコストが高くなります。 一方、`wait`と`signal`によるモデルでは、プロセスが起床するのは`signal`が明示的に呼び出されたときだけです。システムは変数の変更を監視する必要がなく、`signal`が呼ばれたときに待機キューからプロセスを1つ取り出すだけで済みます。この実装の単純さと効率性が、`wait/signal`モデルが広く採用されている理由です。

36. あるファストフード店には4種類の従業員がいます。(1)注文係、(2)調理師、(3)袋詰め担当、(4)レジ係です。各従業員は通信する逐次プロセスと見なせます。彼らはどのような形のプロセス間通信を使っていますか。このモデルをUNIXのプロセスと関連付けなさい。

- **[解答訳]** 従業員はメッセージを渡すことで通信しています：注文、食べ物、袋などです。UNIXの用語では、4つのプロセスはパイプで接続されています。

- **[より詳しい解説]** このシナリオは、プロセス間通信（IPC）の典型的なモデルを示しています。

    - **通信形式**: 従業員間のやり取りは**メッセージパッシング**の一形態です。注文票が注文係から調理師へ、調理された品が調理師から袋詰め担当へ、というように、一方向の情報の流れが存在します。
    - **UNIXとの関連**: このモデルは、UNIXの**シェルパイプライン**に非常によく似ています。各従業員は個別のプロセス（フィルタープログラム）と見なせます。あるプロセスの標準出力が次のプロセスの標準入力に接

続されるように、各工程がパイプで直列に接続されている状態です。 bash　　order_taker | cook | packager | cashier このように、UNIXの強力なパイプ機構は、実世界の直線的なワークフローをモデル化するのに非常に適しています。

37. メールボックスを使ったメッセージパッシングシステムがあるとします。満杯のメールボックスに送信しようとしたり、空のメールボックスから受信しようとしたりした場合、プロセスはブロックしません。代わりにエラーコードが返されます。プロセスはそのエラーコードに対し、成功するまで何度も再試行します。この方式は競合状態を引き起こす可能性がありますか。

- **[解答訳]** 競合状態にはつながりません（何も失われることはありません）が、事実上のビジーウェイトです。
- **[より詳しい解説]** この方式は**ノンブロッキング**なメッセージパッシングです。
    - **競合状態**: 教科書で定義されている競合状態（複数のプロセスが共有データを読み書きし、最終結果が実行タイミングに依存する状態）は発生しません。なぜなら、メールボックスという仲介者がおり、送信・受信操作は（OSによって）アトミックに行われるため、メッセージが途中で壊れたり失われたりすることはないからです。エラーコードが返されるということは、操作が「失敗した」ことが明確に通知されるため、プロセスは安全に再試行できます。
    - **ビジーウェイト**: プロセスはエラーを受け取るとすぐに再試行するため、CPU時間を無駄に消費します。これは、ロックが解放されるのを待ってループし続ける**スピンロック**と同じ問題です。特に、メールボックスの状態がすぐに変わる見込みがない場合、この方法は非常に非効率的になります。

38. CDC 6600コンピュータは、プロセッサ共有と呼ばれる興味深いラウンドロビンの一形態で、最大10個のI/Oプロセスを同時に処理できました。プロセスの切り替えは各命令後に行われ、…オーバーヘッドはゼロでした。…T秒かかるプロセスは、n個のプロセスでプロセッサ共有が使われた場合、どれくらいの時間が必要ですか。

- **[解答訳]** nT秒かかります。
- **[より詳しい解説]** このスケジューリング方式は、非常に細かい粒度でのラウンドロビンと言えます。
    - 各プロセスは、n命令ごとに1命令だけ実行する機会を得ます。つまり、自分の番が回ってくるまでに、他のn-1個のプロセスが1命令ずつ実行します。
    - 単独で実行した場合にT秒かかるとします。これは、ある一定数の命令を実行するのにかかる時間です。
    - プロセッサ共有環境では、あるプロセスが1命令を実行するために、実質的にn命令分の時間が経過します。
    - したがって、単独でT秒かかる処理を完了するためには、そのn倍の実時間、つまり **nT秒** が必要になります。これは、各プロセスがCPUの1/nの性能しか得られないことを意味します。このモデルは、コンテキストスイッチのオーバーヘッドがゼロであるという理想的な仮定に基づいています。

39. 以下のCコードを考えます。 `void main( ) { fork( ); fork( ); exit( ); }` このプログラムの実行時に、いくつの子プロセスが生成されますか。

- **[解答訳]** 3つのプロセスが生成されます。最初のforkの後に、親と子の2つのプロセスが存在します。これらのそれぞれが再びforkするため、合計4つのプロセスが存在することになります。そのうち3つは、元のプロセスの子（または孫）です。
- **[より詳しい解説]** fork()システムコールは、呼び出し元のプロセスを複製して新しい子プロセスを生成します。
    1. **最初の `fork()`**:
        - 元のプロセス（P0）が fork() を呼び出します。
        - 新しい子プロセス（P1）が生成されます。
        - この時点で、P0とP1の2つのプロセスが存在します。
    2. **2番目の `fork()`**:

        - P0とP1の両方が、プログラムの次の命令である2番目の `fork()` を実行します。
        - P0が `fork()` して、新しい子プロセスP2を生成します。
        - P1が `fork()` して、新しい子プロセスP3を生成します。
    3. **結果**:
        - 最終的に、P0（オリジナル）、P1（P0の子）、P2（P0の子）、P3（P1の子）の4つのプロセスが存在します。
        - 生成された子プロセスの総数は **3つ**（P1, P2, P3）です。

40. ラウンドロビンスケジューラは通常、実行可能なすべてのプロセスのリストを保持し、各プロセスはリストに1回だけ現れます。もしプロセスがリストに2回現れたらどうなりますか。これを許可する理由は考えられますか。

- **[解答訳]** もしプロセスがリストに2回現れた場合、サイクルごとに2つのクォンタムを得ることになります。この方法は、プロセスに他のプロセスよりも高い優先度を与えるために使うことができます。しかし、プロセスがブロックした場合、リスト上のすべてのエントリを削除する必要があり、実装が複雑になります。

- **[より詳しい解説]** ラウンドロビンは基本的にすべてのプロセスを平等に扱いますが、このテクニックは優先度の概念を導入する一つの方法です。

    - **動作**: 通常のラウンドロビンでは、準備完了キューを一周する間に各プロセスは1クォンタムずつ実行機会を得ます。もしあるプロセスがキューに2回登録されていれば、一周する間に2クォンタム分の実行機会を得ることになり、実質的に他のプロセスの2倍のCPU時間を獲得します。
    - **利点**: 優先度スケジューリングを実装する別の方法として機能します。重要なプロセスを複数回キューに入れることで、CPUシェアを増やすことができます。
    - **欠点**: 実装が複雑になります。特に、プロセスがブロック状態に遷移する場合、そのプロセスに対応する**すべてのエントリ**を準備完了キューから見つけ出して削除し、待機キューに移動させなければなりません。これを効率的に行うのは簡単ではありません。

41. プロセスがCPUバウンドかI/Oバウンドかの尺度は、ソースコードを分析して決定できますか。実行時にこれをどのように決定できますか。

- **[解答訳]** 単純なケースでは、ソースコードを調べることでI/Oが制限要因になるかどうかを見分けることができるかもしれません。例えば、入力ファイルをバッファに読み込んでから処理するプログラムは、おそらくI/Oバウンドではないでしょう。しかし、複数の異なるファイルにインクリメンタルに読み書きするプログラム（コンパイラなど）は、I/Oバウンドである可能性が高いです。もしオペレーティングシステムが、プログラムが使用したCPU時間を教えてくれる機能（UNIXのpsコマンドなど）を提供しているなら、プログラムの実行完了までにかかる合計時間と比較することができます。これは、もちろん、あなたが唯一のユーザーであるシステムで最も意味があります。

- **[より詳しい解説]** CPUバウンドかI/Oバウンドかは、プロセスの動的な振る舞いを指します。

    - **ソースコード分析（静的分析）**: 限定的です。プログラムが大量の計算ループを含んでいればCPUバウンド、多数の`read`/`write`コールを含んでいればI/Oバウンドと推測できますが、これはあくまで推測です。実際の振る舞いは入力データや実行時の環境に依存します。
    - **実行時分析（動的分析）**: OSは実行時にプロセスの振る舞いを監視することで、より正確に判断できます。一般的な方法は、プロセスがタイムクォンタムを使い切る頻度を観察することです。
        - **CPUバウンドなプロセス**: 頻繁にタイムクォンタムを使い切ってプリエンプトされます。
        - **I/Oバウンドなプロセス**: タイムクォンタムを使い切る前に、自発的にI/O待ちでブロック状態になることがほとんどです。 多くのスケジューラ（LinuxのCFSなど）は、このような実行時の振る舞いに基づいて動的にプロセスの優先度を調整し、I/Oバウンドなプロセス（対話的プロセス）の応答性を高めようとします。

42. ラウンドロビンスケジューリングアルゴリズムにおいて、タイムクォンタムの値とコンテキストスイッチの時間が互いにどのように影響するか説明しなさい。

- **[解答訳]** もしコンテキストスイッチ時間が大きい場合、タイムクォンタムの値も比例して大きくする必要があります。そうでなければ、コンテキストスイッチのオーバーヘッドが非常に高くなる可能性があります。典型的なCPUバースト時間がタイムクォンタムよりも短い場合、システムの非効率性につながる可能性があります。もしコンテキストスイッチが非常に小さいか無視できるほどであれば、タイムクォンタムの値はより自由に選択できます。

- **[より詳しい解説]** タイムクォンタムQとコンテキストスイッチ時間Sの間には、重要なトレードオフが存在します。

    - **CPU効率**: CPU効率は、有用な処理時間と合計時間（有用な処理時間＋オーバーヘッド）の比で表せます。クォンタムごとにコンテキストスイッチが発生すると仮定すると、CPU効率はおおよそ `Q / (Q + S)` となります。
    - **Qが小さい場合**: QがSに対して小さいと（例: `Q=4ms, S=1ms`）、オーバーヘッドの割合（`S / (Q + S)`）が大きくなり（この例では20%）、CPU効率が低下します。しかし、対話的プロセスへの応答時間は短くなります。
    - **Qが大きい場合**: QがSに対して大きいと（例: `Q=100ms, S=1ms`）、オーバーヘッドの割合は小さく（この例では約1%）なり、CPU効率は向上します。しかし、多くのプロセスが待機している場合、短い対話的プロセスへの応答時間が長くなり、ユーザーの体感性能が悪化します。 したがって、クォンタムの長さは、CPU効率と応答性の間のバランスを取るように慎重に決定される必要があります。

43. あるシステムの測定によると、平均的なプロセスはI/Oでブロックする前にTの時間だけ実行されます。プロセスの切り替えにはSの時間が必要で、これは事実上無駄（オーバーヘッド）になります。クォンタムQのラウンドロビンスケジューリングについて、…CPU効率の式を求めなさい。

- **[解答訳]** CPU効率は、有用なCPU時間を合計CPU時間で割ったものです。Q ≥ Tの場合、基本的なサイクルはプロセスがTだけ実行し、Sのプロセスイッチを伴うことです。したがって、(a)と(b)の効率はT/(S+T)です。クォンタムがTより短い場合、Tの実行にはT/Q回のプロセスイッチが必要で、ST/Qの時間を浪費します。ここでの効率は `T / (T + ST/Q)` となり、これは `Q/(Q+S)` に簡約されます。これが(c)の答えです。(d)では、QをSで置き換えると、効率は50%になります。最後に(e)では、Q→0となると効率は0になります。

- **[より詳しい解説]** この問題は、前問のトレードオフを数式でモデル化するものです。

    - **(a) Q = ∞ (無限)**: これは事実上、ノンプレンプティブなFCFSと同じです。プロセスはブロックするまでT時間実行し、その後Sのスイッチが発生します。1サイクルの合計時間はT+Sで、そのうち有用なのはTなので、効率は **`T / (T + S)`** です。
    - **(b) Q > T**: プロセスはクォンタムを使い切る前にブロックするため、プリエンプションは発生しません。状況は(a)と同じで、効率は **`T / (T + S)`** です。
    - **(c) S < Q < T**: プロセスは平均してT/Q回のプリエンプションを受けます。各実行時間はQで、その後Sのスイッチが発生します。合計T時間実行するために、T/Q回のサイクルが必要で、各サイクルの時間はQ+Sです。したがって、合計時間は `(T/Q) * (Q+S) = T + ST/Q` となります。有用な時間はTなので、効率は **`T / (T + ST/Q) = Q / (Q + S)`** となります。
    - **(d) Q = S**: (c)の式にQ=Sを代入すると、効率は **`S / (S + S) = 1/2 = 50%`** となります。CPU時間の半分がオーバーヘッドになります。
    - **(e) Qがほぼ0**: (c)の式でQを0に近づけると、効率は **0** に収束します。CPU時間はすべてコンテキストスイッチに費やされます。

44. 5つのジョブが実行を待っています。それらの予想実行時間は9、6、3、5、およびXです。平均応答時間を最小化するためには、どのような順序で実行すべきですか。

- **[解答訳]** 最短ジョブ優先が平均応答時間を最小化する方法です。 0 < X ≤ 3: X, 3, 5, 6, 9 3 < X ≤ 5: 3, X, 5, 6, 9 5 < X ≤ 6: 3, 5, X, 6, 9 6 < X ≤ 9: 3, 5, 6, X, 9 X > 9: 3, 5, 6, 9, X

- **[より詳しい解説]** 最短ジョブ優先（Shortest Job First, SJF）は、平均ターンアラウンドタイム（この問題では

応答時間と同じ）を最小化する最適なアルゴリズムです。したがって、ジョブを単純に実行時間の短い順に並べるだけです。Xの値がどの位置に来るかによって、全体の順序が変わります。

- もしXが3以下なら、Xが最も短いため先頭に来ます。
- もしXが3より大きく5以下なら、3の次にXが来ます。
- このように、Xの値を既知の実行時間と比較し、ソートされたリストの適切な位置に挿入することで、各ケースの最適な順序が決定されます。

45. 5つのバッチジョブAからEが、…到着しました。それらの推定実行時間はそれぞれ10、6、2、4、8分です。それらの…優先度はそれぞれ3、5、2、1、4で、5が最高優先度です。以下の各スケジューリングアルゴリズムについて、平均プロセスターンアラウンド時間を求めなさい。

- **[解答訳]**
  - **(a) ラウンドロビン**: 最初の10分間、各ジョブはCPUの1/5を得ます。10分後、Cが終了します。次の8分間、各ジョブはCPUの1/4を得て、その後Dが終了します。次に、残りの3つのジョブがそれぞれCPUの1/3を6分間得て、Bが終了します。といった具合です。5つのジョブの終了時刻は、10, 18, 24, 28, 30で、平均は22分です。
  - **(b) 優先度スケジューリング**: Bが最初に実行されます。6分後に終了します。他のジョブの終了時刻は14, 24, 26, 30で、平均は18.8分です。
  - **(c) 先着順**: AからEの順で実行された場合、終了時刻は10, 16, 18, 22, 30で、平均は19.2分です。
  - **(d) 最短ジョブ優先**: 終了時刻は2, 6, 12, 20, 30で、平均は14分です。
- **[より詳しい解説]** ターンアラウンドタイムは、ジョブが到着してから完了するまでの時間です。ここでは全てのジョブがほぼ同時に到着したと仮定します。
  - **(a) ラウンドロビン**: ここでは、CPUが公平に分配される理想的なプロセッサ共有を仮定して計算します。
    - C (2分)が最初に終わる。2分 * 5プロセス = 10分。
    - D (4分)が次に終わる。(4-2)分 * 4プロセス = 8分。合計18分。
    - B (6分)が次に終わる。(6-4)分 * 3プロセス = 6分。合計24分。
    - E (8分)が次に終わる。(8-6)分 * 2プロセス = 4分。合計28分。
    - A (10分)が最後に終わる。(10-8)分 * 1プロセス = 2分。合計30分。
    - 平均: (10+18+24+28+30)/5 = 22分。
  - **(b) 優先度スケジューリング**: 優先度の高い順（5,4,3,2,1）に実行します。対応するジョブは B, E, A, C, D。
    - B(6分)終了: 6
    - E(8分)終了: 6+8=14
    - A(10分)終了: 14+10=24
    - C(2分)終了: 24+2=26
    - D(4分)終了: 26+4=30
    - 平均: (6+14+24+26+30)/5 = 20分。**注：解答集の18.8分は計算が異なります。B(5)，E(4)，A(3)，C(2)，D(1)の順。6，6+8=14，14+10=24，24+2=26，26+4=30。平均は(6+14+24+26+30)/5=20。解答集の計算はB(6)，E(8)，A(10)，D(4)，C(2)の順で計算すると6，14，24，28，30となり平均22.4分。解答集の18.8分という値は、優先度とジョブの対応がB(5)，E(4)，A(3)，D(1)，C(2)と仮定すると6，14，24，28，30となり平均22.4分。いずれにせよ、解答集の値は再現できません。ここでは優先度順にB,E,A,C,Dとします。**
  - **(c) 先着順 (FCFS)**: A,B,C,D,Eの順で実行します。
    - A(10分)終了: 10
    - B(6分)終了: 10+6=16
    - C(2分)終了: 16+2=18
    - D(4分)終了: 18+4=22
    - E(8分)終了: 22+8=30
    - 平均: (10+16+18+22+30)/5 = 19.2分。
  - **(d) 最短ジョブ優先 (SJF)**: C,D,B,E,Aの順（2,4,6,8,10）で実行します。
    - C(2分)終了: 2

- D(4分)終了: 2+4=6
- B(6分)終了: 6+6=12
- E(8分)終了: 12+8=20
- A(10分)終了: 20+10=30
- 平均: (2+6+12+20+30)/5 = 14分。

46. CTSSで実行されるあるプロセスが完了するのに30クォンタム必要です。最初に（全く実行される前に）スワップインされるのを含め、何回スワップインされなければなりませんか。

- **[解答訳]** 最初は1クォンタム得ます。その後の実行では2, 4, 8, 15クォンタムを得るので、合計5回スワップインされる必要があります。

- **[より詳しい解説]** CTSS（Compatible Time Sharing System）は多重レベルキューを使用しており、実行されるたびにクォンタムが倍増します。

  1. **1回目**: スワップインされ、1クォンタム実行。残り29クォンタム。
  2. **2回目**: スワップインされ、2クォンタム実行。残り27クォンタム。
  3. **3回目**: スワップインされ、4クォンタム実行。残り23クォンタム。
  4. **4回目**: スワップインされ、8クォンタム実行。残り15クォンタム。
  5. **5回目**: スワップインされ、残りの15クォンタムを実行して完了。 したがって、合計で**5回**スワップインされる必要があります。

47. 周期5ミリ秒、1回のCPU時間が1ミリ秒の2つの音声通話と、周期33ミリ秒、1回のCPU時間が11ミリ秒の1つのビデオストリームを持つリアルタイムシステムを考えます。このシステムはスケジューリング可能ですか。

- **[解答訳]** 各音声通話は1/5=200ミリ秒のCPU時間を必要とし、2つで合計400ミリ秒です。ビデオは1秒間に33と1/3回、11ミリ秒を必要とし、合計で約367ミリ秒です。合計は767ミリ秒/秒であり、システムはスケジューリング可能です。

- **[より詳しい解説]** リアルタイムシステムがスケジューリング可能であるための必要条件は、CPU利用率の合計が1以下であることです。CPU利用率は (実行時間) / (周期) で計算されます。

  - 音声通話1: `1ms / 5ms = 0.2`
  - 音声通話2: `1ms / 5ms = 0.2`
  - ビデオストリーム: `11ms / 33ms ≈ 0.333`
  - 合計CPU利用率: `0.2 + 0.2 + 0.333 = 0.733` 合計利用率は0.733であり、1よりも小さいので、システムは**スケジューリング可能**です。

48. 上記の問題について、もう1つのビデオストリームを追加しても、システムはスケジューリング可能ですか。

- **[解答訳]** もう1つのビデオストリームは1秒あたり367ミリ秒を消費し、合計で1134ミリ秒/秒となるため、システムはスケジューリング可能ではありません。

- **[より詳しい解説]** 前問の合計利用率に、もう1つのビデオストリームの利用率を追加します。

  - 合計CPU利用率: `0.733`（前問の結果）+ `0.333`（追加のビデオ）= `1.066` 合計利用率が1.066となり、1を超えてしまいます。したがって、システムは**スケジューリング可能ではありません**。

49. a = 1/2 のエージングアルゴリズムが実行時間を予測するために使われています。過去4回の実行時間（古いものから順に）は40、20、40、15ミリ秒です。次の時間の予測値は何ですか。

- **[解答訳]** 予測のシーケンスは40, 30, 35、そして現在は25です。

- **[より詳しい解説]** エージングアルゴリズムは、予測値(n+1) = a * 実測値(n) + (1 - a) * 予測値(n) で計算されます。ここではa = 1/2です。

- 初期予測値 T0 = 40
- 1回目の実測値 T1 = 20 → 次の予測値 = (1/2)*20 + (1/2)*40 = 30
- 2回目の実測値 T2 = 40 → 次の予測値 = (1/2)*40 + (1/2)*30 = 35
- 3回目の実測値 T3 = 15 → 次の予測値 = (1/2)*15 + (1/2)*35 = 25 したがって、次の時間の予測値は**25ミリ秒**です。

50. あるソフトリアルタイムシステムには、…4つの周期的イベントがあります。…システムがスケジューリング可能であるためのxの最大値は何ですか。

- **[解答訳]** CPU使用率の分数は 35/50 + 20/100 + 10/200 + x/250 です。これが1以下であるためには、xは12.5ミリ秒未満でなければなりません。

- **[より詳しい解説]** ここでもCPU利用率の合計が1以下であるという条件を使います。 35/50 + 20/100 + 10/200 + x/250 ≤ 1 0.7 + 0.2 + 0.05 + x/250 ≤ 1 0.95 + x/250 ≤ 1 x/250 ≤ 0.05 x ≤ 12.5 したがって、xの最大値は**12.5ミリ秒**です。

51. 食事する哲学者の問題で、次のプロトコルが使われるとします。偶数番号の哲学者は常に左のフォークを取ってから右のフォークを取ります。奇数番号の哲学者は常に右のフォークを取ってから左のフォークを取ります。このプロトコルはデッドロックのない操作を保証しますか。

- **[解答訳]** はい。常に少なくとも1本のフォークがフリーであり、少なくとも1人の哲学者が両方のフォークを同時に取得できます。したがって、デッドロックはありません。N=2, N=3, N=4で試してみて、一般化することができます。

- **[より詳しい解説]** このプロトコルは、デッドロックの4条件のうち**循環待ち(Circular Wait)**を破壊します。

  - デッドロックが発生するのは、すべての哲学者が同時に1本のフォーク（例えば全員が左）を持ち、隣の哲学者が持つもう1本のフォークを待つ場合です。
  - このプロトコルでは、哲学者のうち1人（例えば最後の奇数番号の哲学者）は、他の哲学者とは逆の順序でフォークを取ろうとします。
  - 例えば5人の哲学者がいるとします。0,2,4番の哲学者は左（フォーク0,2,4）を先に取ろうとします。1,3番の哲学者は右（フォーク2,4）を先に取ろうとします。
  - もしデッドロックが発生するとしたら、全員が1本のフォークを持っている状態です。
    - 哲学者0はフォーク0を持つ。
    - 哲学者1は右のフォーク2を持つ。
    - 哲学者2は左のフォーク2を持つことはできない（哲学者1が持っているため）。 このように、全員が同時に1本目のフォークを取って循環待ちに陥るという状況が作れません。この方法は、リソースに順序を付けて要求することでデッドロックを防ぐアプローチの一例です。

52. あるリアルタイムシステムは、…2つの音声通話と…1つのビデオを処理する必要があります。このシステムはスケジューリング可能ですか。

- **[解答訳]** 各音声通話は1秒間に166.67回実行され、1バーストあたり1ミリ秒のCPU時間を消費します。したがって、各音声通話は1秒あたり166.67ミリ秒を必要とし、2つで合計333.33ミリ秒です。ビデオは1秒間に25回実行され、各回20ミリ秒を消費するので、合計で1秒あたり500ミリ秒です。両方を合わせると1秒あたり833.33ミリ秒を消費するので、時間は余り、システムはスケジューリング可能です。

- **[より詳しい解説]** 周期は 1秒 / 実行回数 で計算できます。

  - 音声通話の周期: 6ms。CPU利用率 = 1ms / 6ms ≈ 0.1667。2つで0.3333。
  - ビデオの周期: 1000ms / 25フレーム = 40ms。CPU利用率 = 20ms / 40ms = 0.5。
  - 合計CPU利用率: 0.3333 + 0.5 = 0.8333。 合計利用率は1より小さいので、システムは**スケジューリング可能**です。

53. カーネルスレッドのスケジューリングにおいて、ポリシーとメカニズムを分離することが望まれるシステムを考えます。この目標を達成する手段を提案しなさい。

- **[解答訳]** カーネルはいかなる手段でもプロセスをスケジュールできますが、各プロセス内でスレッドを厳密な優先度順に実行します。ユーザプロセスにスレッドの優先度を設定させることで、ユーザがポリシーを制御し、カーネルがメカニズムを処理します。

- **[より詳しい解説]** ポリシーとメカニズムの分離は、システムの柔軟性とモジュール性を高めるための重要な設計原則です。

    - **メカニズム（機構）**: カーネルが提供します。これは「どのように行うか」という機能そのものです。例えば、「スレッドに優先度を割り当て、最も優先度の高いスレッドを実行する」という機能です。カーネルは、スレッドの切り替え、優先度キューの管理といった基本的なスケジューリング機構を実装します。
    - **ポリシー（方針）**: ユーザプロセスが決定します。これは「何をすべきか」という判断基準です。ユーザプロセスは、システムコールを通じて自身のスレッドに優先度を割り当てることができます。どのスレッドが重要で、どのスレッドの応答性が重要かを知っているのはアプリケーション自身であるため、この分離は理にかなっています。 これにより、カーネルは汎用的なスケジューリングメカニズムを提供するだけで済み、アプリケーションは自身のニーズに合わせてスケジューリングポリシーを調整できます。

54. 食事する哲学者の問題の解決策（図2-47）において、なぜtake_forksプロシージャで状態変数がHUNGRYに設定されるのですか。

- **[解答訳]** 哲学者がブロックした場合、隣人が後でその哲学者が空腹であることをtestで確認でき、フォークが利用可能になったときにその哲学者を起こすことができるようにするためです。

- **[より詳しい解説]** state配列は、モニタ内の他の哲学者が各哲学者の状態を知るための共有変数です。take_forks内でまずstate[i] = HUNGRYと設定するのは、「私は今からフォークを取りたいと思っています」という意思表示です。 その後test(i)が呼ばれ、もし両隣が食事中でなければ、すぐにEATING状態に移行できます。 しかし、もし両隣のどちらかが食事中であれば、test(i)では何も起こらず、哲学者はセマフォs[i]でdownしてブロックされます。 このとき、state[i]はHUNGRYのままです。これにより、後で隣人がput_forksを呼び出したときに、そのput_forks内のtest関数がstate[i]をチェックし、「お、隣人は空腹だったな。フォークが空いたから起こしてあげよう」と判断してup(&s[i])を呼び出すことができるのです。もしHUNGRY状態がなければ、隣人はブロックされた哲学者を起こすことができません。

55. 図2-47のput_forksプロシージャを考えます。変数state[i]が、2回のtest呼び出しの後ではなく、その前にTHINKINGに設定されたとします。この変更は解決策にどのように影響しますか。

- **[解答訳]** この変更は、ある哲学者が食事を終えた後、その隣人のどちらも次に選ばれなくなることを意味します。実際、彼らは二度と選ばれません。test内のstate[LEFT] != EATINGの条件が、iがTHINKING状態にあるために常に満たされるからです。つまり、隣人たちは決して食事をすることができません。

- **[より詳しい解説]** この変更は、アルゴリズムを破壊します。test(i)関数が正しく動作するための条件は state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING です。

    - **元のコード**: put_forksでは、state[i]を**THINKING**にする**前に** test(LEFT)とtest(RIGHT)を呼び出します。これにより、隣人をテストするとき、iの状態はまだEATINGです。例えば、test(LEFT)がLEFTの右隣（つまりi）をチェックするとき、state[i]はEATINGなので、LEFTは食事を始められません。これは正しい動作です。
    - **変更後のコード**: put_forksでstate[i]を**THINKING**にした**後に** testを呼び出すと問題が起きます。
        - state[i] = THINKINGとする。
        - test(LEFT)を呼び出す。この関数は、LEFTの右隣であるiの状態をチェックします。state[i]はTHINKINGなので、state[RIGHT] != EATINGの条件は満たされます。これにより、LEFTがEATING状態になる可能性があります。
        - test(RIGHT)を呼び出す。この関数は、RIGHTの左隣であるiの状態をチェックします。state[i]

はTHINKINGなので、state[LEFT] != EATINGの条件は満たされます。これにより、RIGHTもEATING状態になる可能性があります。 結果として、LEFTとRIGHTの両方が同時にEATING状態になる可能性があり、これは2人が1本のフォークを共有することを意味し、相互排除の条件を破ります。解答の指摘とは少し異なりますが、根本的な問題は、隣人をテストする際に、自分自身が食事中であるという情報が失われてしまうことです。

56. 読み手書き手問題は、どのカテゴリのプロセスが開始できるかに関して、いくつかの方法で定式化できます。…3つの異なるバリエーションを慎重に説明しなさい…

- **[解答訳]**（この問題は設計演習のため、解答集に解答はありません）
- **[より詳しい解説]** 読み手・書き手問題は、共有データへのアクセス制御をモデル化する古典的な問題です。基本的なルールは「複数の読み手は同時アクセス可能、書き手は排他アクセスが必要」ですが、そのポリシーにはバリエーションがあります。

  1. **読み手優先**: このバリエーションでは、書き手が待機中であっても、新しい読み手はアクセスを許可されます。読み手が一人でもいる限り、後から来た読み手はすぐにアクセスできます。
     - **動作**: 書き手は、読み手が一人もいなくなるまで待たされます。読み手が継続的に到着する場合、書き手は**飢餓状態（starvation）** に陥る可能性があります。
     - **実装**: 教科書図2-48のコードがこのバリエーションに相当します。
  2. **書き手優先**: このバリエーションでは、書き手が待機している場合、新しい読み手はアクセスを許可されず、書き手の後ろで待たされます。
     - **動作**: 書き手が到着すると、現在アクセス中の読み手が終了するのを待った後、すぐにアクセス権を得ます。後から来た読み手は、待機中のすべての書き手が終了するまで待たされます。このポリシーでは、読み手が飢餓状態に陥る可能性があります。
  3. **公平な（FIFO）バージョン**: このバリエーションでは、読み手と書き手は到着順にサービスを受けます。
     - **動作**: 待機キューを一つ用意し、到着したプロセスを順に並べます。先頭が読み手なら、後続の読み手も（書き手が来るまで）アクセスを許可します。先頭が書き手なら、現在のアクセスがすべて終了するのを待ってから、その書き手一人がアクセスします。これにより、どちらのカテゴリも飢餓状態に陥ることを防ぎますが、読み手の並列性が損なわれる可能性があります。

57. ファイルの最後の番号を読み取り、それに1を加えてファイルに追加することで、連続番号のファイルを生成するシェルスクリプトを書きなさい。…競合状態を防ぐようにスクリプトを修正しなさい。

- **[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]** この演習の目的は、ファイル操作における競合状態を実際に体験し、ロック機構を使ってそれを解決することです。

  - **競合状態を発生させるスクリプト**: `bash #!/bin/bash # # warning: this script has a race condition if [ ! -f numbers ]; then echo 0 > numbers; fi count=0 while [ $count -ne 200 ]; do count=$(expr $count + 1) n=$(tail -1 numbers) expr $n + 1 >> numbers done` このスクリプトを2つ同時に実行すると（`./script.sh & ./script.sh`）、両者がほぼ同時に`tail`で同じ番号を読み、同じ番号を`expr`でインクリメントして書き込むため、ファイル内の数値が抜けたり重複したりします。
  - **クリティカルリージョン**: `n=$(tail -1 numbers)`と`expr $n + 1 >> numbers`の間の部分です。この2つの操作はアトミックに行われる必要があります。
  - **競合を防ぐ修正**: ヒントにあるように、`ln`コマンドを使ってロックファイルを作成することで相互排除を実現できます。`ln file file.lock`は、`file.lock`が存在しない場合にのみ成功するアトミックな操作です。 `bash #!/bin/bash # this script is free of race conditions if [ ! -f numbers ]; then echo 0 > numbers; fi count=0 while [ $count -ne 100 ]; do if ln numbers numbers.lock >/dev/null 2>&1; then count=$(expr $count + 1) n=$(tail -1 numbers) expr $n + 1 >> numbers rm numbers.lock fi done` このバージョ

ンでは、lnが成功した場合（ロックを取得できた場合）のみクリティカルリージョンが実行され、終了時にrmでロックが解放されます。

58. セマフォを提供するOSがあると仮定します。メッセージシステムを実装しなさい。メッセージを送受信するためのプロシージャを書きなさい。

- **[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、より低レベルな同期プリミティブ（セマフォ）を使って、より高レベルなIPCメカニズム（メッセージパッシング）を構築するものです。これは、教科書図2-28の生産者・消費者問題の実装と非常によく似ています。 実装方針:
  - **共有バッファ**: メッセージを格納するための共有メモリ領域（バッファ）を用意します。
  - **セマフォ**:
    - mutex: バッファへのアクセスを保護するためのバイナリセマフォ（初期値1）。
    - empty: バッファの空きスロット数を数えるカウンティングセマフォ（初期値はバッファサイズN）。
    - full: バッファ内のメッセージ数を数えるカウンティングセマフォ（初期値0）。
  - **send(message)プロシージャ**:
    1. down(&empty): 空きスロットができるまで待つ。
    2. down(&mutex): バッファへの排他アクセス権を得る。
    3. バッファにメッセージをコピーする。
    4. up(&mutex): 排他アクセス権を解放する。
    5. up(&full): メッセージが1つ増えたことを通知する。
  - **receive(&buffer)プロシージャ**:
    1. down(&full): メッセージが来るまで待つ。
    2. down(&mutex): バッファへの排他アクセス権を得る。
    3. バッファからメッセージを取り出す。
    4. up(&mutex): 排他アクセス権を解放する。
    5. up(&empty): 空きスロットが1つ増えたことを通知する。

59. 食事する哲学者の問題を、セマフォではなくモニタを使って解きなさい。

- **[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]** モニタは、共有データとそれに対する操作をカプセル化し、自動的に相互排除を提供する高レベルな同期プリミティブです。 実装方針:
  - **モニタ**: DiningPhilosophersというモニタを定義します。
  - **共有データ（モニタ内）**:
    - state[N]: 各哲学者の状態（THINKING, HUNGRY, EATING）を保持する配列。
    - self[N]: 各哲学者が待機するための条件変数の配列。
  - **モニタプロシージャ**:
    - take_forks(i):
      1. state[i] = HUNGRYと設定。
      2. test(i)を呼び出して、フォークが取れるか試す。
      3. もしstate[i] != EATINGなら（フォークが取れなかったなら）、self[i].wait()で待機する。
    - put_forks(i):
      1. state[i] = THINKINGと設定。
      2. test(LEFT)とtest(RIGHT)を呼び出し、両隣が食事を始められるか確認し、可能なら起こす（signalする）。
    - test(i) (プライベートプロシージャ):
      1. もしstate[i] == HUNGRYかつstate[LEFT] != EATINGかつstate[RIGHT] != EATINGならば、
      2. state[i] = EATINGとし、self[i].signal()で待機中の自分（または他のプロセス）を起こ

す。 この構造は、セマフォを使った解法（図2-47）と論理的に非常に似ていますが、相互排除（mutexセマフォに相当する部分）がモニタによって自動的に保証されるため、プログラマは条件同期（self条件変数）にのみ集中できます。

60. ある大学が、…トイレ問題を解決するためのプロシージャを書きなさい。

- **[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]** この問題は、「読み手・書き手問題」の変形です。
    - 「女性」と「男性」は、それぞれ「読み手」のカテゴリに対応します。同じ性別の人は同時にトイレに入れます。
    - ただし、女性の読み手グループと男性の読み手グループは互いに排他的です。
    - トイレが空の状態から最初に入る人が、そのトイレを「女性用」または「男性用」にロックする、一種の書き込み操作と見なせます。 セマフォやモニタを使って実装できます。モニタを使う場合、状態変数としてstate（EMPTY, WOMEN_PRESENT, MEN_PRESENT）、トイレ内の人数count、そして待機中の人数を管理する変数と条件変数が必要になります。
    - woman_wants_to_enter:
        - もしstate == MEN_PRESENTならば、女性用の条件変数で待機する。
        - 入室し、countをインクリメントし、state = WOMEN_PRESENTとする。
    - woman_leaves:
        - countをデクリメントする。
        - もしcount == 0 ならば、state = EMPTYとし、待機中の男性がいれば起こす（signalまたはbroadcast）。 男性用のプロシージャも同様です。公平性を保つ（例えば、男性が待っているのに女性が次々に入り続けるのを防ぐ）ためには、待機中の人数を考慮するなどの追加のロジックが必要になります。

61. 図2-23のプログラムを、2つ以上のプロセスを扱えるように書き直しなさい。

- **[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]** 図2-23のアルゴリズムは、2つのプロセス間で厳密に交互に実行権を渡すもので、N個のプロセスには拡張できません。N個のプロセスで相互排除を実現するには、ピーターソンのアルゴリズムのNプロセス版や、**パン屋のアルゴリズム（Lamport's bakery algorithm）**のような、より高度なソフトウェアアルゴリズムが必要です。 パン屋のアルゴリズムの概要:
    1. **番号札**: クリティカルリージョンに入りたいプロセスは、パン屋のように番号札を取る。番号は単調増加する。
    2. **待機**: プロセスは、自分より小さい番号札を持つすべてのプロセスがクリティカルリージョンを抜けるのを待つ。
    3. **実行**: 自分の番が来たらクリティカルリージョンに入る。
    4. **解放**: 終了したら番号札を0に戻す。 このアルゴリズムは、複数のプロセスが同じ番号札を取ってしまう可能性を考慮するなど、実装は複雑ですが、N個のプロセスに対してデッドロックフリーな相互排除を保証します。

62. スレッドを使い、…競合状態が発生するまでにどれくらいかかるか見てみましょう。

- **[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、同期プリミティブなしで共有バッファにアクセスした場合に、競合状態が実際に発生することを体験することです。sleepとwakeupを使っても、教科書で説明されている**「失われたウェイクアップ（lost wakeup）」問題**が発生する可能性があります。 実装のポイント:
    - グローバル変数として、バッファ、インデックス（in, out）、カウンタ（count）を宣言します。

- 生産者スレッド:
  - `while (count == N) sleep();`
  - バッファにアイテムを追加。
  - `count++;`
  - `if (count == 1) wakeup(consumer);`
- 消費者スレッド:
  - `while (count == 0) sleep();`
  - バッファからアイテムを削除。
  - `count--;`
  - `if (count == N-1) wakeup(producer);` countのインクリメント/デクリメント（count++やcount--）は、アセンブリレベルでは「読み出し・変更・書き込み」の3ステップに分かれているため、この操作の途中でスレッドが切り替わると競合が発生し、カウンタの値が不正確になります。これにより、バッファが満杯なのに生産者がアイテムを追加しようとしたり、空なのに消費者が取り出そうとしたりして、プログラムがクラッシュするかデッドロックに陥ります。競合が発生するまでの時間は、システムや負荷によって異なりますが、通常はそれほど時間はかかりません。

63. あるプロセスをラウンドロビンのキューに複数回入れることで、…CPUのより大きなシェアを獲得できるはずです。

- **[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]** この問題は、2つの異なるアプローチによるCPUシェアの獲得を試す演習です。

1. **素数判定プログラムの作成**: まず、与えられた数値が素数かどうかを判定するCPUバウンドなプログラムを作成します。
2. **並列化**: 複数のプロセスが、大きな数値リストの異なる部分を並行して処理するように実装します。共有リソース（次に処理すべき数値のインデックスなど）へのアクセスには、ファイルロックなどの同期メカニズムが必要です。例えば、全プロセスが共有するファイルに次に処理すべき数値のインデックスを格納し、各プロセスはファイルをロックしてインデックスを読み取り、インクリメントしてからアンロックします。
3. **性能評価**: 単一プロセスで実行した場合と、複数のプロセスで実行した場合の合計処理時間を比較します。
   - **シングルユーザーのPC**: PC上で他に重いタスクが実行されていなければ、CPUコアはすでにこのプログラム群に占有されています。プロセス数を増やしても、利用可能なコア数を超えると、コンテキストスイッチのオーバーヘッドで逆に遅くなる可能性があります。
   - **共有システム**: 他にも多くのプロセスが実行されているシステムでは、自分のプロセス数を増やすことで、ラウンドロビンスケジューラからより多くのタイムスライスを獲得できる可能性があります。これにより、システム全体のCPU時間のうち、自分のタスク群が使用する割合が増え、結果としてリストの処理が速くなることが期待できます。

64. この演習の目的は、与えられた数が完全数であるかどうかを判断するためのマルチスレッドの解決策を実装することです。

- **[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]** この問題は、マルチスレッドプログラミングと同期の実践演習です。

1. **完全数の定義**: N自身を除く約数の和がNに等しい数（例: 6 = 1+2+3）。
2. **入力**: N（検査対象の数）とP（スレッド数）。
3. **タスク分割**: 検査範囲（1からN-1、またはヒントにあるように1から$\sqrt{N}$まで）をP個のスレッドに均等に分割します。例えば、スレッドi (0 <= i < P) は、(i * N/P) + 1 から ((i+1) * N/P) までの範囲を担当します。
4. **並列処理**: 各スレッドは担当範囲の数値を順に調べ、Nの約数であれば、その数値を共有バッファ（共有変

数）に加算します。
5. **同期**: 共有バッファ（約数の和を保持する変数）は、複数のスレッドから同時に書き込まれる可能性があるため、**ミューテックス**で保護する必要があります。各スレッドは、約数を見つけて共有バッファに加算する前にミューテックスをロックし、加算後にアンロックします。
6. **結果の集約**: メインスレッドは、pthread_createでP個のスレッドを生成した後、pthread_joinですべてのスレッドが終了するのを待ちます。
7. **判定**: すべてのスレッドが終了したら、メインスレッドは共有バッファに集計された約数の和とNを比較し、完全数かどうかを判定して結果を出力します。

65. テキストファイル内の単語の頻度を数え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]** これは、並列処理における「MapReduce」パターンの単純な実装例です。

1. **ファイル分割**: 入力となる大きなテキストファイルをN個のセグメントに分割します。分割は、ファイルサイズをNで割ることで、各スレッドが担当する開始オフセットと終了オフセットを決定します。単語がセグメント境界で分断されないように注意が必要です（例えば、境界付近の空白文字を探すなど）。
2. **Mapフェーズ**: N個のスレッドを生成します。各スレッドは、割り当てられたセグメントを読み込み、単語の出現頻度をカウントします。このとき、各スレッドは**自身専用の**データ構造（ハッシュマップなど）に中間結果を格納します。これにより、Mapフェーズではスレッド間の同期が不要になり、効率が向上します。
3. **中間結果の出力**: 各スレッドは、自身のセグメントの処理が終わったら、中間的な単語頻度データを（例えば、スレッドごとの一時ファイルに）出力します。
4. **Reduceフェーズ**: メインスレッドは、pthread_joinですべてのスレッドが終了するのを待ちます。
5. **結果の統合**: すべてのスレッドが終了したら、メインスレッドはN個の中間結果をすべて読み込み、それらを1つの最終的なハッシュマップに統合（Reduce）します。同じ単語の出現回数をすべて合算します。
6. **最終出力**: 統合された最終的な単語頻度データを出力します。