第6章 デッドロック 演習問題の解答と解説

1. 政治から取られたデッドロックの例を挙げなさい。

- **[解答訳]** 米国において、3人以上の候補者がある政党の指名を争う大統領選挙を考えます。すべての予備選挙が終わり、代議員が党大会に到着したとき、どの候補者も過半数を得ておらず、どの代議員も投票先を変える意思がないという状況が起こり得ます。これがデッドロックです。各候補者はいくつかのリソース（票）を持っていますが、目的を達成するためにはさらに多くのリソースを必要とします。議会に複数の政党が存在する国では、各党がそれぞれ異なる年間予算案を支持し、予算を可決するための過半数を形成することが不可能になる場合があります。これもまたデッドロックです。

- **[より詳しい解説]** デッドロックとは、「あるプロセスの集合がデッドロック状態にあるとは、その集合内の各プロセスが、その集合内の他のプロセスしか引き起こすことのできないイベントを待っている状態」と定義されます。 政治の例では、プロセスは「候補者」や「政党」に、リソースは「票」や「議席」に相当します。各候補者（プロセス）は、指名獲得（タスク完了）に必要な過半数の票（リソース）を得るために、他の候補者の支持者（他のプロセスが保持するリソース）の投票を待っています。しかし、どの支持者も投票先を変えないため、どの候補者も必要な票を得ることができず、永久に待ち続けることになります。これは**循環待ち**の条件に合致しています。

2. コンピュータ研究室の個々のPCで作業している学生が、ハードディスク上のスプール領域にファイルをスプールするサーバーに印刷ファイルを送信します。プリントスプールのディスク領域が限られている場合、どのような状況でデッドロックが発生する可能性がありますか。そのデッドロックはどのように回避できますか。

- **[解答訳]** スプールパーティション上のディスク領域は有限のリソースです。例えば、スプール領域が10MBで、10個の2MBのジョブの最初の半分が到着した場合、ディスクは満杯になり、それ以上のブロックは保存できなくなるためデッドロックが発生します。このデッドロックは、ジョブが完全にスプールされる前に印刷を開始し、解放された領域をそのジョブの残りの部分のために予約することで回避できます。この方法では、まず1つのジョブが完全に印刷され、次に別のジョブが同様に処理されます。

- **[より詳しい解説]** これは典型的なリソースデッドロックの例です。

    - **デッドロック発生の状況**: 複数の印刷ジョブ（プロセス）が、スプール領域という有限のリソースを要求します。各ジョブは一部のディスク領域を**保持したまま**、さらに追加の領域を**待機**します（**保持と待機**の条件）。もし、システム内の全てのジョブが、他のジョブが確保している領域が解放されるのを待つ状態になると、**循環待ち**が発生しデッドロックとなります。
    - **デッドロックの回避**: 解答で示されている方法は、「**横取りなし (no preemption)**」の条件を崩すアプローチです。ジョブが一部でも印刷を開始すれば、そのジョブが使用していたスプール領域が徐々に解放（横取り）され、他のジョブが待っているリソースが利用可能になります。もう一つの回避策は、「**保持と待機**」の条件を崩すことです。つまり、ジョブをスプールする前に、そのジョブ全体を格納するのに十分な領域を一度に確保します。もし十分な領域がなければ、ジョブは一切領域を確保せずに待機します。

3. 前の問題において、どのリソースがプリエンプティブル（横取り可能）で、どれがノンプリエンプティブル（横取り不可能）ですか。

- **[解答訳]** プリンタはノンプリエンプティブルです。システムは前のジョブが完了するまで別のジョブの印刷を開始できません。スプールディスクはプリエンプティブルです。大きすぎる不完全なファイルを削除し、ユーザーに後で再送させることができます（プロトコルがそれを許す場合）。

- **[より詳しい解説]** リソースの分類は、その性質によります。

    - **プリエンプティブル（横取り可能）リソース**: それを保持しているプロセスから、悪影響なく取り上げることができるリソースです。スプールディスク領域がこれにあたります。印刷ジョブの途中のスプールファイルを削除しても、最悪の場合ユーザーが印刷をやり直すだけで、システム全体に致命的な影響はあ

りません。メモリも典型的なプリエンプティブルリソースです。
    - **ノンプリエンプティブル（横取り不可能）リソース**: 現在の所有者から強制的に取り上げると、処理の失敗を引き起こすリソースです。プリンタがこれにあたります。あるジョブの印刷中にプリンタを横取りして別のジョブを印刷させると、印刷結果はめちゃくちゃになります。

4. 図6-1では、リソースは取得した順序とは逆の順序で返却されています。これを別の順序で返却しても同様にうまく機能しますか。

- **[解答訳]** はい。全く違いはありません。

- **[より詳しい解説]** デッドロックの発生は、リソースの**取得**フェーズにおける競合によって引き起こされます。プロセスが必要なリソースをすべて取得してしまえば、そのプロセスはもはや他のプロセスを待つことはなく、処理を進めていずれはリソースを解放します。リソースの**解放**順序は、他のプロセスが待っているかどうかには影響を与えません。どの順序で解放しても、解放されたリソースは待機しているプロセスに割り当てられます。したがって、解放の順序はデッドロックの発生とは無関係です。

5. 4つの条件（相互排他、保持と待機、横取りなし、循環待ち）は、リソースデッドロックが発生するために必要です。これらの条件がリソースデッドロックが発生するための十分条件ではないことを示す例を挙げなさい。これらの条件がリソースデッドロックの発生に十分となるのはどのような場合ですか。

- **[解答訳]** 3つのプロセスA, B, Cと、2種類のリソースR, Sを考えます。Rのインスタンスは1つ、Sのインスタンスは2つあるとします。以下の実行シナリオを考えます：AがRを要求して取得する。BがSを要求して取得する。CがSを要求して取得する（Sのインスタンスは2つある）。BがRを要求してブロックされる。AがSを要求してブロックされる。この段階で4つの条件はすべて満たされています。しかし、デッドロックは発生していません。Cが終了すると、Sのインスタンスが1つ解放され、それがAに割り当てられます。するとAは実行を完了でき、Rを解放します。それがBに割り当てられ、Bも実行を完了できます。これらの4つの条件は、各種類のリソースが1つしかない場合には十分条件となります。

- **[より詳しい解説]** この問題は、デッドロックの理論における重要な点を指摘しています。

    - **4条件が十分でない例**: リソース割り当てグラフにサイクルが存在しても、デッドロック状態にあるとは限りません。これは、サイクル外のプロセス（この例ではC）が、サイクル内のプロセス（A）が待っているリソース（S）を解放できる可能性があるためです。CがSを解放すると、AはSを獲得して処理を進め、やがてRを解放します。するとBもRを獲得でき、サイクルが解消されます。このように、**リソースの各種類に複数のインスタンスが存在する場合**、サイクルはデッドロックの必要条件ですが、十分条件ではありません。
    - **4条件が十分となる場合**: **各種類のリソースのインスタンスが1つしかない場合**、リソース割り当てグラフにサイクルが存在することは、デッドロックの十分条件となります。なぜなら、サイクル内の各プロセスは、サイクル内の次のプロセスが保持している（唯一の）リソースを待つことになり、サイクル外のプロセスがこの状況を打開することはできないからです。

6. 都市の街路は、グリッドロックと呼ばれる循環的な閉塞状態に陥りやすいです。…ニューヨーク市の予防アルゴリズムは「交差点を塞ぐな（don’t block the box）」と呼ばれ…これはどの予防アルゴリズムに該当しますか。…

- **[解答訳]**「交差点を塞ぐな」は事前割り当て戦略であり、「**保持と待機**」というデッドロックの前提条件を無効にします。なぜなら、我々は車が交差点の先の道路空間に入ることができると仮定し、それによって交差点を解放すると考えるからです。他の予防アルゴリズムとしては、車が一時的にガレージに退避してグリッドロックを解消するのに十分なスペースを確保することを許可する方法があります。一部の都市では、交通を形成するための交通制御ポリシーがあります。都市の道路が混雑するにつれて、交通管理者は赤信号の設定を調整し、非常に混雑したエリアへの交通流入を抑制します。交通量を減らすことでリソースの競合が少なくなり、グリッドロックの発生確率が低下します。

- **[より詳しい解説]** この問題は、デッドロック予防策を実世界の例に当てはめるものです。

- **「保持と待機」の打破**:「交差点を塞ぐな」のルールは、プロセス（車）が次のリソース（交差点の先の道路）を確保できる見込みがない限り、現在保持しているリソース（交差点の手前の道路）を保持したまま待つことを許しません。これは、「**プロセスはリソースを要求する前に、必要なリソースをすべて要求しなければならない**」という「保持と待機」の予防策に似ています。
- **他の予防策**:
  - **横取りなしの打破**: 解答にある「車がガレージに退避する」のは、リソース（道路上のスペース）を**横取り (preemption)** することに相当します。
  - **循環待ちの打破**: 一方通行の道路網を設計したり、交差点で右折のみを許可する（あるいは特定の順序でしか曲がれない）ルールを設けることで、**循環待ち**を構造的に防ぐことができます。

7. 4台の車がそれぞれ4つの異なる方向から同時に交差点に近づくとします。交差点の各角には一時停止標識があります。交通規則では、2台の車が隣接する一時停止標識に同時に近づいた場合、左側の車が右側の車に道を譲らなければならないと仮定します。したがって、4台の車がそれぞれの一時停止標識に近づくと、各車は左側の車が進むのを（無期限に）待ちます。この異常は通信デッドロックですか。それともリソースデッドロックですか。

- **[解答訳]** この異常は通信デッドロックではありません。なぜなら、これらの車は互いに独立しており、もし競合が発生しなければ、最小限の遅延で交差点を通過できたはずだからです。これはリソースデッドロックでもありません。なぜなら、どの車も他の車が要求しているリソースを保持していないからです。また、リソースの事前割り当てや横取りといったメカニズムも、この異常の制御には役立ちません。しかし、この異常は**競合同期 (competition synchronization)** の一種であり、車が循環的な連鎖の中でリソースを待っている状態です。リソースデッドロックと区別するために、この異常は「**スケジューリングデッドロック**」と呼ぶことができます。同様のデッドロックは、共有の鉄道路線に合流しようとする2つの列車が、互いに相手が先に行くのを待つことを法律で定められている場合にも発生する可能性があります。警官が競合する車や列車の一つに（他ではなく）進むよう信号を送ることで、このデッド状態は、ロールバックやその他のオーバーヘッドなしに解消できることに注意してください。

- **[より詳しい解説]** この問題は、デッドロックの定義を厳密に適用し、リソースデッドロックと通信デッドロックの境界を考察させるものです。

  - **デッドロックの定義**:「あるプロセスの集合がデッドロック状態にあるとは、その集合内の各プロセスが、その集合内の他のプロセスしか引き起こすことのできないイベントを待っている状態」です。この車の状況は、この一般的な定義には当てはまります。各車は、右隣の車が進むというイベントを待っており、そのイベントは循環しています。
  - **リソースデッドロックではない理由**: リソースデッドロックの4条件のうち、「**保持と待機 (Hold and Wait)**」が満たされていません [6.2.1]。どの車もリソース（交差点の特定の部分）を**保持**しておらず、単にリソースを**待機**しているだけです。全車が交差点に進入しようとして待っている状態です。
  - **通信デッドロックではない理由**: 通信デッドロックは、プロセスが互いにメッセージを待ち合うことで発生します [6.7.2]。車は互いに通信しているわけではありません。
  - **スケジューリングデッドロック**: 解答で示されているこの用語は、標準的なものではありませんが、状況を的確に表しています。ここでは、リソース（交差点）のスケジューリングポリシー（交通ルール「右方優先」）そのものが、循環待ちを引き起こし、デッドロックの原因となっています。解決策もスケジューリングによるもの、つまり警官（スケジューラ）が特定のプロセス（車）を強制的に先に進ませることで解決します。

8. あるリソースタイプの複数のユニットと、別のタイプの一つのユニットが関与するリソースデッドロックは可能ですか。もしそうなら、例を挙げなさい。

- **[解答訳]** はい、可能です。あるプロセスがあるリソースタイプの一部のユニットを保持し、別のリソースタイプを要求している一方で、別のプロセスがその2番目のリソースを保持し、最初のタイプのリソースの利用可能なユニットを要求している場合です。もし他のプロセスが最初のタイプのリソースのユニットを解放できず、そのリソースが横取り不可能であったり、同時に使用できなかったりする場合、システムはデッドロックに陥ります。 例えば、2つのプロセスが実メモリシステムのメモリセルを割り当てられているとします（ページやプ

ロセスのスワッピングはサポートされず、メモリの動的リクエストはサポートされると仮定）。最初のプロセスが別のリソース、例えばデータセルをロックします。2番目のプロセスがそのロックされたデータを要求し、ブロックされます。最初のプロセスは、データを解放するコードを実行するために、より多くのメモリを必要とします。システム内の他のプロセスが完了してメモリセルを解放できないと仮定すると、デッドロックが存在します。

- **[より詳しい解説]** この問題は、リソースのインスタンスが複数あってもデッドロックが発生しうることを確認するものです。
    - **プロセスA**: 2つのテープドライブを保持し、1つのプリンタを要求している。
    - 、リソース割り当てグラフにサイクルが存在し、かつ、サイクル内の各リソースタイプについて、そのリソースを待っているプロセスが、利用可能なインスタンス数よりも多くのインスタンスを要求している場合にデッドロックが発生することを示しています。

9. 図6-3はリソースグラフの概念を示しています。不正なグラフ、つまり、我々が使用してきたリソース利用のモデルを構造的に違反するグラフは存在しますか。もしそうなら、その例を挙げなさい。

- **[解答訳]** はい、不正なグラフは存在します。リソースは単一のプロセスによってのみ保持されると述べてきました。リソースの四角からプロセス円への矢印は、そのプロセスがリソースを所有していることを示します。したがって、一つの四角から二つ以上のプロセスへ矢印が出ているグラフは、それらのプロセスすべてがリソースを保持していることを意味し、これはルールに違反します。したがって、一つの四角から複数の矢印が異なる円に向かって出ているグラフは、リソースのインスタンスが複数存在しない限り、ルールに違反します。四角から四角へ、または円から円への矢印もルールに違反します。

- **[より詳しい解説]** リソース割り当てグラフのルールは、リソース利用のモデルを反映しています [6.2.2]。
    - **リソースノード（四角）から出る矢印**: 矢印 `R → P` は、プロセスPがリソースRを保持していることを示します。もしリソースRのインスタンスが1つしかない場合、Rから複数のプロセス（P1, P2, …）へ矢印が出ることは**不正**です。これは、単一のリソースを複数のプロセスが同時に保持していることを意味し、「**相互排他 (mutual exclusion)**」の原則に反します。
    - **プロセスノード（円）から出る矢印**: 矢印 `P → R` は、プロセスPがリソースRを要求していることを示します。一つのプロセスが同時に複数のリソースを要求することは可能なので、Pから複数のRへ矢印が出ることは**正当**です。
    - **その他の不正なグラフ**:
        - `R1 → R2` のような矢印は意味をなしません（リソースがリソースを保持することはない）。
        - `P1 → P2` のような矢印も、このモデルでは定義されていません（プロセスが直接プロセスを待つのではなく、リソースを介して待つ）。

10. 図6-4を考えます。ステップ(o)でCがRではなくSを要求したとします。これはデッドロックにつながりますか。SとRの両方を要求した場合はどうなりますか。

- **[解答訳]** どちらの変更もデッドロックにはつながりません。どちらの場合も循環待ちは存在しません。

- **[より詳しい解説]** 図6-4(o)の直前の状態（図6-4(n)）を分析します。
    - **状態**: プロセスAはリソースSを要求し、プロセスCはリソースRを要求しています。この時点で利用可能なリソースはありません。
    - **デッドロックフリーなシーケンス**: この後、AがRとSを解放し（図6-4(p), (q)）、Cが必要なRを獲得して完了し、Tを解放します。
    - **もしCがSを要求したら**: プロセスCはリソースSを要求してブロックされます。しかし、Aが完了すればRとSの両方を解放するため、ブロックされているCとBはその後進行できます。循環待ちは発生しません。
    - **もしCがSとRの両方を要求したら**: 同様に、Cはブロックされますが、Aが完了すれば両方のリソースが解放されるため、デッドロックには至りません。 **デッドロックが発生するのは、循環的な依存関係が形成された場合のみ**です [6.2.1]。これらのシナリオでは、プロセスAは他のどのプロセスが保持するリソースも

待CがAの保持するSを待っているとします。A, B, Cの3つのプロセスすべてがデッドロック状態にあります。しかし、循環鎖に含まれているのはAとBだけです。

- **[より詳しい解説]** この例のリソース割り当てグラフを描くと理解しやすくなります。
    - `A → S` (AがSを要求)
    - `S → B` (BがSを保持)
    - `B → R` (BがRを要求)
    - `R → A` (Aの一部ではありません。

12. トラフィックを制御するために、ネットワークルータAは定期的に隣接するルータBにメッセージを送り…そのメッセージが失われました。…これはどのタイプのデッドロックですか。

- **[解答訳]** これは明らかに**通信デッドロック**です。タイムアウト（ヒューリスティック）を用いてAがウィンドウサイズを増やす有効化メッセージを再送することで制御できます。しかし、Bが元のメッセージと重複したメッセージの両方を受信する可能性もあります。この重複を検出するためには、このようなメッセージにシーケンス番号を付けることが効果的です。

- **[より詳しい解説]** この状況は、リソースの競合ではなく、**通信の失敗によってプロセスが互いに待ち続ける状態**であるため、通信デッドロックに分類されます [6.7.2]。
    - **プロセスAの状態**: トラフィックが減少したので、Bからのデータ受信を再開する準備ができています。しかし、Bが送信を再開するには、Aからの許可メッセージが必要です。Aはメッセージを送ったと思っているので、Bからのデータを待っています。
    - **プロセスBの状態**: Aから送信停止を指示されたため、送信を停止しています。Aからの再開許可メッセージを待っています。
    - **デッドロック**: AはBからのデータを待ち、BはAからのメッセージを待っています。メッセージが失われたため、どちらのイベントも発生せず、両者は永久に待ち続けます。 解答にあるように、**タイムアウト**は通信デッドロックを検出・回復するための一般的なメカニズムです。Aは許可メッセージの応答（例えば、Bからの最初のデータパケット）が一定時間内に来なければ、メッセージが失われたと判断し、再送します。

13. ダチョウアルゴリズムの議論では、プロセステーブルのスロットや他のシステムテーブルが満杯になる可能性について言及されています。システム管理者がそのような状況から回復できるようにする方法を提案できますか。

- **[解答訳]** そのようなリソースの一部は、管理者所有のプロセスのみが使用できるよう予約しておくことができます。これにより、管理者はいかなるデッドロック状況でも常にシェルや必要なプログラムを実行でき、状況を評価し、どのプロセスを強制終了させてシステムを再び利用可能にするかを決定できます。

- **[より詳しい解説]** この問題は、ダチョウアルゴリズム（問題を無視する戦略） を採用した場合の現実的な回復策を問うています。システムテーブル（プロセステーブルなど）が満杯になることは、一種のリソース枯渇であり、デッドロックやライブロックにつながる可能性があります。
    - **問題点**: もし全てのプロセステーブルのスロットが一般ユーザーのプロセスで埋まってしまうと、システム管理者でさえ新しいプロセス（例えば、状況を調査するための `ps` コマンドや、プロセスを終了させるための `kill` コマンド）を起動できなくなります。
    - **解決策**: 解答が示すように、**リソースを予約しておく**ことが有効です。例えば、全1000個のプロセステーブルスロットのうち、最後の10個はスーパーユーザー（管理者）専用として予約します。一般ユーザーは990個までしか使えません。これにより、システムが一般ユーザーのプロセスで満杯になっても、管理者は常にログインして調査・回復作業を行うためのプロセスを起動できます。これは、デッドロックそのものを防ぐのではなく、**デッドロックが発生した際に手動で回復するための手段を確保する**という実用的なアプローチです。

14. 4つのプロセスP1, P2, P3, P4と5種類のリソースRS1, RS2, RS3, RS4, RS5を持つシステムの以下の状態を考えま

す…デッドロック検出アルゴリズムを使用して、システムにデッドロックがあることを示しなさい。

- **[解答訳]** まず、マークされていないプロセスの集合 P = {P1, P2, P3, P4} R1 は A 以下ではない R2 は A 以下である。P2をマークする。A = (0 2 0 3 1)。 P = {P1, P3, P4} R1 は A 以下ではない R3 は A と等しい。P3をマークする。A = (0 2 0 3 2)。 P = {P1, P4} R1 は A 以下ではない R4 は A 以下ではない したがって、プロセスP1とP4はマークされずに残る。これらがデッドロック状態にある。

- **[より詳しい解説]** この問題は、複数のインスタンスを持つリソースに対するデッドロック検出アルゴリズム の適用を問うものです。解答集に記載された数値（C, R, E, Aの各行列・ベクトル）に基づいてアルゴリズムを追跡します。

  1. **初期状態**:
     - E = (2 4 1 4 4) (存在するリソース総数)
     - C (現在割り当て) と R (要求) は問題文の通り。
     - A = (0 1 0 2 1) (利用可能なリソース)
     - 最初は全てのプロセス {P1, P2, P3, P4} がマークされていません。
  2. **第1パス**:
     - P1の要求 R1=(0 2 0 1 1) は A=(0 1 0 2 1) 以下ではありません（RS2が足りない）。
     - P2の要求 R2=(0 0 0 1 1) は A=(0 1 0 2 1) 以下です。**P2を実行可能**と判断し、マークします。
     - P2が完了したと仮定し、P2が保持していたリソース C2=(1 0 0 0 1) を A に返却します。
     - 新しい A = (0 1 0 2 1) + (1 0 0 0 1) = (1 1 0 2 2)。
  3. **第2パス**:
     - P1の要求 R1 は新しい A 以下ではありません（RS2が足りない）。
     - P3の要求 R3=(1 0 0 1 0) は A=(1 1 0 2 2) 以下です。**P3を実行可能**と判断し、マークします。
     - P3が完了したと仮定し、P3が保持していたリソース C3=(0 1 0 0 0) を A に返却します。
     - 新しい A = (1 1 0 2 2) + (0 1 0 0 0) = (1 2 0 2 2)。
  4. **第3パス**:
     - P1の要求 R1=(0 2 0 1 1) は A=(1 2 0 2 2) 以下です。**P1を実行可能**と判断し、マークします。
     - P1が完了したと仮定し、P1が保持していたリソース C1=(1 0 0 1 1) を A に返却します。
     - 新しい A = (1 2 0 2 2) + (1 0 0 1 1) = (2 2 0 3 3)。
  5. **第4パス**:
     - P4の要求 R4=(0 1 0 1 0) は A=(2 2 0 3 3) 以下です。**P4を実行可能**と判断し、マークします。 **結論**: 全てのプロセスを完了させるシーケンス（例: P2→P3→P1→P4）が存在するため、このシステムはデッドロック状態に**ありません**。 *（注：ご提供の解答集の結論と計算過程が異なります。解答集の計算では、Aの更新値が異なるか、CまたはRの行列値が設問と異なっている可能性があります。上記は教科書のアルゴリズムと設問の数値に基づいた正しい計算手順です。）*

15. 前の問題のデッドロックからシステムがどのように回復できるかを、以下の方法で説明しなさい。(a) 横取りによる回復 (b) ロールバックによる回復 (c) プロセス強制終了による回復

- **[解答訳]**
  - **(a) 横取りによる回復**: P2とP3が完了した後、P1はRS3の1ユニットを強制的に横取りさせられます。これによりA = (0 2 1 3 2) となり、プロセスP4が完了可能になります。P4が完了してリソースを解放すれば、P1も完了できます。
  - **(b) ロールバックによる回復**: P1を、RS3を取得する前にチェックポイントされた状態までロールバックします。
  - **(c) プロセス強制終了による回復**: P1を強制終了させます。
- **[より詳しい解説]** この解答は、前問でP1とP4がデッドロック状態にあるという（解答集の）結論に基づいています。デッドロックからの回復には主に3つの方法があります。
  - **(a) 横取り (Preemption)**: デッドロック状態にあるプロセスの一つ（P1）から、他のプロセス（P4）が必要としているリソース（RS3）を強制的に取り上げます。取り上げたリソースをP4に与えることでP4が実行を完了でき、リソースを解放します。その結果、デッドロックの循環が断ち切られ、最終的にP1も完了

できるようになります。
    - **(b) ロールバック (Rollback)**: プロセスが定期的に自身の状態（メモリイメージ、リソース状態など）をチェックポイントとして保存しておきます。デッドロックが検出されたら、デッドロックの原因となっているプロセス（P1）を、問題のリソースを取得する前のチェックポイントまで巻き戻します。これにより、P1が保持していたリソースが解放され、他のプロセスが進行可能になります。
    - **(c) プロセス強制終了 (Killing Processes)**: 最も単純で手荒い方法です。デッドロックの循環に関与しているプロセス（P1）を一つ選んで強制終了させます。これにより、そのプロセスが保持していた全てのリソースが解放され、デッドロックが解消されます。どのプロセスを終了させるかは、優先度や完了までの進捗、再実行の容易さなどを考慮して決定されます。

16. 図6-6において、あるiに対して `Cij + Rij > Ej` であるとします。これはシステムにどのような影響を及ぼしますか。

- **[解答訳]** そのプロセスは、システムが持つ以上のリソースを要求していることになります。そのプロセスは、たとえ他の全てのプロセスがどんなリソースも要求しなかったとしても、決して完了することはできません。

- **[より詳しい解説]** この式は、バンカーアルゴリズムやデッドロック検出アルゴリズムの前提条件に関わる重要な点を示しています。

    - `Cij`: プロセスiが現在保持しているリソースjの数。
    - `Rij`: プロセスiが今後さらに要求する可能性のあるリソースjの数。
    - `Ej`: システムに存在するリソースjの総数。`Cij + Rij`は、プロセスiがリソースjに対して**主張する最大必要量**を意味します。もしこの最大必要量が、システムに物理的に存在するリソースの総数Ejを超えている場合、それは論理的な矛盾です。そのプロセスiの要求は、他の全てのプロセスがリソースを解放したとしても、決して満たされることはありません。 このようなプロセスは、システム設計上、**最初から実行を許可されるべきではありません**。銀行家のアルゴリズムでは、プロセスが最初に最大必要量を申告した時点で、どのリソースjに対しても最大必要量 `<= Ej`であるかチェックされます。

17. 図6-8の軌跡はすべて水平または垂直です。斜めの軌跡も可能となるような状況を想像できますか。

- **[解答訳]** もしシステムに2つ以上のCPUがあれば、2つ以上のプロセスが並列に実行される可能性があり、斜めの軌跡も可能となります。

- **[より詳しい解説]** 図6-8のリソース軌跡グラフ は、**単一CPU**のシステムをモデル化しています。

    - **水平な線**: プロセスAが実行され、プロセスBが待機している状態を示します。
    - **垂直な線**: プロセスBが実行され、プロセスAが待機している状態を示します。 単一CPUでは、ある瞬間に実行できるプロセスは一つだけなので、軌跡は水平か垂直のどちらかになります。 しかし、**マルチプロセッサ（またはマルチコア）システム**では、複数のプロセスが**同時に**実行できます。もしプロセスAとプロセスBが異なるCPUで同時に実行されれば、両方の実行命令数が同時に増加していくことになり、これはグラフ上で**斜めの軌跡**として表現されます。

18. 図6-8のリソース軌跡の仕組みは、3つのプロセスと3つのリソースを持つデッドロック問題を示すためにも使用できますか。もしそうなら、どのようにできますか。もしできなければ、なぜですか。

- **[解答訳]** はい。3次元で同じことができます。z軸は、3番目のプロセスによって実行された命令の数を表します。

- **[より詳しい解説]** 2つのプロセスの状態は2次元平面（x-y平面）で表現できました。同様に、3つのプロセスの状態は**3次元空間（x-y-z空間）**で表現できます。

    - x軸: プロセスAの実行命令数
    - y軸: プロセスBの実行命令数
    - z軸: プロセスCの実行命令数 各リソースは、2つのプロセスによる同時使用を禁止する「平面」として表現

されます。例えば、リソースRをプロセスAとBが使用する場合、これはx-y平面上の禁止領域（長方形）となりますが、3次元ではこれがz軸方向に伸びた**直方体の禁止領域**となります。3つのプロセスと3つのリソースが関わるデッドロックは、これらの禁止領域によって形成される**進入不可能な空間**として視覚化できます。ただし、4つ以上のプロセスになると、人間が直感的に視覚化することはできなくなります。

19. 理論的には、リソース軌跡グラフはデッドロックを回避するために使用できます。巧妙なスケジューリングによって、オペレーティングシステムは危険な領域を避けることができます。これを実際に実行するための実用的な方法はありますか。

- **[解答訳]** この方法は、あるリソースが要求される正確な瞬間を事前に知っている場合にのみ、スケジューリングを導くために使用できます。実際には、これはめったにありません。

- **[より詳しい解説]** リソース軌跡グラフによるデッドロック回避は、**全てのプロセスが将来どのタイミングでどのリソースを要求・解放するかを完全に予測できる**ことを前提としています。これは、銀行家のアルゴリズムが最大リソース要求数を事前に知る必要があるのと似ていますが、さらに厳しく、要求の**タイミング**まで知る必要があります。 現実の汎用OSでは、プロセスの振る舞いは実行時の入力や条件によって変化するため、このような完全な予測は不可能です。したがって、このグラフはデッドロックの概念を視覚的に理解するためのモデルとしては優れていますが、実用的なデッドロック回避アルゴリズムとしてOSに実装することはできません。

20. システムがデッドロック状態でもなく安全状態でもない状態になることは可能ですか。もしそうなら、例を挙げなさい。もしできなければ、すべての状態がデッドロック状態か安全状態のどちらかであることを証明しなさい。

- **[解答訳]** デッドロックでも安全でもないが、デッドロック状態につながる状態は存在します。例として、テープ、プロッタ、スキャナ、CD-ROMという4種類のリソースと3つのプロセスを持つシステムを考えます。以下の状況がありえます： 確保している | 必要としている | 利用可能 —|—|— A: 2 0 0 0 | 1 0 2 0 | 0 1 2 1 B: 1 0 0 0 | 0 1 3 1 | C: 0 1 2 1 | 1 0 1 0 | この状態はデッドロックではありません。なぜなら、多くの動作がまだ可能だからです。例えば、Aはまだプリンタを2つ要求できます。しかし、もし各プロセスが残りの要求を全て行うと、デッドロックが発生します。

- **[より詳しい解説]**

    - **安全状態 (Safe state)**: 全てのプロセスが完了できるような実行順序が**少なくとも一つ存在する**状態です。システムはデッドロックを確実に回避できます。
    - **デッドロック状態 (Deadlocked state)**: 進行不可能な循環待ちが発生している状態です。
    - **危険状態 (Unsafe state)**: 安全状態でもなく、デッドロック状態でもない状態です。この状態からは、**デッドロックに陥る可能性がある**ものの、必ずしもそうなるとは限りません。プロセスが最大要求数よりも少ないリソースで完了する可能性があるためです。 解答の例は、まさにこの危険状態を示しています。現在利用可能なリソース (0 1 2 1) では、どのプロセスの要求も満たすことができません。しかし、これは即座にデッドロックを意味するわけではありません。例えば、プロセスAが保持しているプリンタを解放すれば、他のプロセスが進行できるかもしれません。しかし、もし全てのプロセスが最大要求数を要求し続けると、デッドロックは避けられません。したがって、この状態は**危険状態**です。

21. 図6-11(b)を注意深く見てください。もしDがもう1ユニット要求した場合、これは安全状態につながりますか、それとも危険状態につながりますか。要求がDではなくCから来た場合はどうなりますか。

- **[解答訳]** Dからの要求は危険状態につながりますが、Cからの要求は安全です。

- **[より詳しい解説]** 図6-11(b)の初期状態を確認します。

    - 各プロセスの(確保済み, 最大必要量): A(1, 6), B(1, 5), C(2, 4), D(4, 7)
    - 利用可能(Free): 2
    - **Dがもう1ユニット要求した場合**:
        - Dは5ユニット確保し、Freeは1になります。

        - 各プロセスの残り必要量: A(5), B(4), C(2), D(2)。
        - Free(1)では、どのプロセスの要求も満たせません。したがって、これは**危険状態**です。
    - **Cがもう1ユニット要求した場合**:
        - Cは3ユニット確保し、Freeは1になります。
        - 残り必要量: A(5), B(4), C(1), D(3)。
        - Free(1)でCの要求(1)を満たせます。Cが実行を完了すると、4ユニットを返却し、Freeは 1+4=5 になります。
        - Free(5)でBの要求(4)を満たせます。Bが完了すると、Freeは 5+5=10 になります。
        - Free(10)でAとDの要求をどちらも満たせます。
        - 完了シーケンス（例: C→B→A→D）が存在するため、これは**安全状態**です。

22. あるシステムには2つのプロセスと3つの同一のリソースがあります。各プロセスは最大2つのリソースを必要とします。デッドロックは可能ですか。あなたの答えを説明しなさい。

- **[解答訳]** このシステムはデッドロックフリーです。各プロセスが1つのリソースを持っていると仮定します。すると、1つのリソースが空いています。どちらかのプロセスがそれを要求して取得できます。その場合、そのプロセスは完了し、両方のリソースを解放できます。したがって、デッドロックは不可能です。

- **[より詳しい解説]** デッドロックが発生する最悪のシナリオを考えます。それは、各プロセスが必要なリソースを1つだけ残して、できるだけ多くのリソースを保持している状態です。

    - プロセス数: `p = 2`
    - 最大必要数: `m = 2`
    - 総リソース数: `r = 3` デッドロックの可能性がある状態は、各プロセスが `m-1 = 1` つのリソースを保持している場合です。
    - プロセス1が1つ保持。
    - プロセス2が1つ保持。 このとき、消費されたリソースは合計2つです。システムには `3 - 2 = 1` つの空きリソースが残っています。この空きリソースをどちらかのプロセスに割り当てれば、そのプロセスは要求を満たして完了し、保持していた2つのリソースを解放します。すると、残りのプロセスも完了できます。したがって、**デッドロックは不可能**です。

23. 前の問題を再び考えますが、今度はp個のプロセスがそれぞれ最大m個のリソースを必要とし、合計r個のリソースが利用可能であるとします。システムをデッドロックフリーにするためには、どのような条件が満たされなければなりませんか。

- **[解答訳]** あるプロセスがm個のリソースを持っていれば、それは完了でき、デッドロックに関与しえません。したがって、最悪のケースは、全てのプロセスがm-1個のリソースを持ち、さらにもう1つを必要とする場合です。もし1つでもリソースが残っていれば、1つのプロセスが完了し、全てのリソースを解放し、残りのプロセスも完了できます。したがって、デッドロックを回避する条件は `r >= p(m-1) + 1` です。

- **[より詳しい解説]** これは前問の一般化であり、デッドロックフリーであるための重要な条件式を導き出します。

    - **デッドロックが発生する最悪のシナリオ**: p個全てのプロセスが、完了するために必要な最後の1リソースを待っている状態。つまり、各プロセスが `m-1` 個のリソースを保持している状態です。
    - **この最悪の状態で保持されているリソースの総数**: `p * (m-1)` 個。
    - **デッドロックを回避するために**: この最悪の状態でも、少なくとも1つのプロセスを完了させられるだけの空きリソースがあればよい。1つのプロセスを完了させるには、あと1つのリソースが必要です。
    - したがって、`p * (m-1)` 個のリソースが全て保持されていても、**最低でも1つの空きリソース**があればデッドロックは回避できます。
    - よって、システム内の総リソース数 `r` は、`p * (m-1)` 個（最悪の状態で保持される数）と 1 個（デッドロックを破るために必要な数）の合計以上でなければなりません。
    - これが、条件式 **`r >= p(m-1) + 1`** の意味するところです。

24. 図6-12において、プロセスAが最後のテープドライブを要求したとします。この行動はデッドロックにつながりますか。

- **[解答訳]** いいえ。Dはまだ完了できます。Dが完了すると、A（またはE）が完了するのに十分なリソースを返却し、以下同様に続きます。

- **[より詳しい解説]** 図6-12の初期状態を確認します。

    - 利用可能 `A = (1 0 2 0)` (テープ, プロッタ, プリンタ, Blu-ray)
    - Aの残り必要量 `NeedA = (1 1 0 0)`
    - **Aが最後のテープドライブを要求した場合**:
        - 要求 `RequestA = (1 0 0 0)` は利用可能 `A = (1 0 2 0)` 以下なので、**仮に**割り当ててみます。
        - 仮割り当て後の状態:
            - 新しい利用可能 `A' = (0 0 2 0)`
            - Aの確保済み `AllocA' = (4 0 1 1)`
            - Aの残り必要量 `NeedA' = (0 1 0 0)`
    - **この仮の状態が安全かチェックします**:
        1. `A'=(0 0 2 0)` で、どのプロセスの残り要求も満たせません (Dの要求 `NeedD=(0 0 1 0)` はプリンタが足りない)。 *（注：解答集はDが完了できるとしていますが、図6-12の数値ではプリンタが足りず完了できません。ここでも解答集のロジックに従います。おそらく解答集の執筆者はA=(1 0 1 0)と誤読したか、図のPとAが矛盾していると考えている可能性があります。）*
    - **解答集のロジックに従った解説**: 解答集は「Dはまだ完了できる」としています。Dの残り要求は `(0 0 1 0)` です。もしこれが利用可能リソース以下であると仮定すると、
        1. Dが完了し、リソース `(1 1 0 1)` を返却します。Aは `(0 0 2 0) + (1 1 0 1) = (1 1 2 1)` になります。
        2. この新しいAで、Aの要求 `(0 1 0 0)` や Eの要求 `(2 1 1 0)` を満たすことができます。
        3. 完了シーケンスが存在するため、安全であると判断されます。したがって、この行動はデッドロックにつながりません。

25. m個のリソースクラスとn個のプロセスを持つシステムで銀行家のアルゴリズムが実行されています。…ある状態が安全かどうかをチェックするために実行しなければならない操作の数は、`m^a * n^b` に比例します。aとbの値は何ですか。

- **[解答訳]** 行列の行と利用可能リソースのベクトルを比較するには、m回の操作が必要です。このステップは、完了してマークされるプロセスを見つけるために、最大でn回繰り返される必要があります。したがって、1つのプロセスをマークするにはmnオーダーのステップが必要です。アルゴリズムを全てのn個のプロセスに対して繰り返すと、操作の数はmn^2になります。よって、a=1, b=2です。

- **[より詳しい解説]** 銀行家のアルゴリズム（の安全性チェック部分）の計算量を分析します。

    1. **内側のループ**: アルゴリズムは、完了可能なプロセスを探します。そのためには、マークされていない各プロセスiについて、その要求ベクトル `Need[i]` が利用可能ベクトル `Available` 以下であるか比較します。
        - `Need[i]` と `Available` はどちらも m 次元のベクトルです。この比較には m 回の要素ごとの比較が必要です。
        - 最悪の場合、マークされていない全てのプロセス（最大 n 個）をチェックする必要があります。
        - したがって、完了可能なプロセスを1つ見つけるための操作数は **O(mn)** です。
    2. **外側のループ**: この「完了可能なプロセスを探してマークする」という操作を、全てのプロセスがマークされるまで繰り返します。
        - 最悪の場合、一度に1つのプロセスしかマークできず、この操作を n 回繰り返す必要があります。
    3. **総計算量**: 内側のループ `O(mn)` を n 回繰り返すので、総計算量は **$O(mn^2)$** となります。
    - したがって、`m^a * n^b` に当てはめると、**a=1, b=2** となります。

26. あるシステムには4つのプロセスと5つの割り当て可能なリソースがあります。…これが安全状態であるためのxの

最小値は何ですか。

- **[解答訳]** Need（必要）行列は以下の通りです：０１００２０２１００１０３００００１１１もしxが0なら、即座にデッドロックです。もしxが1なら、プロセスDが完了できます。完了すると、利用可能ベクトルは (1 1 2 2 1) になります。残念ながら、ここからデッドロックです。もしxが2なら、Dが実行された後、利用可能ベクトルは (1 1 3 2 1) となり、Cが実行できます。Cが完了してリソースを返却すると、利用可能ベクトルは (2 2 3 3 1) となり、Bが実行・完了し、次にAが実行・完了できます。したがって、デッドロックを回避するためのxの最小値は2です。

- **[より詳しい解説]** この問題も銀行家のアルゴリズムの安全性チェックを応用するものです。

  1. まず、各プロセスが**今後必要とするリソース数** (Need = Max - Allocated) を計算します。
     - Need A = (1 1 2 1 3) - (1 0 2 1 1) = (0 1 0 0 2)
     - Need B = (2 2 2 1 0) - (2 0 1 1 0) = (0 2 1 0 0)
     - Need C = (2 1 3 1 0) - (1 1 0 1 0) = (1 0 3 0 0)
     - Need D = (1 1 2 2 1) - (1 1 1 1 0) = (0 0 1 1 1)
  2. **x=1の場合**:
     - 利用可能 Available = (0 0 1 1 1)。
     - Need D = (0 0 1 1 1) は Available 以下なので、Dは完了可能です。
     - D完了後、Available = (0 0 1 1 1) + Allocated D (1 1 1 1 0) = (1 1 2 2 1)。
     - この新しいAvailableで、A, B, CのNeedを満たすことはできません。よって**危険状態**です。
  3. **x=2の場合**:
     - 利用可能 Available = (0 0 2 1 1)。
     - Need D は Available 以下なので、Dは完了可能です。
     - D完了後、Available = (0 0 2 1 1) + (1 1 1 1 0) = (1 1 3 2 1)。
     - この新しいAvailableで、Need C = (1 0 3 0 0) を満たせます。Cは完了可能です。
     - C完了後、Available = (1 1 3 2 1) + (1 1 0 1 0) = (2 2 3 3 1)。
     - このAvailableで、Need B = (0 2 1 0 0) を満たせます。Bは完了可能です。
     - B完了後、Available = (2 2 3 3 1) + (2 0 1 1 0) = (4 2 4 4 1)。
     - このAvailableで、Need A = (0 1 0 0 2) を満たせます。Aは完了可能です。
     - 完了シーケンス (D→C→B→A) が存在するので、**安全状態**です。 したがって、xの最小値は**2**です。

27. 循環待ちをなくす一つの方法は、プロセスが一度に一つのリソースしか保有できないという規則を設けることです。多くの場合、この制約が受け入れられないことを示す例を挙げなさい。

- **[解答訳]** テープからプリンタへ巨大なファイルをコピーする必要があるプロセスを考えます。メモリ量が限られており、ファイル全体がメモリに収まらないため、プロセスは以下の文をファイル全体が印刷されるまでループさせる必要があります： テープドライブを取得 → ファイルの次の部分をメモリにコピー → テープドライブを解放 → プリンタを取得 → メモリからファイルを印刷 → プリンタを解放 これはプロセスの実行時間を長くします。さらに、プリンタは印刷ステップごとに解放されるため、ファイルの全ての部分が連続したページに印刷される保証はありません。

- **[より詳しい解説]** この制約は、「**保持と待機 (Hold and Wait)**」の条件を非常に厳しく制限することで**循環待ち (Circular Wait)** を間接的に防ごうとするものです。しかし、多くのタスクは複数のリソースを**同時に**必要とします。

  - **コピー操作**: 解答の例は典型的です。ソース（テープ）とデスティネーション（プリンタ）を同時に保持できなければ、データをチャンクごとに読み書きするしかなく、非常に非効率です。また、他のプロセスが間に割り込むことで、出力が断片的になる可能性があります。
  - **データベースのトランザクション**: 2つの口座間での送金処理では、両方の口座レコードを同時にロックする必要があります。片方だけロックして処理を進めることはできません。
  - **コンパイラ**: ソースファイルを読み込み、オブジェクトファイルを書き出すコンパイラは、少なくとも2つのファイルを同時に開いておく必要があります。 このように、複数のリソースを同時に保持できないという制約は、多くの現実的なアプリケーションにとって非現実的です。

28. 2つのプロセスAとBが、…デッドロックフリーであることが保証される割合はどれくらいですか。

- **[解答訳]** プロセスAがレコードをa, b, cの順で要求すると仮定します。もしプロセスBもaを最初に要求すれば、どちらかがそれを取得し、もう一方はブロックされます。この状況は常にデッドロックフリーです。残りの4つの組み合わせのうち、いくつかはデッドロックにつながる可能性があり、いくつかはデッドロックフリーです。6つのケースは以下の通りです： a b c : デッドロックフリー a c b : デッドロックフリー b a c : デッドロックの可能性あり b c a : デッドロックの可能性あり c a b : デッドロックの可能性あり c b a : デッドロックの可能性あり 6つのうち4つがデッドロックにつながる可能性があるため、デッドロックを回避する確率は1/3、デッドロックになる確率は2/3です。

- **[より詳しい解説]** この問題は、**循環待ち** の発生確率を組み合わせ論的に考えるものです。

    - 全事象: Bが3つのリソースを要求する順序は 3! = 6 通り。
    - デッドロックフリーが保証される場合:
        - **リソース要求順序の統一**: BがAと同じ順序 (1，2，3) で要求する場合。この場合、両者がリソース1で競合し、勝者が全てのリソースを取得して完了するため、デッドロックは発生しません。
        - **Aの要求順序が 1，2，3 のとき**:
            - Bの要求順序が (1，2，3) または (1，3，2) の場合、最初に1で競合するためデッドロックフリー。
            - Bの要求順序が (2，...) または (3，...) の場合、Aが1を取得し、Bが2または3を取得する可能性があります。その後、Aが2を要求し、Bが1を要求するといった循環待ちが発生する可能性があります。
    - **分析**:
        - Bの順序 (1, 2, 3): デッドロックフリー
        - Bの順序 (1, 3, 2): デッドロックフリー
        - Bの順序 (2, 1, 3): Aが1、Bが2を取得→Aが2を要求(待ち)、Bが1を要求(待ち)→デッドロックの可能性あり
        - Bの順序 (2, 3, 1): Aが1、Bが2を取得→Aが2を要求(待ち)、Bが3を取得→Bが1を要求(待ち)→デッドロックの可能性あり
        - Bの順序 (3, 1, 2): Aが1、Bが3を取得→Aが2を取得→Bが1を要求(待ち)、Aが3を要求(待ち)→デッドロックの可能性あり
        - Bの順序 (3, 2, 1): Aが1、Bが3を取得→Aが2を要求(待ち)、Bが2を取得→Bが1を要求(待ち)→デッドロックの可能性あり したがって、6通りのうち2通りがデッドロックフリーなので、その割合は 2/6 = 1/3 となります。

29. メールボックスを使用する分散システムには、sendとreceiveという2つのIPCプリミティブがあります。…デッドロックは可能ですか。議論しなさい。

- **[解答訳]** はい。全てのメールボックスが空だと仮定します。ここで、AがBに送信して返信を待ち、BがCに送信して返信を待ち、CがAに送信して返信を待つとします。全てのデッドロックの条件が満たされています。

- **[より詳しい解説]** これはリソースデッドロックではなく、**通信デッドロック** の例です。リソース（メールボックス）は共有されていますが、問題はリソースの保持ではなく、プロセスの通信パターンにあります。

    - **プロセスA**: Bからのメッセージを `receive` してブロック。
    - **プロセスB**: Cからのメッセージを `receive` してブロック。
    - **プロセスC**: Aからのメッセージを `receive` してブロック。 この状況は、デッドロックの4条件に当てはまります。

    1. **相互排他**: 一つのプロセスしかメッセージを受信できません（これは少しこじつけですが、メッセージをイベントと考えると成り立ちます）。
    2. **保持と待機**: 各プロセスは「返信を待つ」という状態を保持しつつ、他のプロセスからの「メッセージ」というイベントを待っています。
    3. **横取りなし**: プロセスを強制的に `receive` 待ちから解放することはできません。

4. **循環待ち**: AはBを待ち、BはCを待ち、CはAを待つという循環が形成されています。 この種のデッドロックは、通信プロトコルに**タイムアウト**を導入することで解決できます。一定時間待ってもメッセージが来ない場合、`receive` はエラーを返してブロックを解除します。

30. 電子資金移動システムには、…デッドロックを回避する仕組みを考案しなさい。…

- **[解答訳]** 循環待ちを回避するために、リソース（口座）を口座番号で順序付けます。入力行を読み取った後、プロセスはまず番号の小さい方の口座をロックし、次に（待つ可能性はあるが）ロックを取得したら、もう一方の口座をロックします。どのプロセスも、既に保持している口座よりも番号の小さい口座を待つことは決してないため、循環待ちはなくなり、デッドロックも発生しません。

- **[より詳しい解説]** これは、デッドロック予防策の一つである「**循環待ちの打破 (Attacking the Circular Wait Condition)**」 の典型的な適用例です。

    - **問題**: プロセス1が口座Xをロックし口座Yを待つ一方で、プロセス2が口座Yをロックし口座Xを待つ、という状況がデッドロックを引き起こします。
    - **解決策**: 全てのリソース（この場合は銀行口座）に**一意の順序**を付けます（口座番号は自然な順序付けです）。そして、全てのプロセスに「**必ず番号の昇順でリソースを要求する**」というルールを強制します。
    - **なぜ機能するか**:
        - プロセス1が口座100と200を扱う場合、必ず100→200の順でロックします。
        - プロセス2が口座100と200を扱う場合も、必ず100→200の順でロックします。
        - もしプロセス1が100を、プロセス2が300をロックしたとします。プロセス1は次に200を要求し、プロセス2は次に400を要求するかもしれません。しかし、プロセス2が100を要求することはありません。なぜなら、300を保持している状態で、それより番号の小さい100を要求することはルール違反だからです。 このように、リソース要求の順序を固定することで、循環待ちは構造的に発生しなくなり、デッドロックは予防されます。

31. デッドロックを防ぐ一つの方法は、「保持と待機」の条件をなくすことです。…この仕組みに対する改善案を提案しなさい。

- **[解答訳]** 新しいリソースを要求する際のセマンティクスを次のように変更します。もしプロセスが新しいリソースを要求し、それが利用可能であれば、そのリソースを取得し、既に保持しているものも維持します。もし新しいリソースが利用不可であれば、既存のリソースは全て解放されます。このシナリオでは、デッドロックは不可能であり、新しいリソースは取得されても既存のリソースが競合するプロセスに失われる危険もありません。もちろん、このプロセスが機能するのは、リソースの解放が可能な場合に限られます（ページ間のスキャナやCD間のCDレコーダーなど）。

- **[より詳しい解説]** これは、「**保持と待機 (Hold and Wait)**」条件を打破するための、より洗練されたアプローチです。

    - **元の提案の問題点**: 要求したリソースが利用不可だった場合に、**無条件に**保持しているリソースを全て解放すると、それらを他のプロセスに取られてしまう可能性があります。次に再試行したとき、元々持っていたリソースすら確保できないかもしれず、**飢餓状態 (starvation)** に陥る可能性があります。
    - **改善案のロジック**:
        1. プロセスPがリソースR1を保持中に、リソースR2を要求する。
        2. **まず、R2が利用可能かチェックする**。
        3. もしR2が**利用可能**なら、PにR2を割り当て、PはR1とR2の両方を保持して処理を続ける。
        4. もしR2が**利用不可**なら、PにR2を割り当てず、**かつ、Pが保持しているR1も解放させる**。その後、Pは再度R1とR2の両方を要求しなおす必要がある。 この改善案は、「全てのリソースを一度に要求する」という厳しい制約を緩和しつつ、飢餓状態のリスクを減らしています。ただし、一度リソースを解放すると、再度確保するまで待たなければならないという問題は残ります。

32. デッドロックに取り組むよう割り当てられた学生が、…時間制限を超えると、プロセスは解放され、再び実行が

許可されます。もしあなたが教授なら、この提案にどのような評価を与え、その理由は何ですか。

- **[解答訳]** 私はF（不可）の評価を与えます。プロセスは何をするでしょうか？明らかにそのリソースを必要としているので、再度要求し、再びブロックされるでしょう。これはブロックされ続けるのと何ら変わりません。実際、システムは競合するプロセスの待ち時間を記録し、最も長く待っているプロセスに新たに解放されたリソースを割り当てるかもしれないので、状況はさらに悪化する可能性があります。定期的にタイムアウトして再試行することで、プロセスはその優先順位を失います。

- **[より詳しい解説]** この学生の提案は、一見するとデッドロックを解消するように見えますが、根本的な問題を解決していません。

    - **問題の先送り**: プロセスがブロックされるのは、必要なリソースが利用できないからです。タイムアウトでブロックを解除しても、リソースの状況は変わっていません。プロセスは処理を続けられないため、結局すぐに同じリソースを要求して再びブロックされるだけです。これは**ライブロック (Livelock)** に近い状態を引き起こします。プロセスはCPU時間を消費しますが、全く進展しません。
    - **公平性の喪失**: 解答が指摘するように、多くのスケジューリングアルゴリズムやリソース割り当てポリシーは、待ち時間が長いプロセスを優先します。タイムアウトして要求をやり直すことは、この「待ち時間」をリセットしてしまい、プロセスがいつまでたってもリソースを得られない**飢餓状態 (starvation)** に陥る原因となります。 この提案はデッドロックの根本原因である「リソースの循環待ち」を解決しておらず、別の問題（ライブロックや飢餓状態）を引き起こすため、不適切な解決策です。

33. 主メモリユニットは、スワッピングや仮想メモリシステムで横取りされます。プロセッサは、タイムシェアリング環境で横取りされます。これらの横取り方法は、リソースデッドロックを処理するために開発されたと思いますか、それとも他の目的のためですか。それらのオーバーヘッドはどの程度高いですか。

- **[解答訳]** どちらも主にシステムユーザーを支援するために開発されました。ハードウェアを仮想化することで、ユーザーはニーズの事前申告、リソース割り当て、オーバーレイといった詳細から解放され、さらにデッドロックも防止されます。しかし、コンテキストスイッチングと割り込み処理のコストは相当なものです。特殊なレジスタ、キャッシュ、回路が必要です。おそらく、デッドロック防止という目的のためだけでは、このコストは支払われなかったでしょう。

- **[より詳しい解説]** この問題は、OSの機能の主目的とその副次的な効果を区別させるものです。

    - **主目的**:
        - **プロセッサの横取り（プリエンプティブ・スケジューリング）**: 主目的は、**対話的なシステムの応答性を高め、複数のユーザーやプロセスに公平にCPU時間を分配する**ことです。
        - **主メモリの横取り（スワッピング、ページング）**: 主目的は、**物理メモリのサイズを超える大きさのプログラムや、合計サイズが物理メモリを超える複数のプログラムを同時に実行可能にする**ことです。
    - **デッドロックへの効果**: これらの横取り（プリエンプション）機能は、デッドロックの4条件の一つである「**横取りなし (No Preemption)**」 を破ることができます。メモリやCPUがデッドロックの原因となっている場合、OSがそれらを横取りすることでデッドロックを解消できます。しかし、これはあくまで**副次的な効果**であり、これらの機能が開発された主目的ではありません。
    - **オーバーヘッド**:
        - プロセッサの横取り（コンテキストスイッチ）は、レジスタの保存・復元、キャッシュのフラッシュ、TLBの無効化などを伴い、数マイクロ秒のオーバーヘッドがあります。
        - メモリの横取り（ページフォールト）は、ディスクアクセスを伴うため、数ミリ秒の非常に高いオーバーヘッドがあります。 これらの高いオーバーヘッドは、対話性向上やメモリの有効活用という主目的のために支払われているものであり、デッドロック防止のためだけでは正当化できません。

34. デッドロック、ライブロック、飢餓状態（スターベーション）の違いを説明しなさい。

- **[解答訳]** デッドロックは、あるプロセスの集合が、その集合内の他のプロセスしか引き起こせないイベントを待ってブロックされている状態です。一方、ライブロック状態にあるプロセスはブロックされていません。代わりに、それらは継続的に実行され、決して真になることのない条件をチェックします。したがって、それらが保持しているリソースに加えて、ライブロック状態のプロセスは貴重なCPU時間を消費し続けます。最後に、プロセスの飢餓状態は、他のプロセスの存在や、その飢餓状態のプロセスよりも高い優先度を持つ新しいプロセスの流入によって発生します。デッドロックやライブロックとは異なり、飢餓状態は自然に終了することがあります。例えば、より高い優先度のプロセスが終了し、飢餓状態のプロセスよりも高い優先度を持つ新しいプロセスが到着しない場合などです。

- **[より詳しい解説]** これらの概念は、プロセスの進行が妨げられるという点で似ていますが、その原因とプロセスの状態が異なります。

    - **デッドロック (Deadlock)**:
        - **プロセスの状態**: **ブロック状態** (待機状態) であり、CPUを消費しません。
        - **原因**: プロセス群が互いのリソースを待ち合う**循環待ち**が形成されている。
        - **回復**: OSによるプロセスの強制終了など、**外部からの介入がなければ自然に解消されません**。
    - **ライブロック (Livelock)**:
        - **プロセスの状態**: **実行状態** にあり、CPUを消費し続けますが、**有用な処理は全く進展しません**。
        - **原因**: プロセス群が互いの状態変化に過剰に反応し、状態を繰り返し変更し続けるため、全体として進展がなくなります。例えば、2人が狭い廊下で互いに道を譲ろうとして、同じ方向にステップを繰り返し続ける状況です。
        - **回復**: デッドロックと同様、自然解消は稀です。
    - **飢餓状態 (Starvation)**:
        - **プロセスの状態**: **準備完了状態 (Ready)** にあるかもしれませんが、スケジューラによって長期間（あるいは無期限に）実行機会を与えられません。
        - **原因**: **不公平なスケジューリングポリシー**。例えば、優先度の高いプロセスが常に存在するため、優先度の低いプロセスが実行されない、などです。
        - **回復**: 状況によっては**自然に解消される可能性があります**（高優先度プロセスが終了するなど）。

35. 2つのプロセスが、ディスクにアクセスするためのメカニズムを再配置し…このシーケンスが絶えず繰り返されます。これはリソースデッドロックですか、それともライブロックですか。この異常を処理するためにどのような方法を推奨しますか。

- **[解答訳]** このデッド状態は競合同期の異常であり、リソースの事前割り当てによって制御できます。しかし、プロセスはリソースに対してブロックされていません。加えて、リソースは既に線形順序で要求されています。この異常はリソースデッドロックではなく、**ライブロック**です。リソースの事前割り当てがこの異常を防ぎます。ヒューリスティックとしては、プロセスはタイムアウトしてリソースを解放し、一定時間スリープしてから再試行することができます。

- **[より詳しい解説]**

    - **ライブロックである理由**: プロセスはブロックされていません。各プロセスはCPUを割り当てられてシークコマンドを再発行するという**処理を実行**しています。しかし、相手のプロセスによってその処理が無駄にされるため、**進展がありません**。これはライブロックの典型的な特徴です。
    - **リソースデッドロックではない理由**: プロセスはリソースを待ってブロックされているわけではなく、処理をやり直している状態です。
    - **解決策**:
        1. **ロックの導入（相互排他）**: ディスクアームのシークから読み取りまでの一連の操作を**クリティカルセクション**とし、ミューテックスなどで保護します。これにより、一度アームを動かしたプロセスは、必ず読み取りを完了するまで他のプロセスに邪魔されません。
        2. **リソースの事前割り当て**: ディスクアームとデータチャネルの両方を一度に獲得しない限り、処理を開始できないようにします。これが解答の言う「リソースの事前割り当て」です。

36. ローカルエリアネットワークは、CSMA/CDと呼ばれるメディアアクセス方式を利用します…

- **[解答訳]**
  - **(a)** これは競合同期の異常であり、**ライブロック**です。リソースライブロックでもデッドロックでもありません。なぜなら、ステーションは他のステーションが要求するリソースを保持しておらず、したがってリソースを保持しながら他を待つ循環的な連鎖は存在しないからです。これは通信デッドロックでもありません。なぜなら、ステーションは独立して実行されており、順次スケジュールされれば送信を完了するからです。
  - **(b)** イーサネットとスロット付きALOHAは、衝突を検出したステーションが、再送する前にランダムな数のタイムスロットを待つことを要求します。タイムスロットが選択される間隔は、連続する衝突のたびに倍になります。
  - **(c)** チャネルへのアクセスは確率的であり、新しく到着したステーションが、既に何回か再送したステーションよりも先にチャネルを獲得して競合する可能性があるため、**飢餓状態は発生し得ます**。
- **[より詳しい解説]**
  - **(a) ライブロック**: 各ステーション（プロセス）は、送信→衝突検出→待機→再送信、という一連の動作を繰り返しています。ブロックはしていませんが、進展がありません。これはライブロックです。
  - **(b) 解決策**: 解答が示す「ランダムな時間待機し、衝突のたびに待機時間間隔を倍にする」アルゴリズムは、**バイナリ指数バックオフ (binary exponential backoff)** と呼ばれます。これにより、衝突したステーションが次の再送で再び衝突する確率を劇的に下げることができます。これはイーサネットで実際に採用されている方式です。
  - **(c) 飢餓状態**: バイナリ指数バックオフは衝突の確率を下げるだけで、完全に無くすわけではありません。運の悪いステーションが何度も衝突を繰り返し、待機時間が非常に長くなっている間に、新しく送信したいステーションが容易にチャネルを獲得してしまう可能性があります。これにより、特定のステーションがいつまでも送信機会を得られない**飢餓状態**が発生する可能性があります。

37. あるプログラムには、協調と競合のメカニズムの順序に誤りがあり、…これはリソースデッドロックですか、それとも通信デッドロックですか。その制御方法を提案しなさい。

- **[解答訳]** この異常はリソースデッドロックではありません。プロセスはミューテックスという競合メカニズムを共有していますが、リソースの事前割り当てやデッドロック回避手法はこのデッド状態には効果がありません。…両方のプロセスが、他方しか引き起こせないイベントを待ってブロックされるという循環的なデッド状態は存在します。これは**通信デッドロック**です。このデッドロックを解消するには、タイムアウトがコンシューマのミューテックスを横取りすれば機能します。より良い解決策は、注意深くコードを書くか、相互排他のためにモニタを使用することです。

- **[より詳しい解説]**

  - **通信デッドロック**: この問題の核心は、プロセスの**協調**（プロデューサがデータを生産し、コンシューマに通知する）と**競合**（共有バッファへのアクセス）の同期メカニズムの**順序間違い**です。
    - コンシューマ: ミューテックスをロック → バッファが空なので**シグナルを待って**ブロック
    - プロデューサー: ミューテックスを要求 → コンシューマが保持しているのでブロック → **シグナルを送れない** コンシューマはプロデューサーからのシグナル（イベント）を待ち、プロデューサーはコンシューマがミューテックスを解放する（イベント）のを待つという循環が発生しています。これはリソースの枯渇ではなく通信の不成立によるデッドロックなので、**通信デッドロック** に分類されます。
  - **制御方法**:
    1. **プログラミングのバグを修正**: 最も正しい解決策です。コンシューマは、バッファが空でないことを確認した**後**にミューテックスをロックするように、順序を正しく修正します。
    2. **モニタの使用**: モニタ を使えば、この問題は構造的に解決できます。モニタでは、プロセスがモニタ内部で条件変数（バッファが空など）を待つ場合、**モニタのロック（ミューテックス）を自動的に解放します**。これにより、プロデューサーがモニタに入ってデータを生産し、コンシューマにシグナルを送ることが可能になります。

38. シンデレラと王子様が離婚することになりました。…コンピュータはまだ交渉中です。なぜですか。デッドロックは可能ですか。飢餓状態は可能ですか。あなたの答えを議論しなさい。

- **[解答訳]** もし両方のプログラムが最初にウーファーを要求した場合、コンピュータは「ウーファーを要求、要求をキャンセル、ウーファーを要求、要求をキャンセル」という無限のシーケンスで**飢餓状態**に陥るでしょう。もし一方が犬小屋を要求し、他方が犬を要求した場合、**デッドロック**が発生します。これは両者によって検出され、解消されますが、次のサイクルで再び繰り返されるだけです。いずれにせよ、両方のコンピュータが犬か犬小屋を最初に狙うようにプログラムされている場合、飢餓状態かデッドロックのいずれかが発生します。ここでは両者に実質的な違いはほとんどありません。ほとんどのデッドロック問題では、ランダムな遅延を導入することで飢餓状態の可能性は非常に低くなるため、深刻とは見なされません。そのアプローチはここでは機能しません。

- **[より詳しい解説]** この問題は、デッドロック、ライブロック、飢餓状態の微妙な違いを探るものです。

    - **デッドロック**: シンデレラがウーファーを、王子が犬小屋を要求し、それぞれが取得したとします。次の日、シンデレラは犬小屋を、王子はウーファーを要求します。ここで**循環待ち**が発生し、デッドロックとなります。プロトコルによれば、彼らはこのデッドロックを検出し、要求をキャンセルして回復します。しかし、次のサイクルで同じ行動を繰り返す可能性があるため、問題は解決しません。
    - **飢餓状態 (Starvation) / ライブロック (Livelock)**: 両者が同時にウーファーを要求した場合、プロトコルに従い両者とも要求をキャンセルします。そして次の日、また両者ともウーファーを要求し、またキャンセル…というサイクルを繰り返します。プロセスはブロックされていませんが（処理は行われている）、全く進展がありません。これは**ライブロック**です。また、一方が常に譲歩し続ける場合、**飢餓状態**になる可能性もあります。このシナリオでは、単純なランダム遅延では解決しません。なぜなら、両者の「ウーファーが欲しい」という優先度が非常に高いため、ランダムなタイミングで同じ要求を繰り返してしまうからです。解決策としては、要求が衝突した場合に、どちらかが優先権を得るような非対称なルール（例：名前のアルファベット順）を導入する必要があります。

39. 人類学を専攻する学生が、ヒヒにデッドロックについて教えるプロジェクトに着手しました。…セマフォを使用してデッドロックを回避するプログラムを書きなさい。飢餓状態については心配しなくてよいです。

- **[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、読者・書き込み問題の変種です。ここでのデッドロックは、東向きと西向きのヒヒが同時にロープに乗ることで発生します。これを防ぐには、ロープへのアクセスを相互排他にする必要がありますが、同じ方向のヒヒは同時に渡れるようにすべきです。飢餓を考慮しない最も簡単な解決策は以下の通りです。

    1. **必要な変数**:
        - `mutex`: ロープの状態を変更する際の相互排他のためのセマフォ（初期値1）。
        - `east_count`, `west_count`: それぞれ東向き、西向きに渡っているヒヒの数を記録するカウンタ。
        - `east_sem`, `west_sem`: それぞれ東向き、西向きに渡りたいヒヒをブロックさせるためのセマフォ（初期値0）。
    2. **東向きのヒヒのロジック (`cross_east`)**:
        - `down(&mutex)` でロック。
        - もし `west_count > 0` なら（反対方向のヒヒがいるなら）、`up(&mutex)` して `down(&east_sem)` で待機。待機から復帰したら再度 `down(&mutex)`。
        - `east_count++`。
        - `up(&mutex)` でアンロック。
        - *<ロープを渡る>*
        - `down(&mutex)` でロック。
        - `east_count--`。
        - もし `east_count == 0` なら、待機している西向きのヒヒを起こすために `up(&west_sem)` を複数回実行。

    - up(&mutex) でアンロック。 西向きのヒヒも対称的なロジックで実装します。これは、ある方向にヒヒが渡っている間、反対方向のヒヒは待たされ続けるため、飢餓状態が発生する可能性があります。

40. 前の問題を繰り返しますが、今度は飢餓状態を回避してください。…

- **[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]** 飢餓状態を回避するには、一方の方向のヒヒがロープを独占し続けるのを防ぐ必要があります。これは、ロープを待っている反対方向のヒヒがいる場合に、新たに同じ方向のヒヒがロープに入るのを禁止することで実現できます。

    1. **追加の変数**:
        - east_waiting, west_waiting: それぞれ待っているヒヒの数を記録するカウンタ。
    2. **東向きのヒヒのロジックの変更点**:
        - ロープを渡る前 (down(&mutex) の後): east_waiting++。もし west_count > 0 または west_waiting > 0 なら待機。待機後、east_waiting--。
        - ロープを渡り終えた後 (down(&mutex) の後): east_count--。もし east_count == 0 かつ west_waiting > 0 なら、西向きのヒヒを起こす。 このロジックにより、例えば東向きのヒヒが渡っている最中に西向きのヒヒが待機を始めると (west_waiting > 0)、それ以上新しい東向きのヒヒはロープに入れなくなります。現在渡っている東向きのヒヒがすべて渡り終えたら、待っていた西向きのヒヒが渡るチャンスを得られます。

41. 銀行家のアルゴリズムのシミュレーションをプログラムしなさい。…

- **[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]** 銀行家のアルゴリズムのシミュレーションを実装するには、以下の要素が必要です。

    1. **データ構造**:
        - n (プロセス数) と m (リソースクラス数)。
        - Available: 利用可能な各リソースのインスタンス数を示す m次元ベクトル。
        - Max: 各プロセスが必要とする各リソースの最大数を示す n x m 行列。
        - Allocation: 各プロセスに現在割り当てられているリソース数を示す n x m 行列。
        - Need: 各プロセスが今後必要とするリソース数を示す n x m 行列 (Need = Max - Allocation)。
    2. **安全性チェックアルゴリズム**:
        - Work = Available, Finish[i] = false で初期化。
        - Finish[i] == false かつ Need[i] <= Work を満たすプロセス i を探す。
        - 見つからなければ、全ての Finish[i] が true なら安全、そうでなければ危険、として終了。
        - 見つかれば、Work = Work + Allocation[i], Finish[i] = true として、前のステップに戻る。
    3. **リソース要求アルゴリズム**:
        - プロセス i から Request[i] の要求があった場合、まず Request[i] <= Need[i] かつ Request[i] <= Available かをチェック。
        - 満たされなければエラー。満たされていれば、リソースを仮に割り当ててみる (Available -= Request[i], Allocation[i] += Request[i], Need[i] -= Request[i])。
        - この仮の状態が安全かどうかを安全性チェックアルゴリズムで確認する。
        - 安全なら要求を許可し、危険なら要求を拒否して状態を元に戻す。

42. 各タイプのリソースが複数ある場合のデッドロック検出アルゴリズムを実装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]** これは、教科書 Section 6.4.2 で説明されているデッドロック検出アルゴリズムの実装です。銀行家のアルゴリズムの安全性チェックと非常によく似ています。

1. **データ構造**:
    - E: 存在する各リソースの総数を示すベクトル。
    - A: 利用可能なリソース数を示すベクトル。
    - C: 現在の割り当てを示す行列。
    - R: 現在の要求を示す行列。
2. **アルゴリズム**:
    - `Work = A`, `Finish[i] = false` で初期化 (ただし、`Allocation[i]`が全て0のプロセスは `true`)。
    - `Finish[i] == false` かつ `Request[i] <= Work` を満たすプロセス `i` を探す。
    - 見つからなければ、`Finish` が `false` のプロセスが少なくとも1つ残っていれば、それらのプロセスがデッドロック状態にあるとして終了。
    - 見つかれば、そのプロセスが完了できると仮定し、`Work = Work + C[i]`, `Finish[i] = true` として、前のステップに戻る。 プログラムは、入力ファイルから `E`, `C`, `R` を読み込み、`A = E - (Cの各列の合計)` を計算し、上記アルゴリズムを実行して、最後に `Finish` が `false` であるプロセスのIDをリストアップします。

43. リソース割り当てグラフを使用してシステムにデッドロックがあるかどうかを検出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]** これは、教科書 Section 6.4.1 で説明されている、各リソースのインスタンスが1つしかない場合のデッドロック検出アルゴリズムの実装です。

    1. **グラフの構築**:
        - プロセスとリソースをノードとしてグラフデータ構造（例：隣接リスト）を構築します。
        - 入力に基づき、プロセスとリソース間に有向エッジを追加します。
            - プロセス `P` がリソース `R` を保持している場合: `R -> P` のエッジ。
            - プロセス `P` がリソース `R` を要求している場合: `P -> R` のエッジ。
    2. **サイクル検出**:
        - 構築したグラフにサイクルが存在するかどうかを検出します。最も一般的な方法は**深さ優先探索 (DFS)** です。
        - DFSの実行中、訪問中のノードの集合（`visiting` set）を管理します。もし、現在探索中のパスで `visiting` set に含まれるノードに再び到達した場合、サイクルが検出されたことになります。
        - 一度探索が完了したノードは `visited` set に移し、再探索を避けます。
    3. **出力**:
        - サイクルが検出された場合、システムはデッドロック状態にあります。サイクルを構成するプロセスを特定して出力します。サイクルがなければ、デッドロックはありません。

44. ある国では、2人が会うとお互いにお辞儀をします。…デッドロックしない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]** この問題は、2つのプロセス間のデッドロックまたはライブロックを回避する古典的な同期問題です。デッドロックは、両者が同時にお辞儀をして、相手がお辞儀をやめるのを永遠に待ち続けることで発生します。 この問題を解決する鍵は、**対称性を破る**ことです。もし両者が同じプロトコルに従うなら、デッドロックの可能性があります。非対称なプロトコルを導入すれば回避できます。

    - **解決策**:
        1. **優先順位付け**: 2人の人（プロセス）に一意のID（例えば、名前のアルファベット順や身長順など）を割り当てます。
        2. **プロトコル**:
            - IDが低い方が先にお辞儀をする。
            - IDが高い方は、相手がお辞儀をするのを待ってからお辞儀をする。 このルールにより、どちら

か一方が必ず先に行動するため、両者が同時にお辞儀をして待ち続けるという**循環待ち**の状態は発生しません。これは、デッドロック予防における「**循環待ちの打破**」アプローチ（リソースの順序付け）に相当します。