オペレーティングシステム

第7回

デッドロック

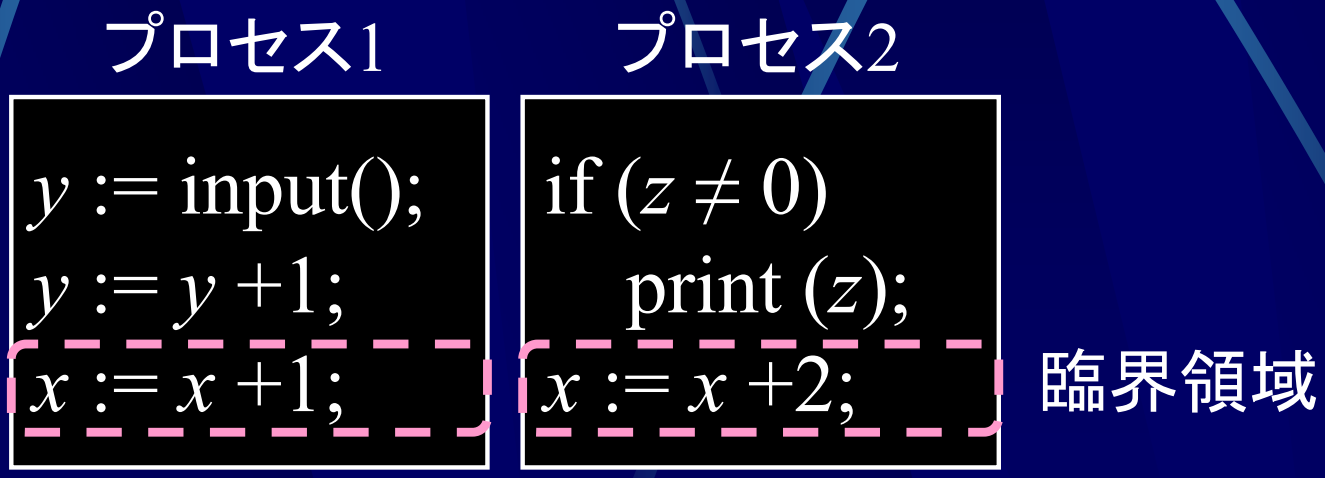
http://www.info.kindai.ac.jp/OS

E号館3階E-331 内線5459

takasi-i@info.kindai.ac.jp

臨界領域
(critical section, critical region)

- 臨界領域(critical section, critical region)
  - 逐次的資源を使用しているプロセスの部分



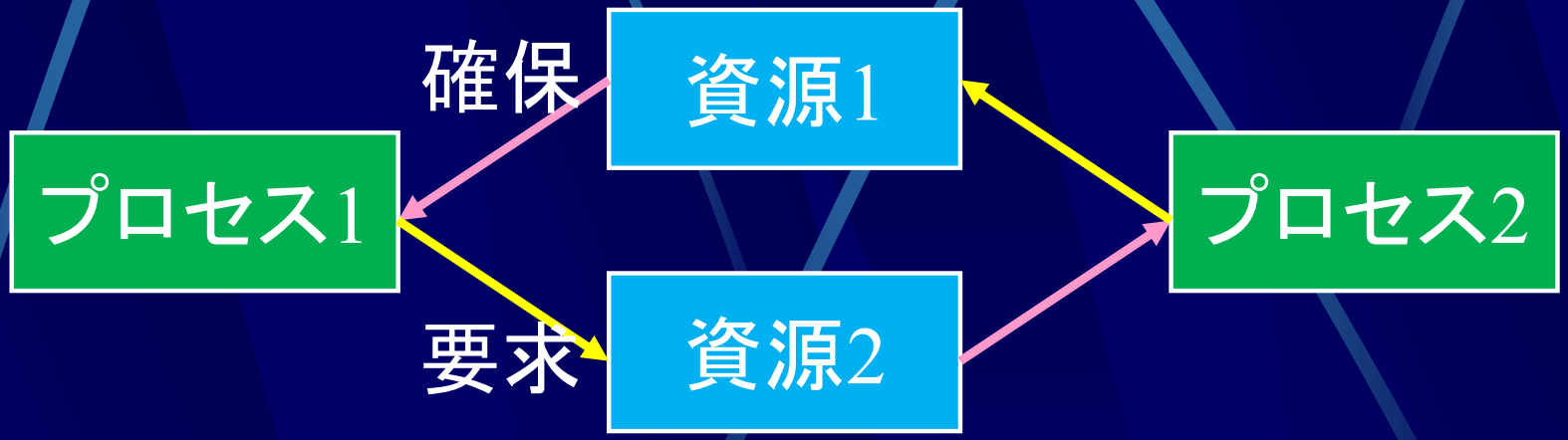
臨界領域に入るときは
他のプロセスが逐次的資源を使わないように
資源を占有する必要がある

相互排除, 排他制御
(mutual exclusion, exclusive control)

- 相互排除(mutual exclusion),
  排他制御(exclusive control)
  - ある資源を高々1つのプロセスが占有するようにする
  - あるプロセスが資源を使用しているときは、他のプロセスは資源が解放されるまで待つ

資源確保の競合

プロセス1,2共に資源1,2が必要



資源2が確保されるまで
ブロック状態

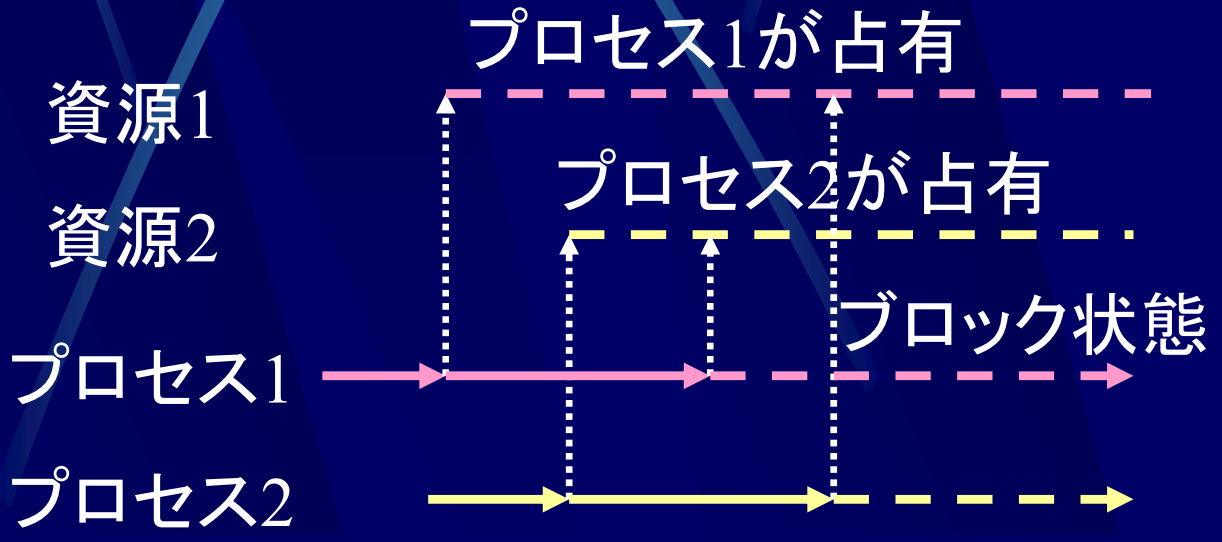
資源1が確保されるまで
ブロック状態

どちらのプロセスも永久に資源を確保できない

デッドロック(deadlock)

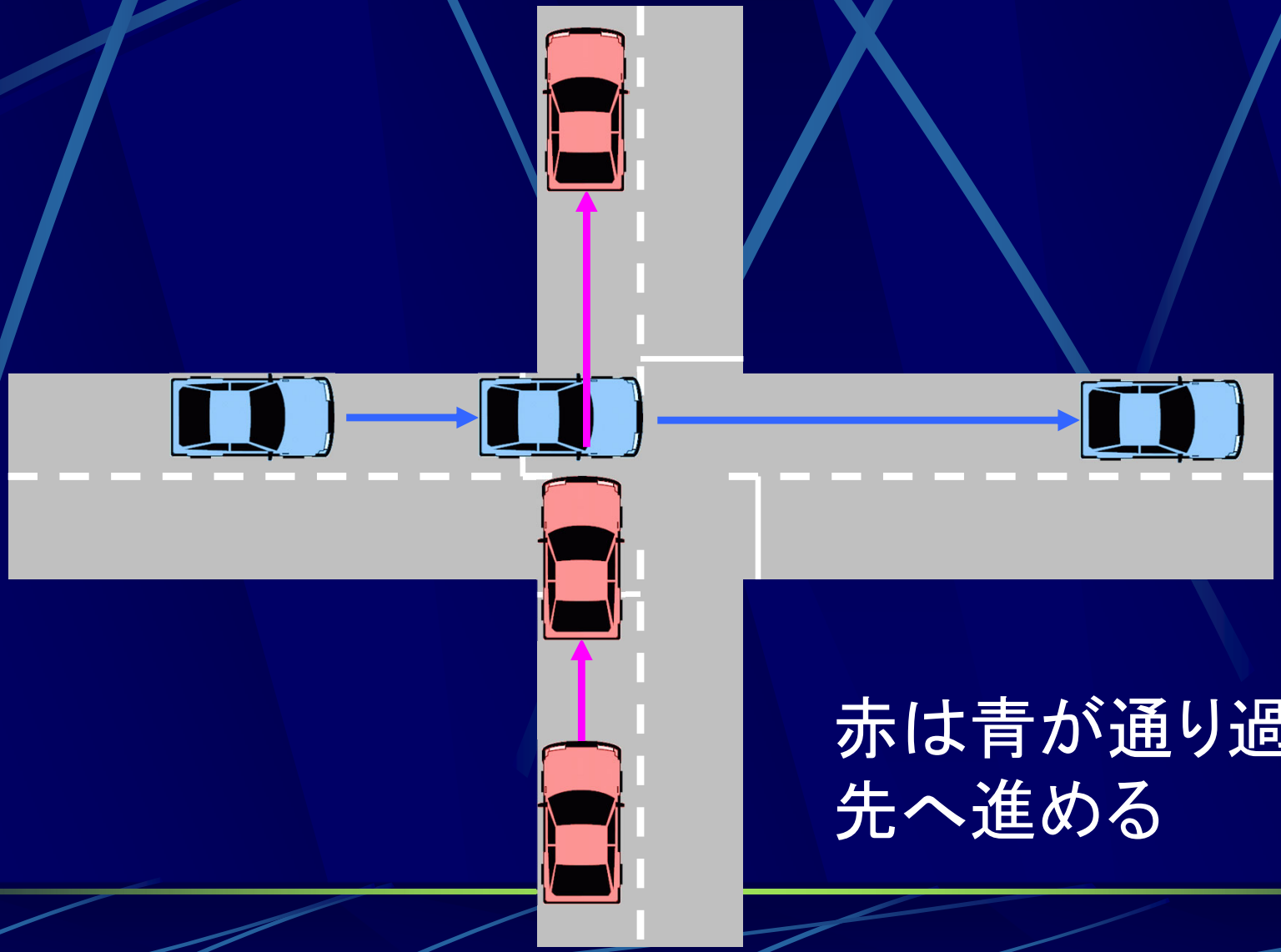
デッドロック, 死の抱擁
(deadlock, deadly embrace)

- デッドロック(deadlock), 死の抱擁(deadly embrace)
  - 複数のプロセスが資源を占有しているために互いにブロックされてしまう状態



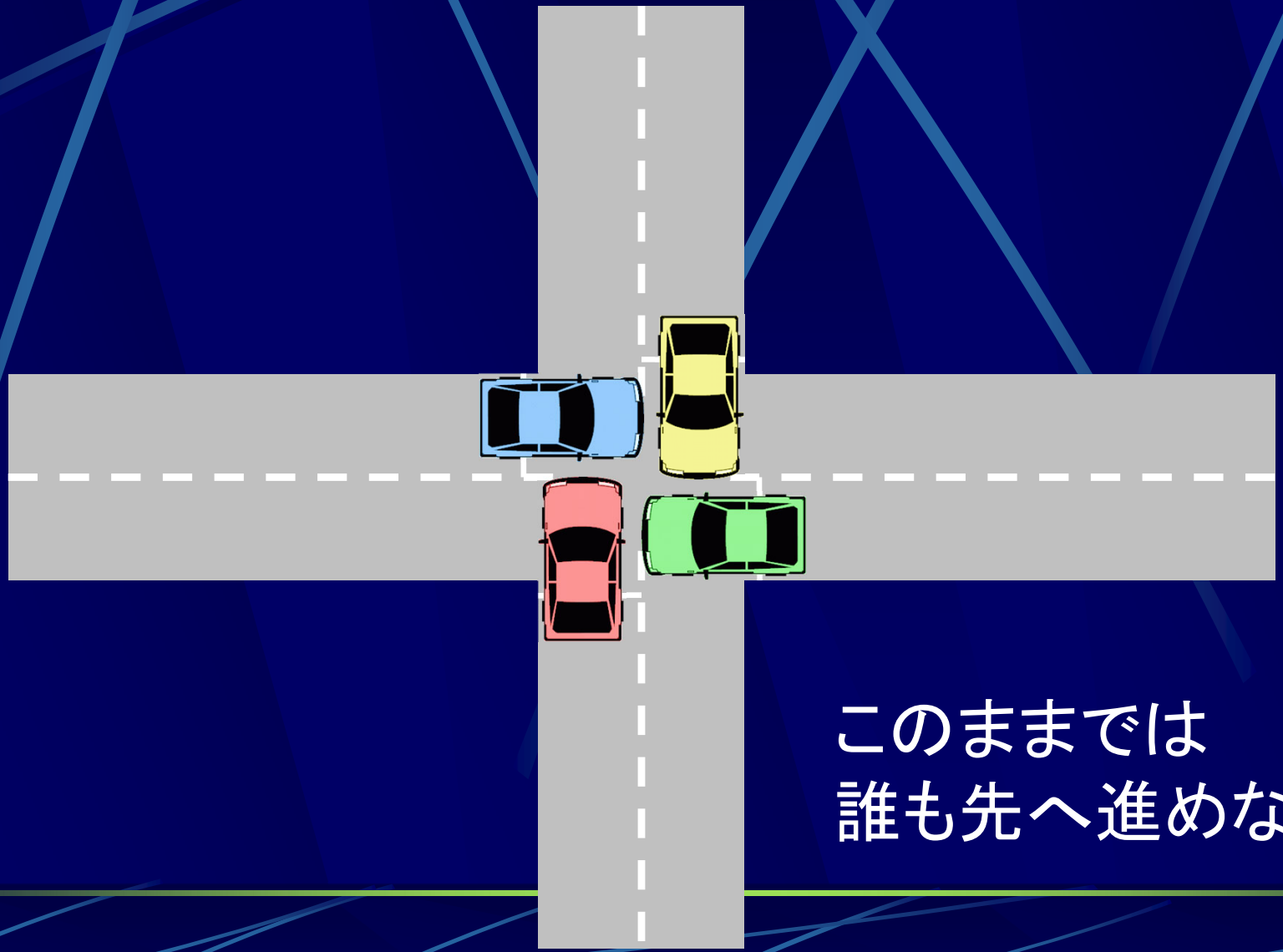
プロセス1,プロセス2共に
ブロック状態のまま先へ進めない

デッドロック



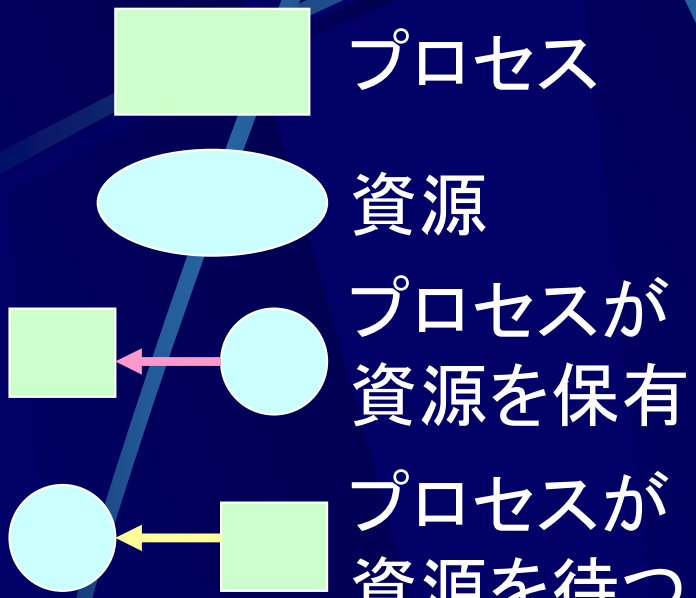
赤は青が通り過ぎれば
先へ進める

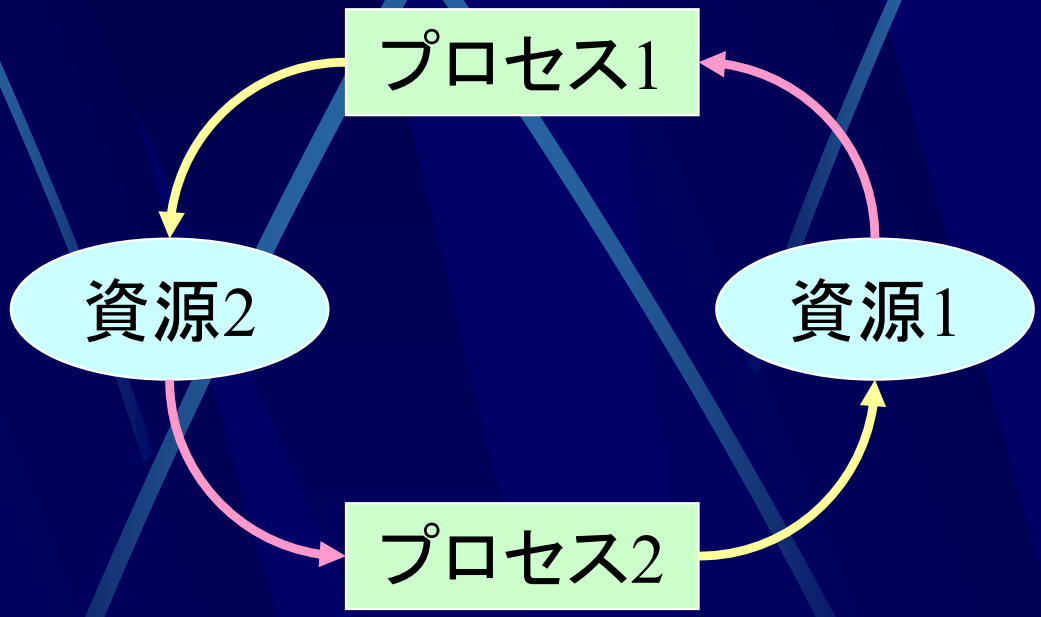
デッドロック



このままでは
誰も先へ進めない

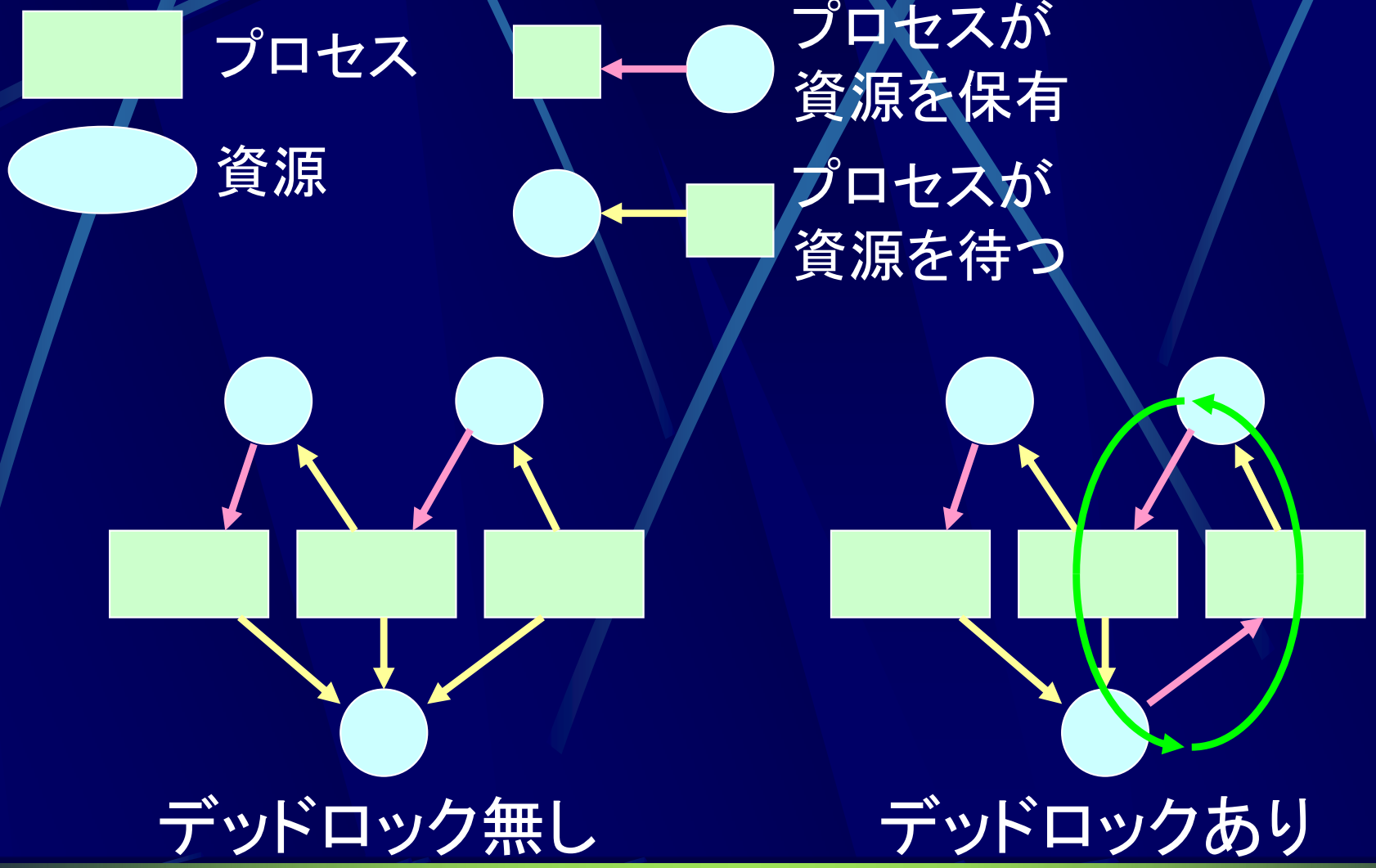
デッドロック発生の原理





プロセス-資源の間で閉じた
ループができるとデッドロック発生

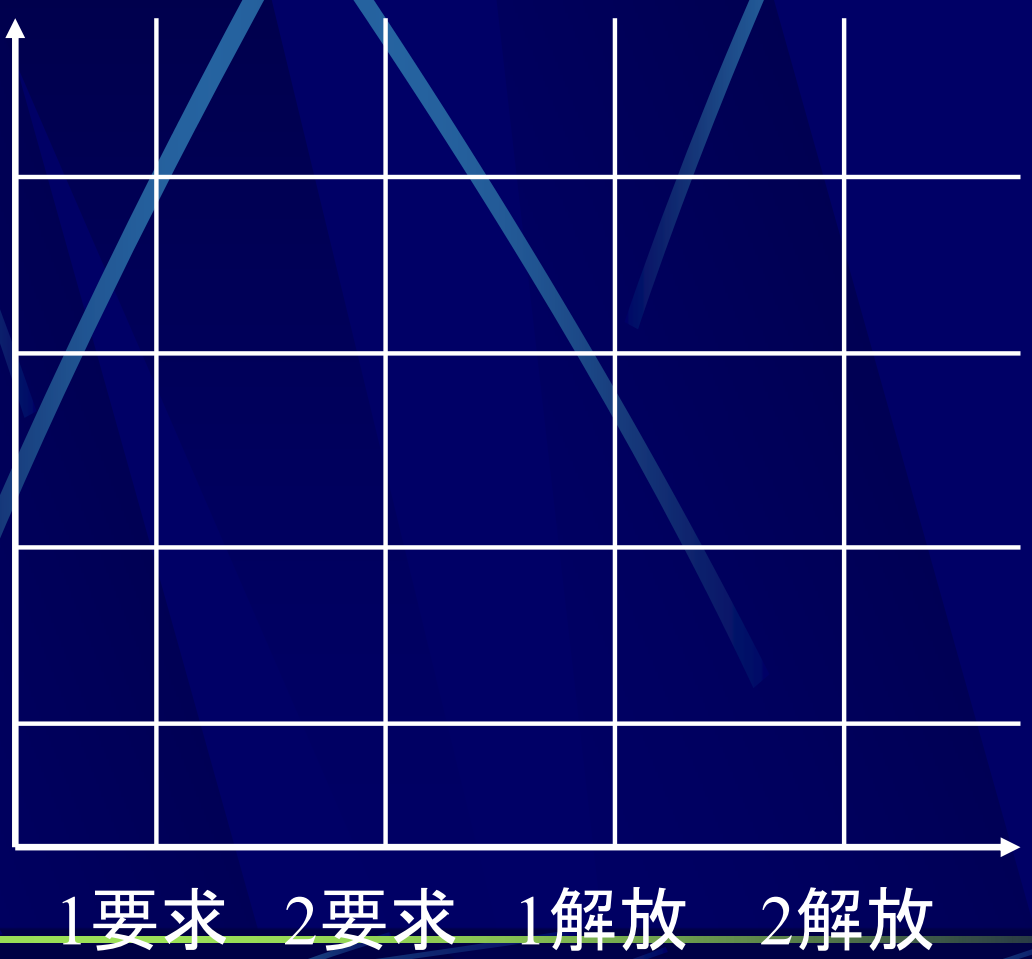
デッドロック発生の原理



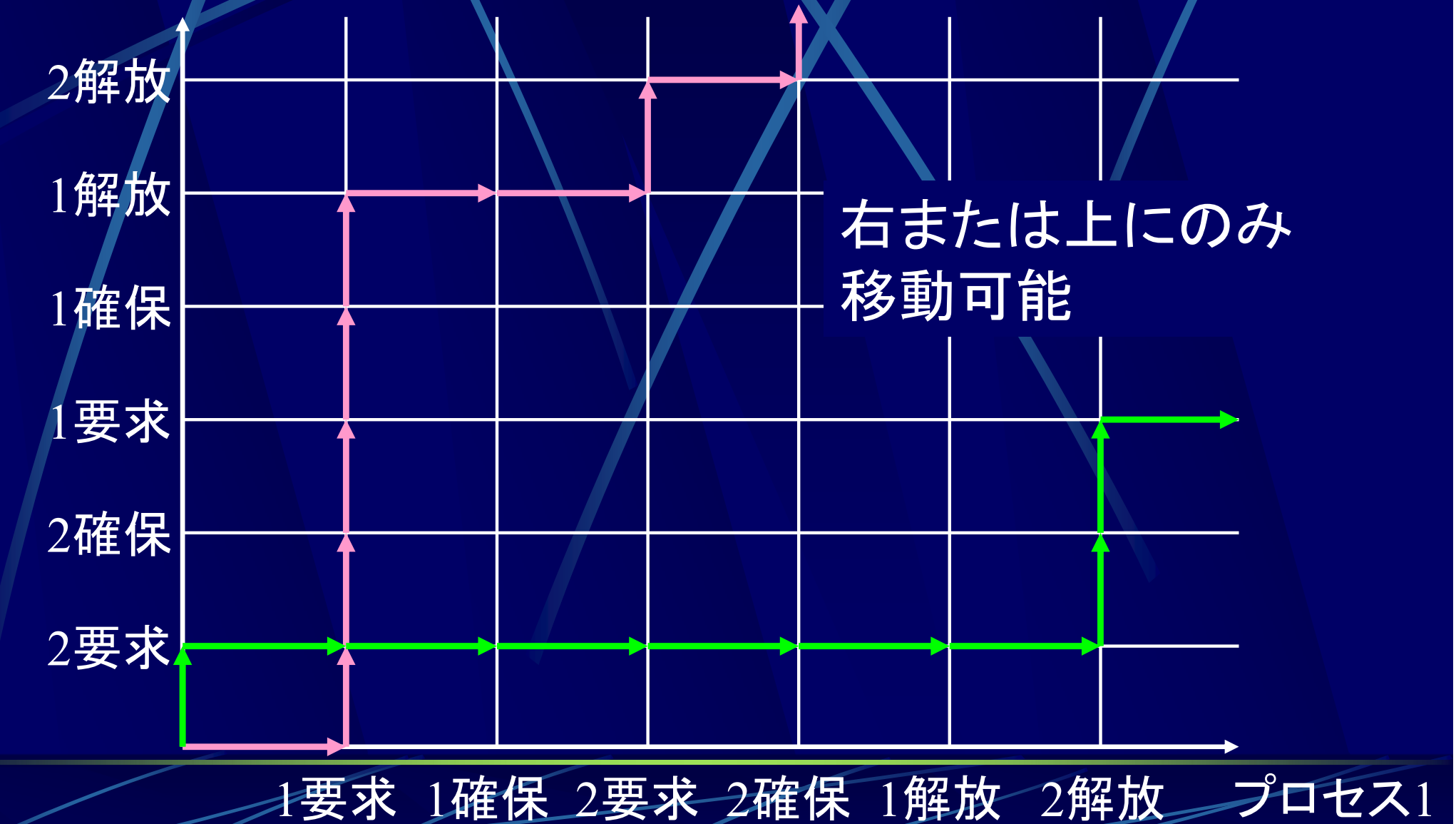
2プロセスでのデッドロック

プロセスの実行を2次元で表現する

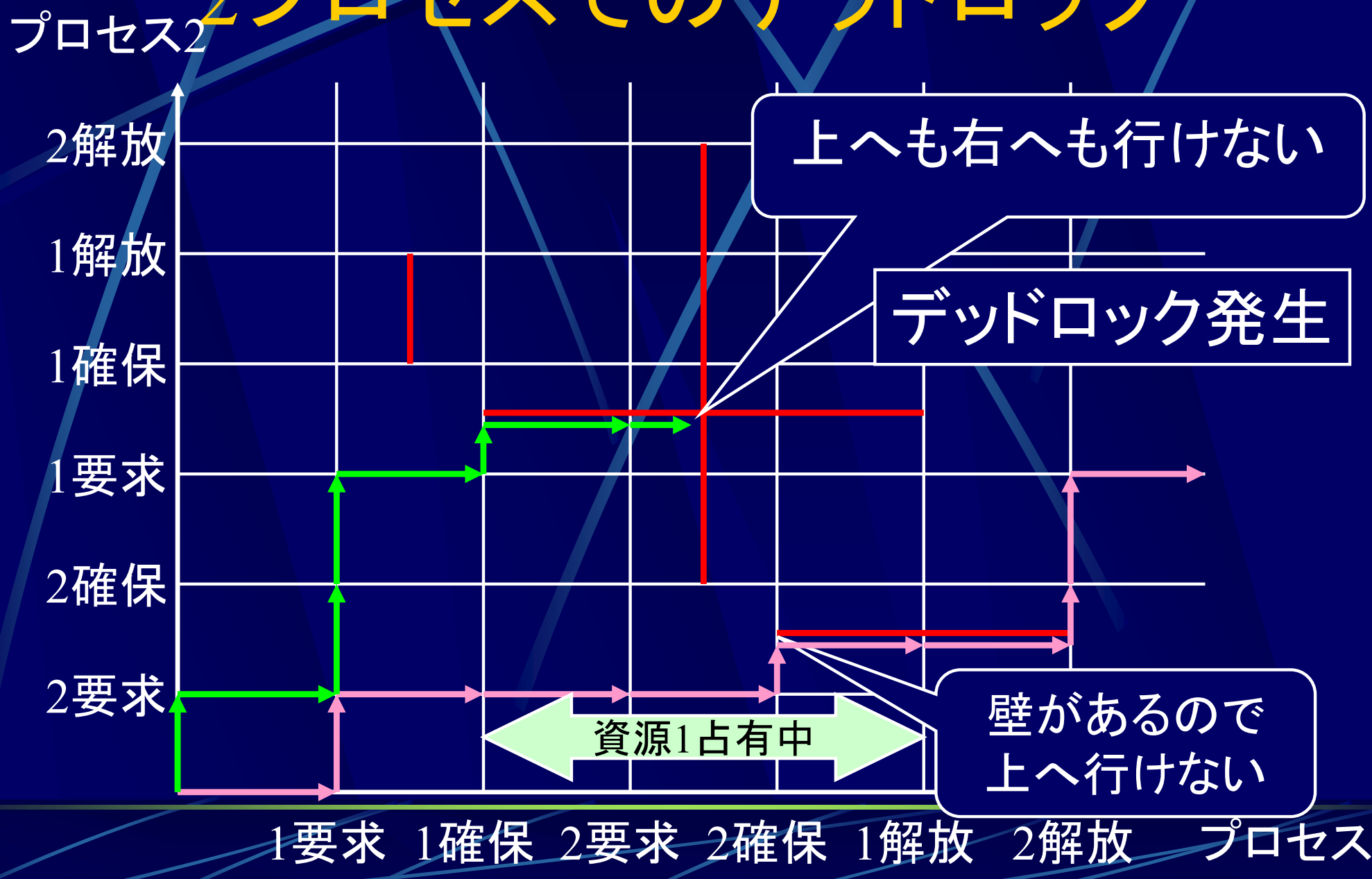
プロセス1	プロセス2
資源1要求	資源2要求
資源2要求	資源1要求
資源1解放	資源1解放
資源2解放	資源2解放



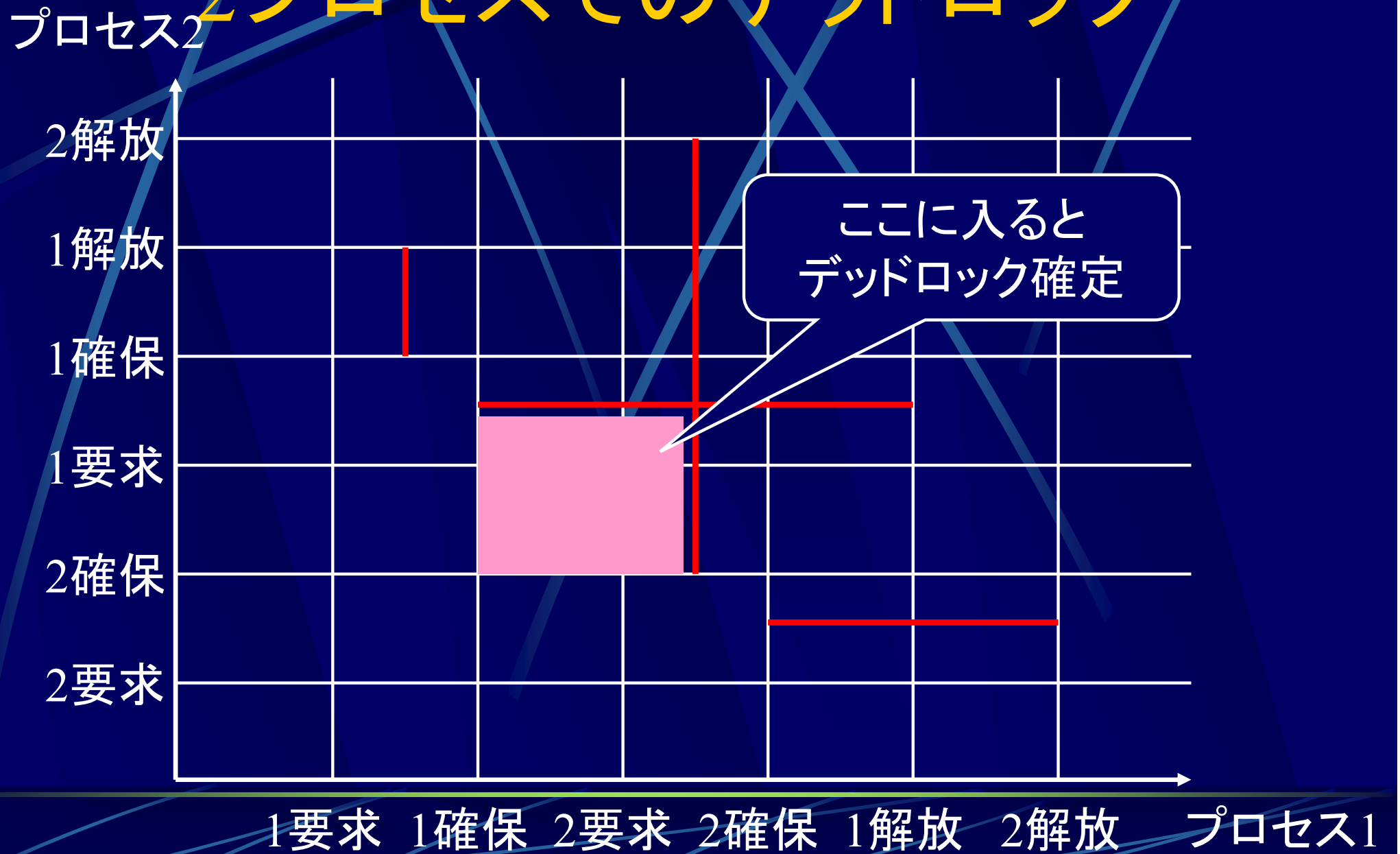
2プロセスでのデッドロック



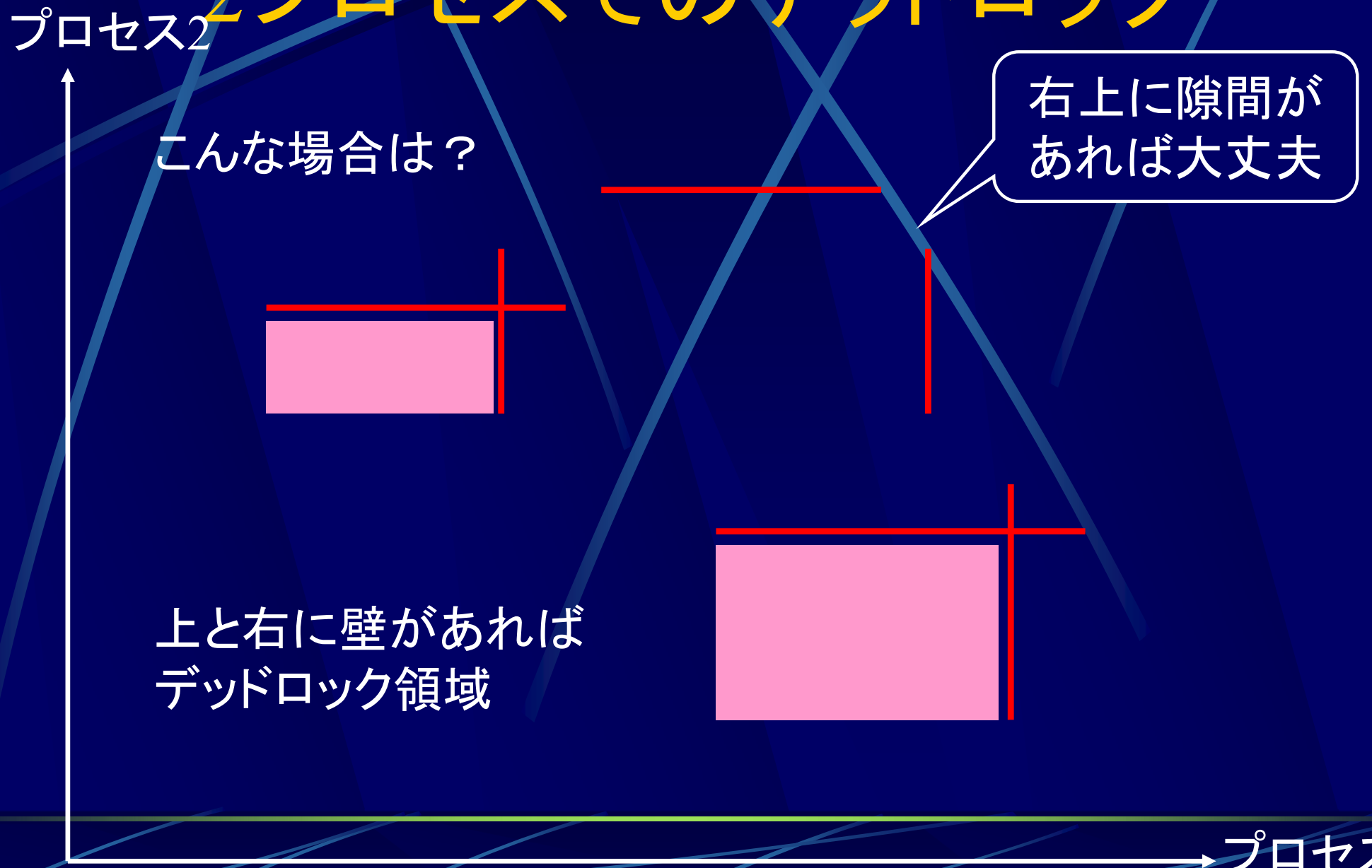
2プロセスでのデッドロック



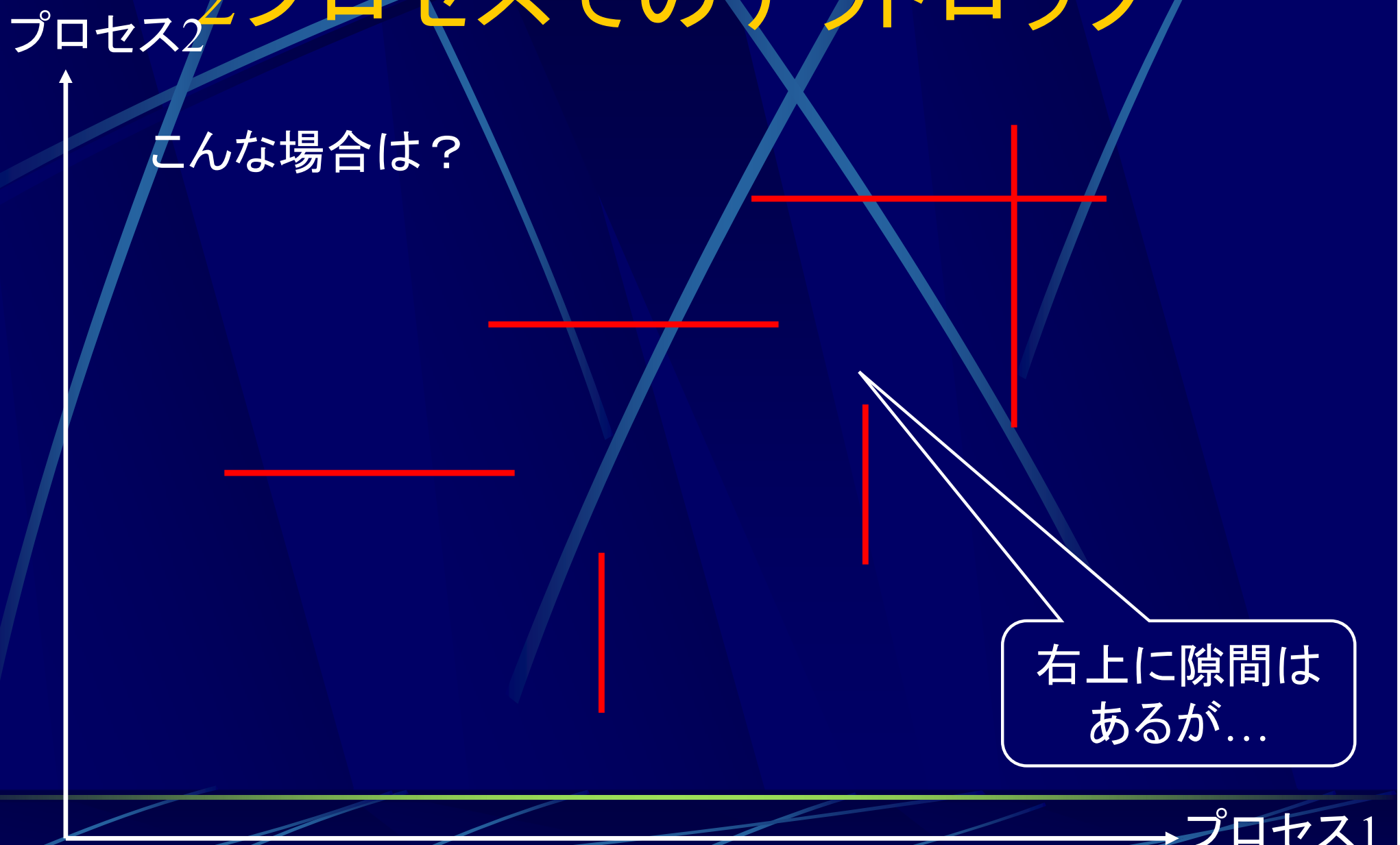
2プロセスでのデッドロック



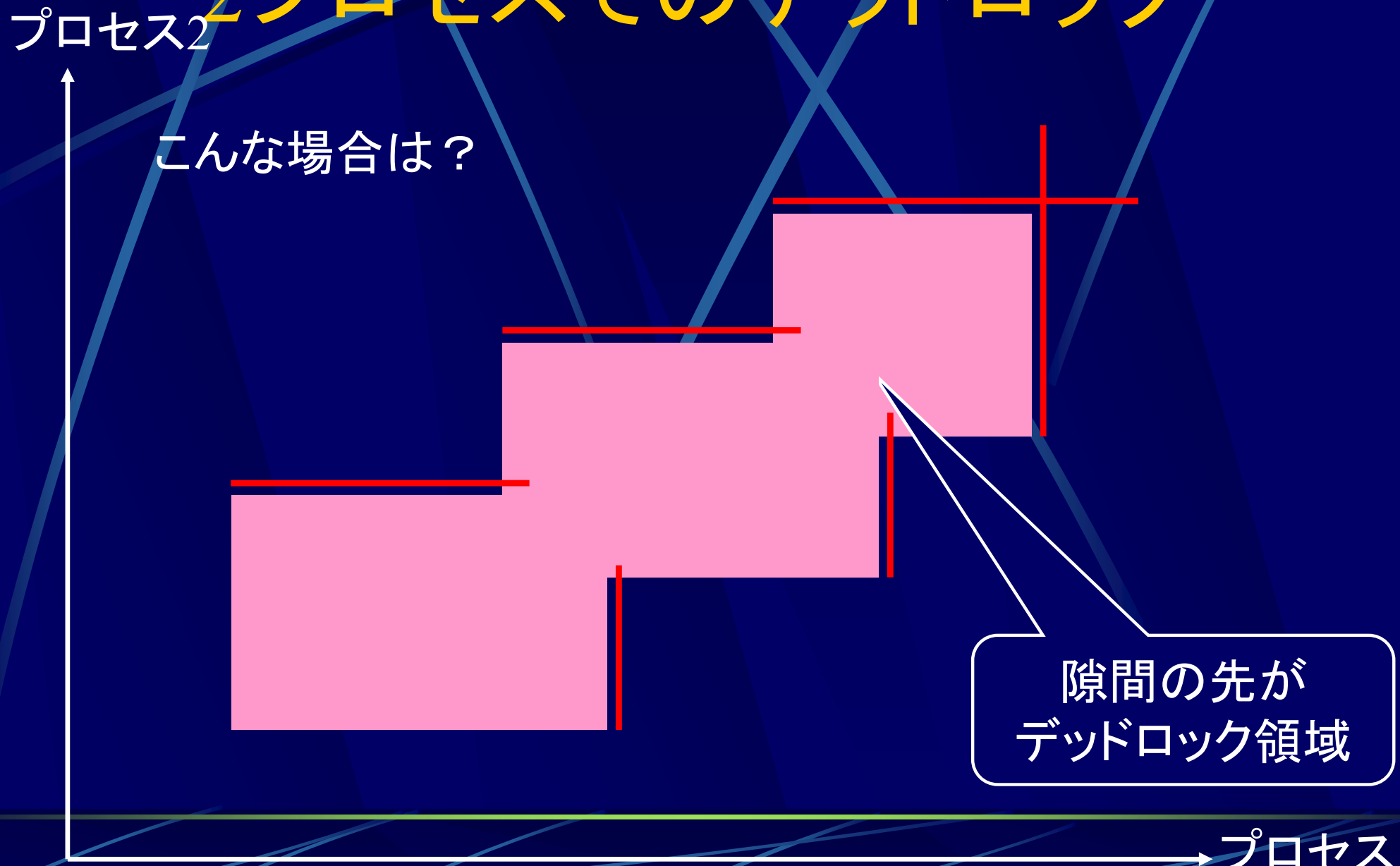
2プロセスでのデッドロック



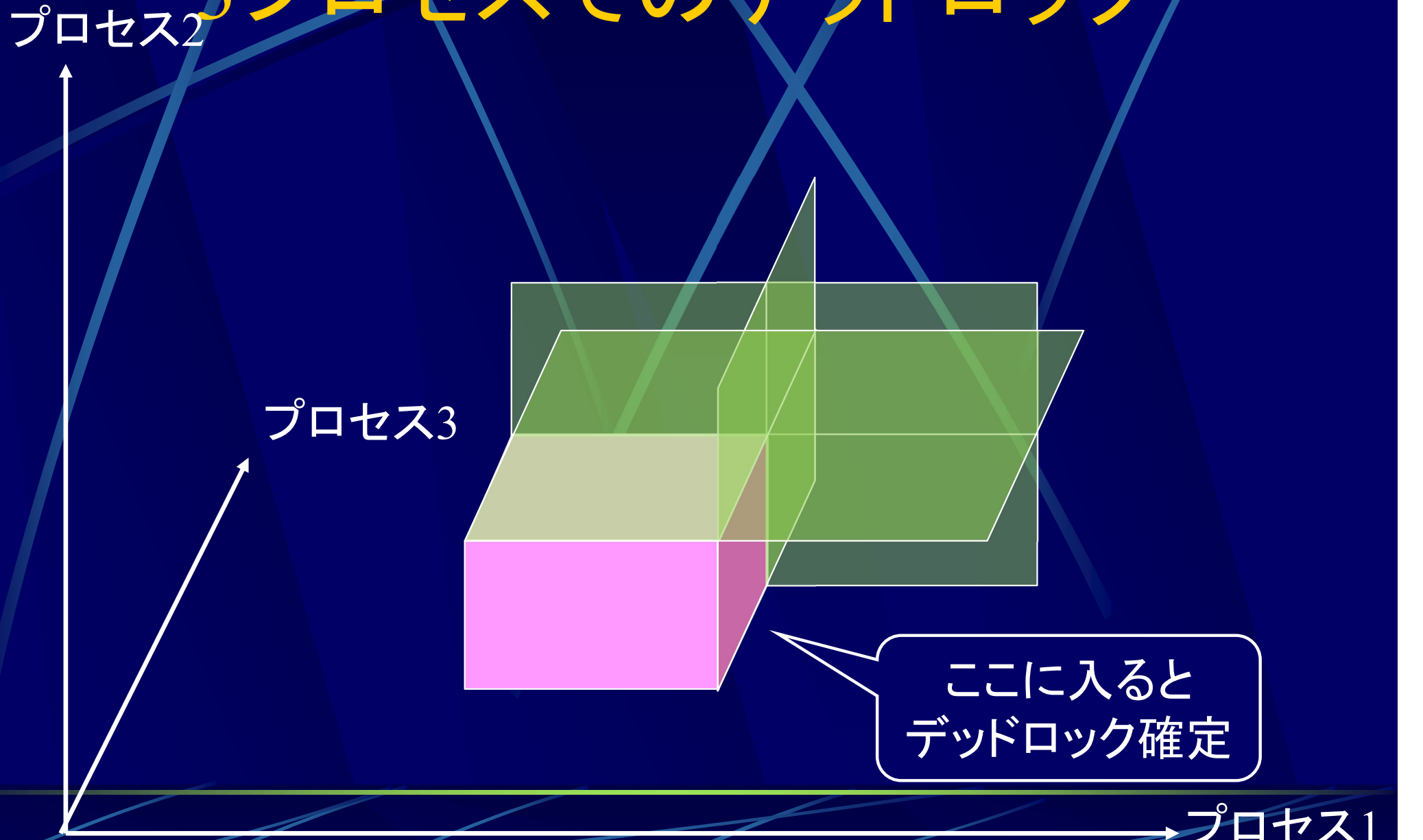
2プロセスでのデッドロック



2プロセスでのデッドロック



3プロセスでのデッドロック



デッドロック発生の条件

- 相互排除条件(mutual exclusion condition)
  - 排他的な資源要求をしている
- 待機条件(wait for condition)
  - 資源の確保が不可能な場合は待つ(ブロック状態)
- 横取り不能条件(no preemption condition)
  - 資源を確保したプロセスは強制的に資源を取られることは無い
- 循環待機条件(circular wait condition)
  - 各プロセスが資源を1つ以上確保し、他のプロセスがそれを要求している

上記の4条件が揃うとデッドロックが発生する

デッドロックへの対処法

- 無策
- デッドロックの検出(deadlock detection)
  - デッドロックの通知(deadlock notification)
  - デッドロックの回復(deadlock recovery)
- デッドロックの回避(deadlock avoidance)
- デッドロックの防止(deadlock prevention)

デッドロックへの対処法
無策

- 無策
  - デッドロック発生頻度と対策のトレードオフ

プロセス番号
i ノードテーブル } どちらも有限の値

⇩

値がオーバフローすると
デッドロックの可能性

しかしまずそんなことは起こらない、はず…

計算機を100年くらい電源いれっぱなしなら起きるかも…

デッドロックへの対処法
無策

- 無策でもOKとなる条件
  - デッドロック発生頻度が極めて低い
  - デッドロック発生時の被害が軽微
  - 管理者が常に監視

デッドロックの防止
(deadlock prevention)

- デッドロック発生の条件
  - 相互排除条件(mutual exclusion condition)
  - 待機条件(wait for condition)
  - 横取り不能条件(no preemption condition)
  - 循環待機条件(circular wait condition)

上記の4条件の1つを成り立たなくすれば
デッドロックは発生しない

デッドロックの防止

- 相互排除条件の回避
  - これは難しい…

同時に印刷できるのは
1枚だけ

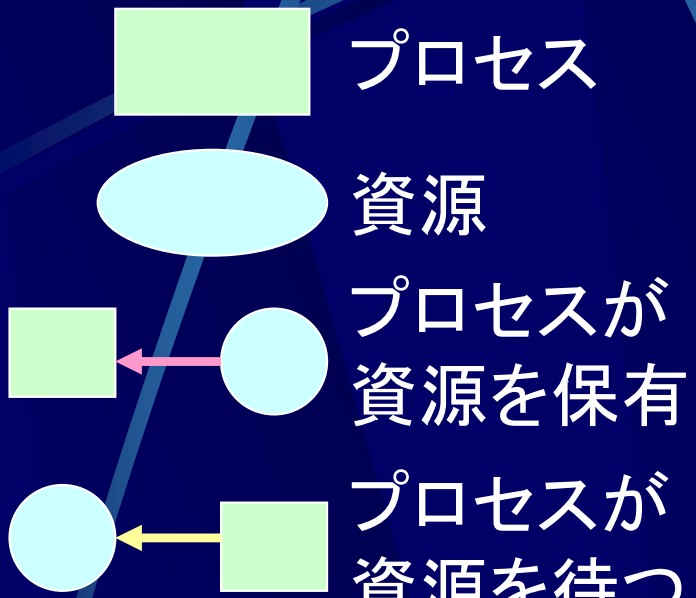
プリンタ

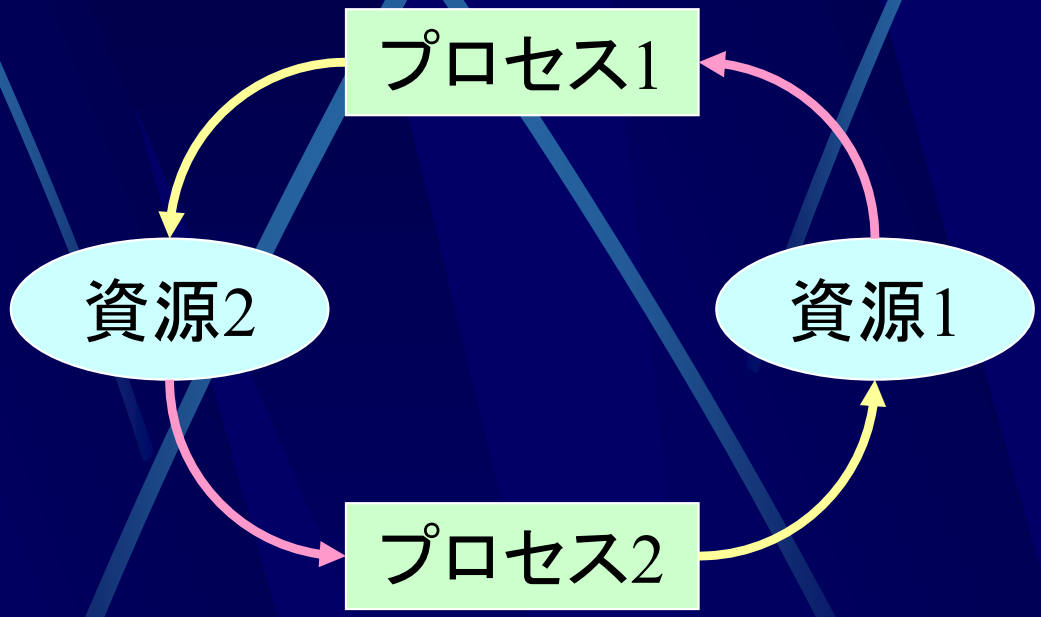
複数のプロセスが同時に使えないのは
資源そのものが持つ性質

デッドロックの防止

- 待機条件の回避
  - 必要な資源は全て同時に要求する
- 横取り不能条件の回避
  - 必要な資源を全て得られない場合、保持する資源を解放する
- 循環待機条件の回避
  - 資源を獲得する順番を決めておく

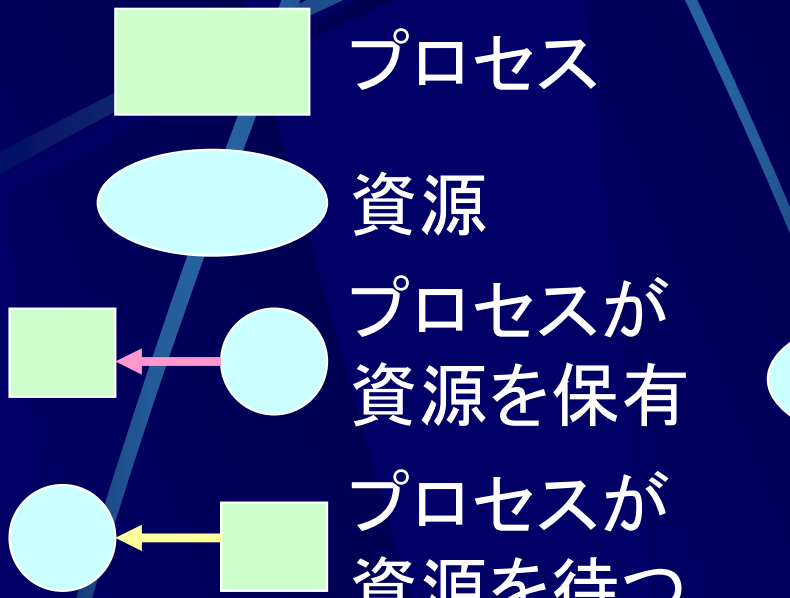
デッドロック発生の原理(再掲)

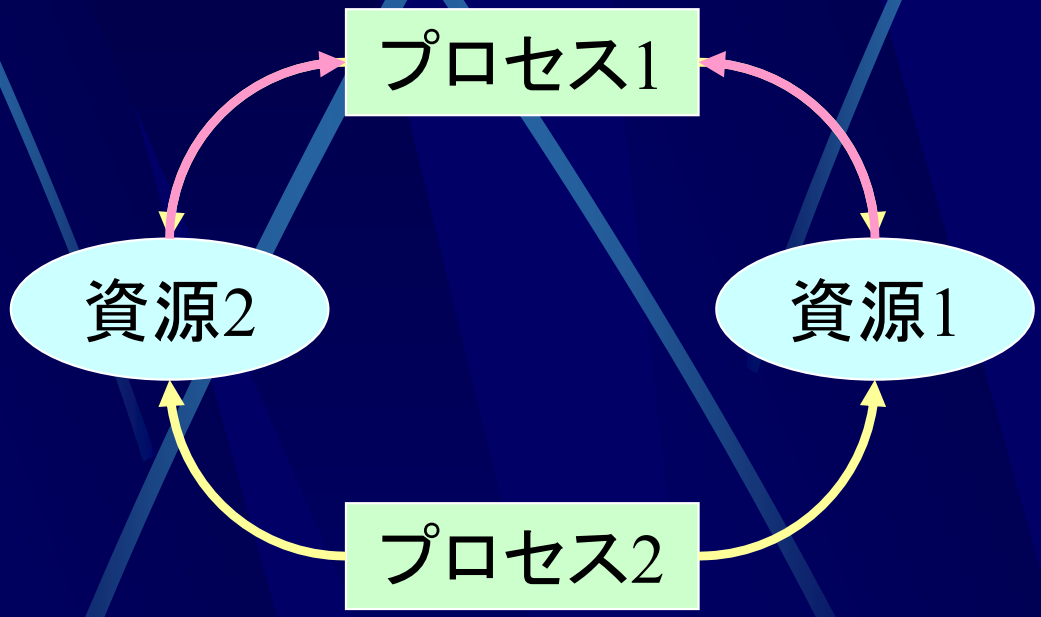




プロセス-資源の間で閉じた
ループができるとデッドロック発生

デッドロックの防止
待機条件の回避

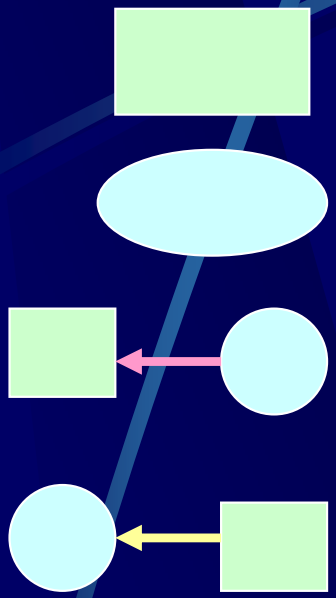


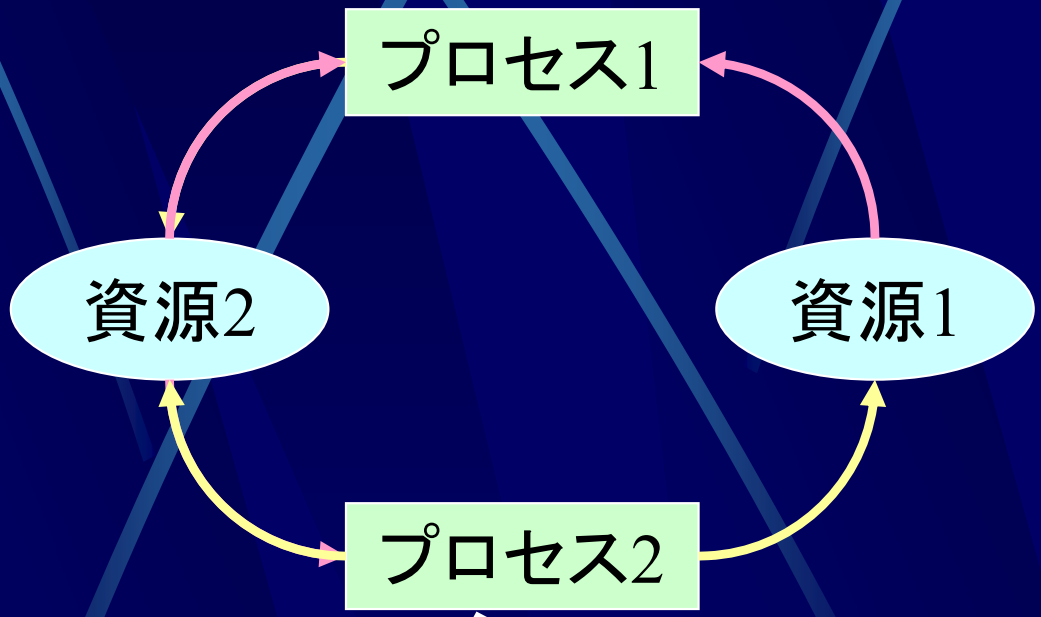


資源は全て同時に要求，
全ての資源が得られるまで先へ進まない

全ての資源が確保できるか、全く確保できないかのいずれか

デッドロックの防止
横取り不能条件の回避

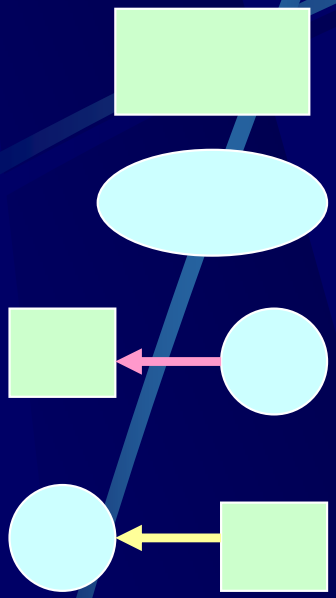


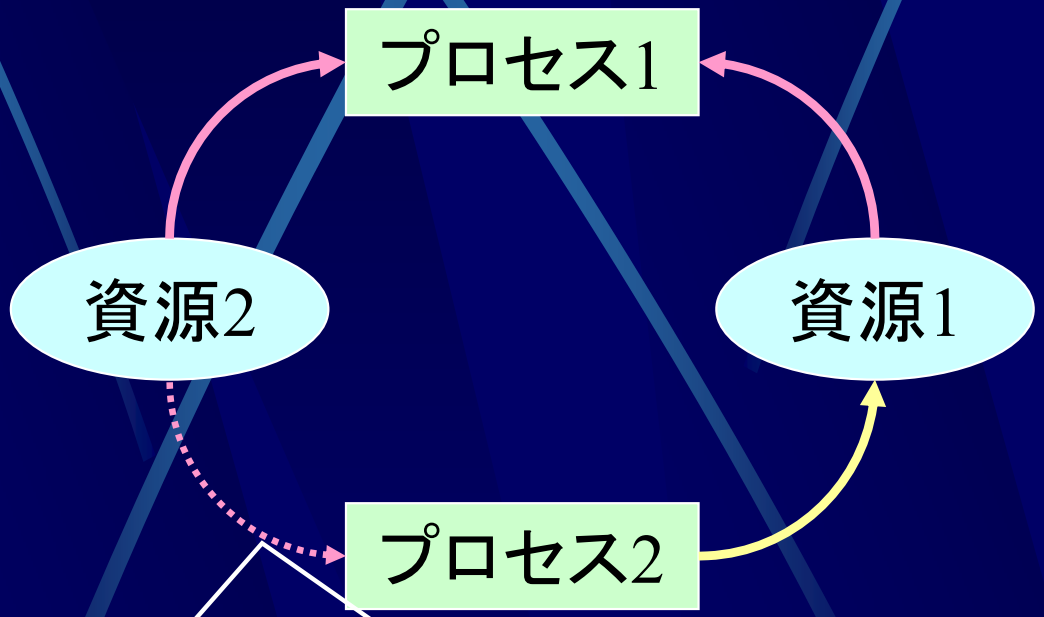


必要な資源を全て
得られなければ
資源を一旦解放する

資源1を得られないので
一旦資源2を解放する

デッドロックの防止
循環待機条件の回避





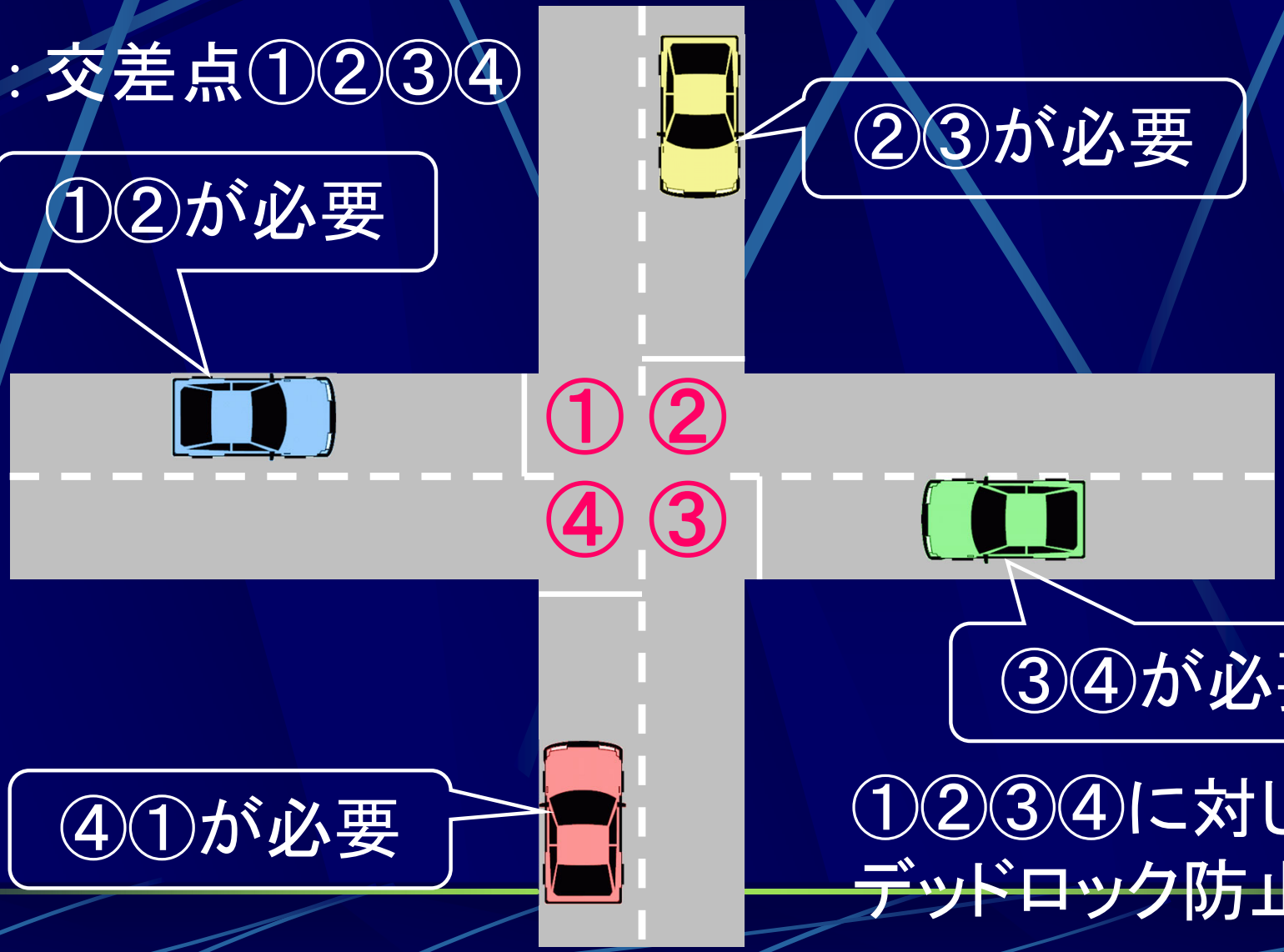
資源獲得順
資源1→資源2

資源1をまだ確保していないので
資源2を確保できない

プロセス1が両方の資源を獲得できる

デッドロックの防止

資源：交差点①②③④

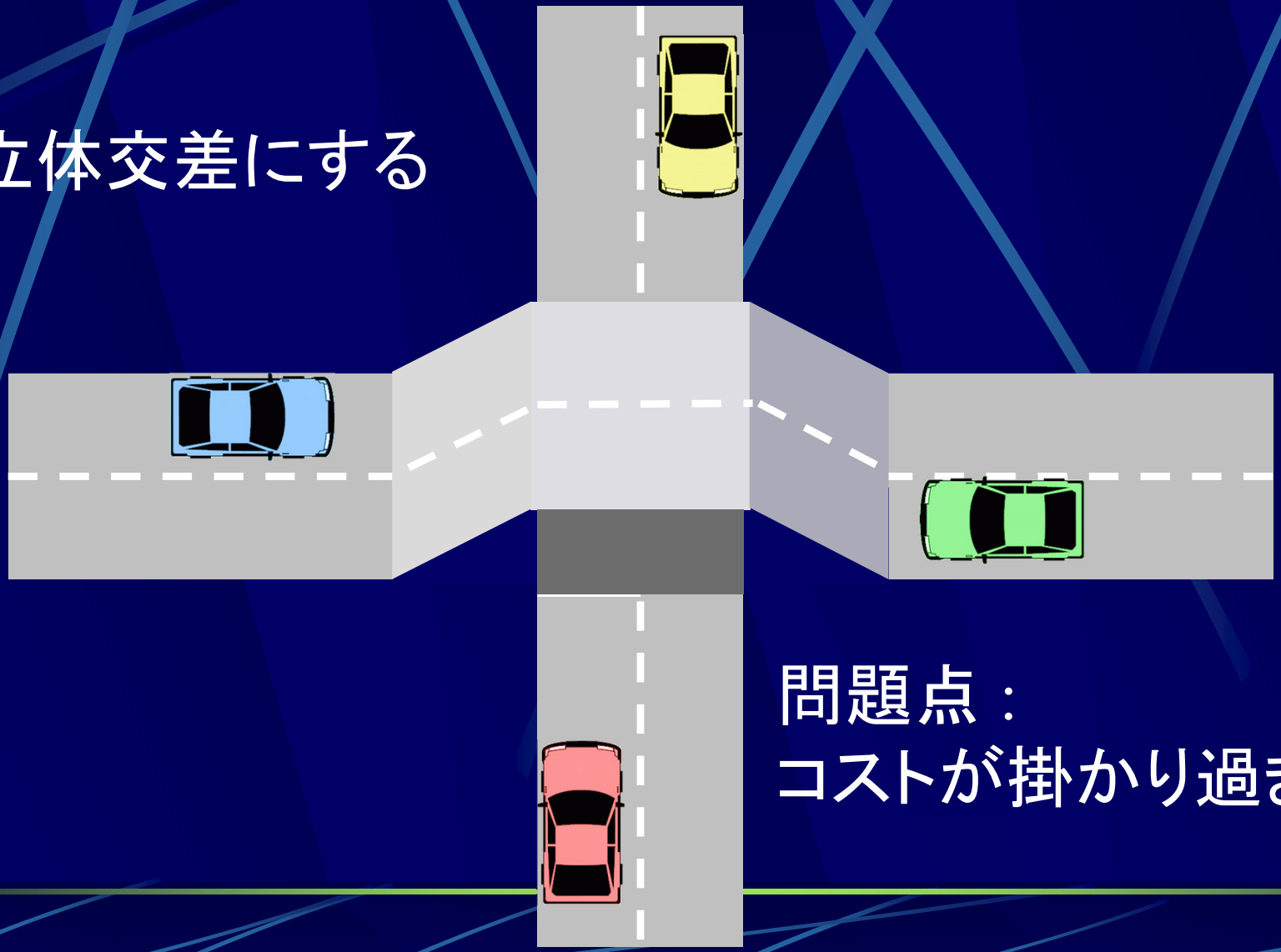


①②③④に対して
デッドロック防止可能？

デッドロックの防止

相互排除条件の回避

立体交差にする

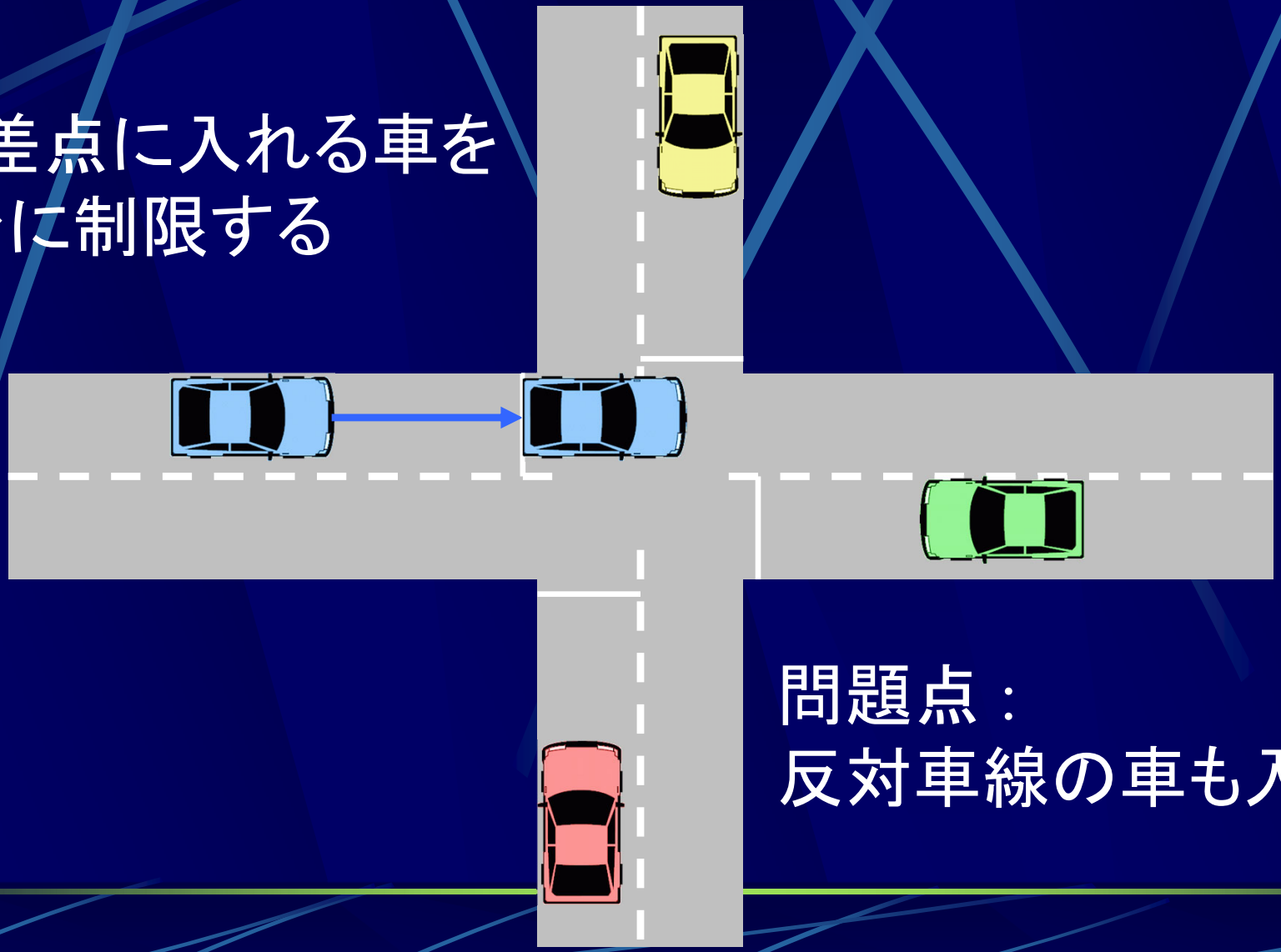


問題点：
コストが掛かり過ぎる

デッドロックの防止

待機条件の回避

交差点に入れる車を
1台に制限する

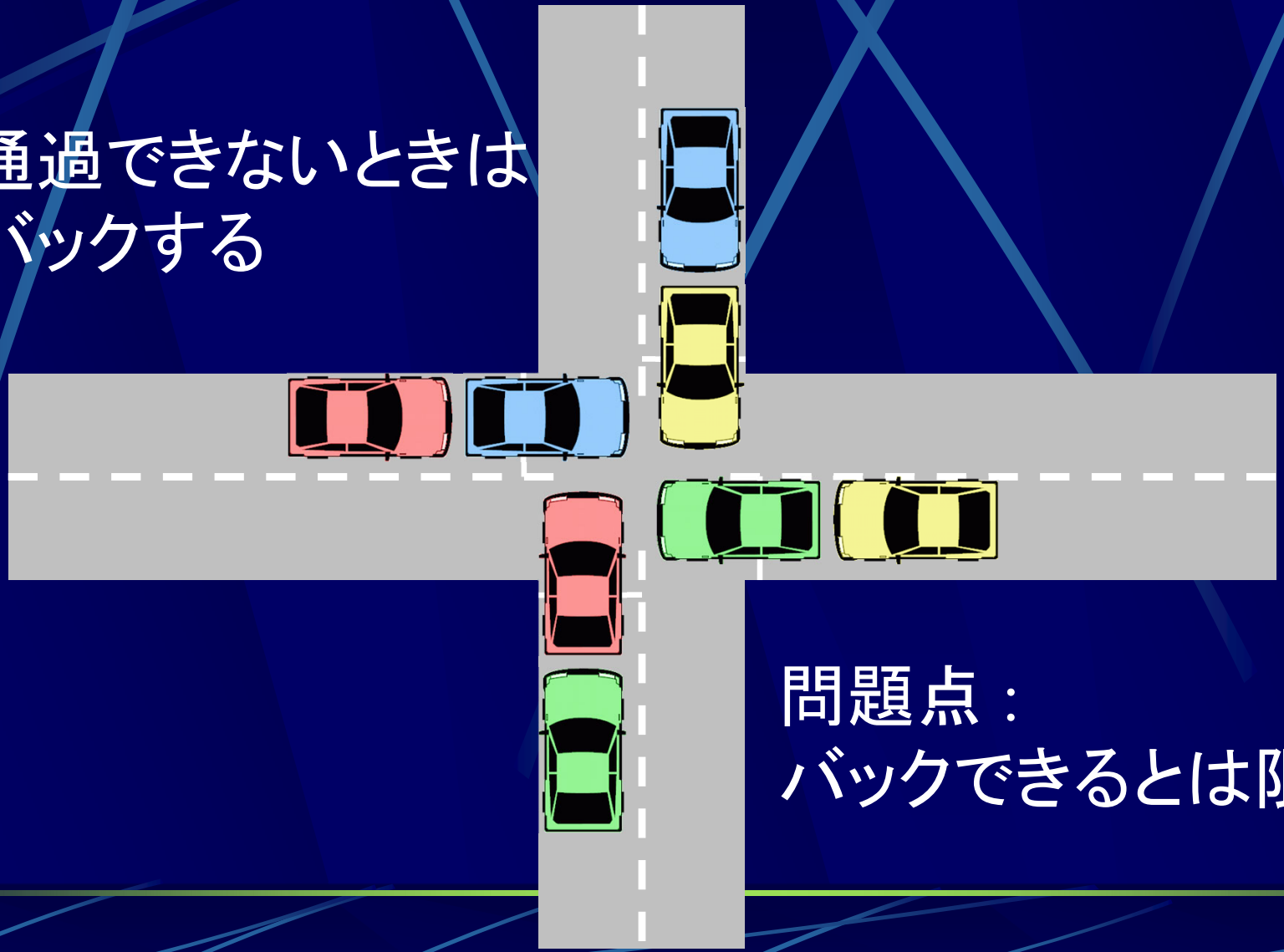


問題点：
反対車線の車も入れない

デッドロックの防止

横取り不能条件の回避

通過できないときは
バックする

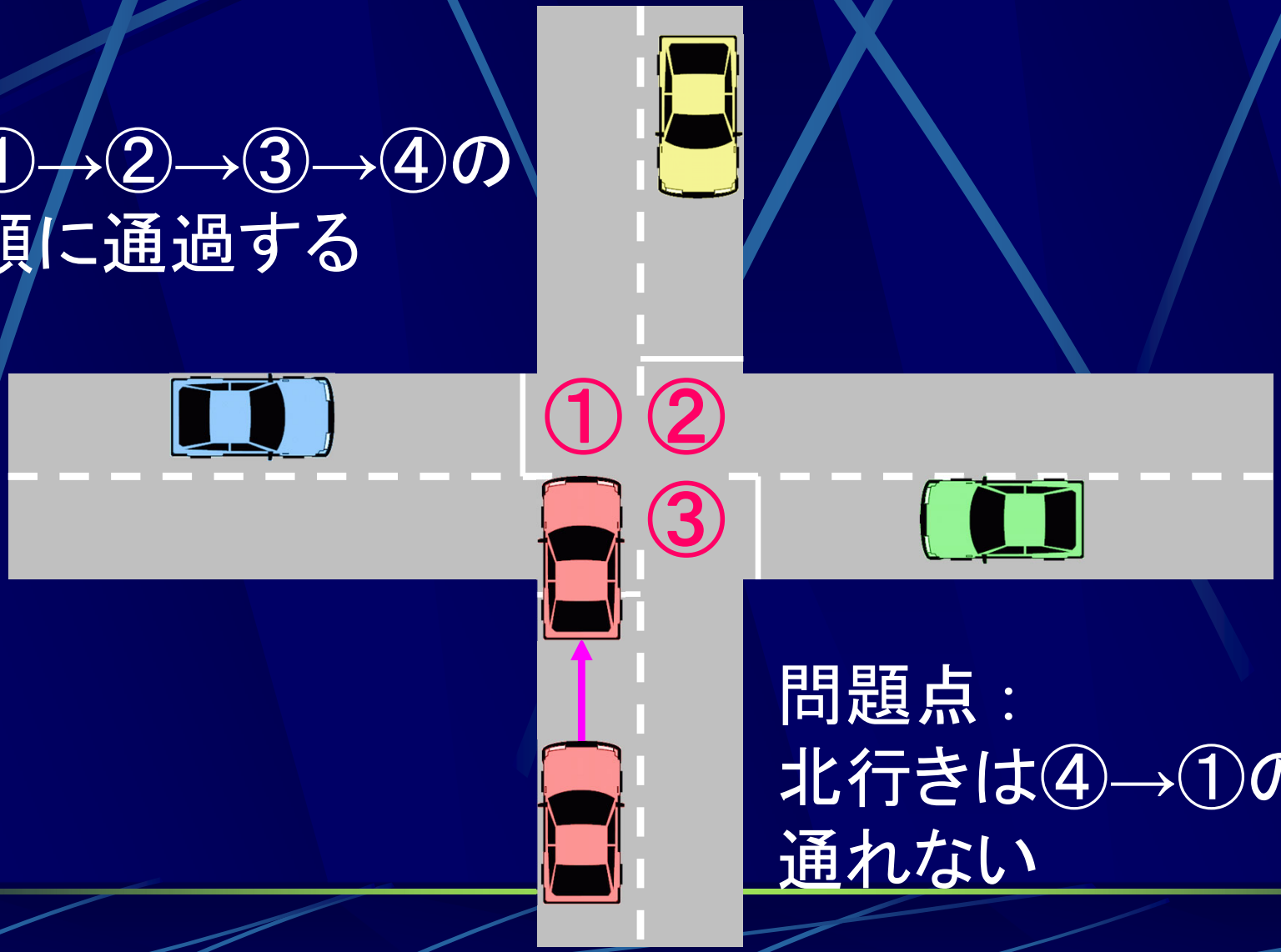


問題点：
バックできるとは限らない

デッドロックの防止

循環待機条件の回避

①→②→③→④の
順に通過する



問題点：
北行きは④→①の順でしか
通れない

デッドロックの防止

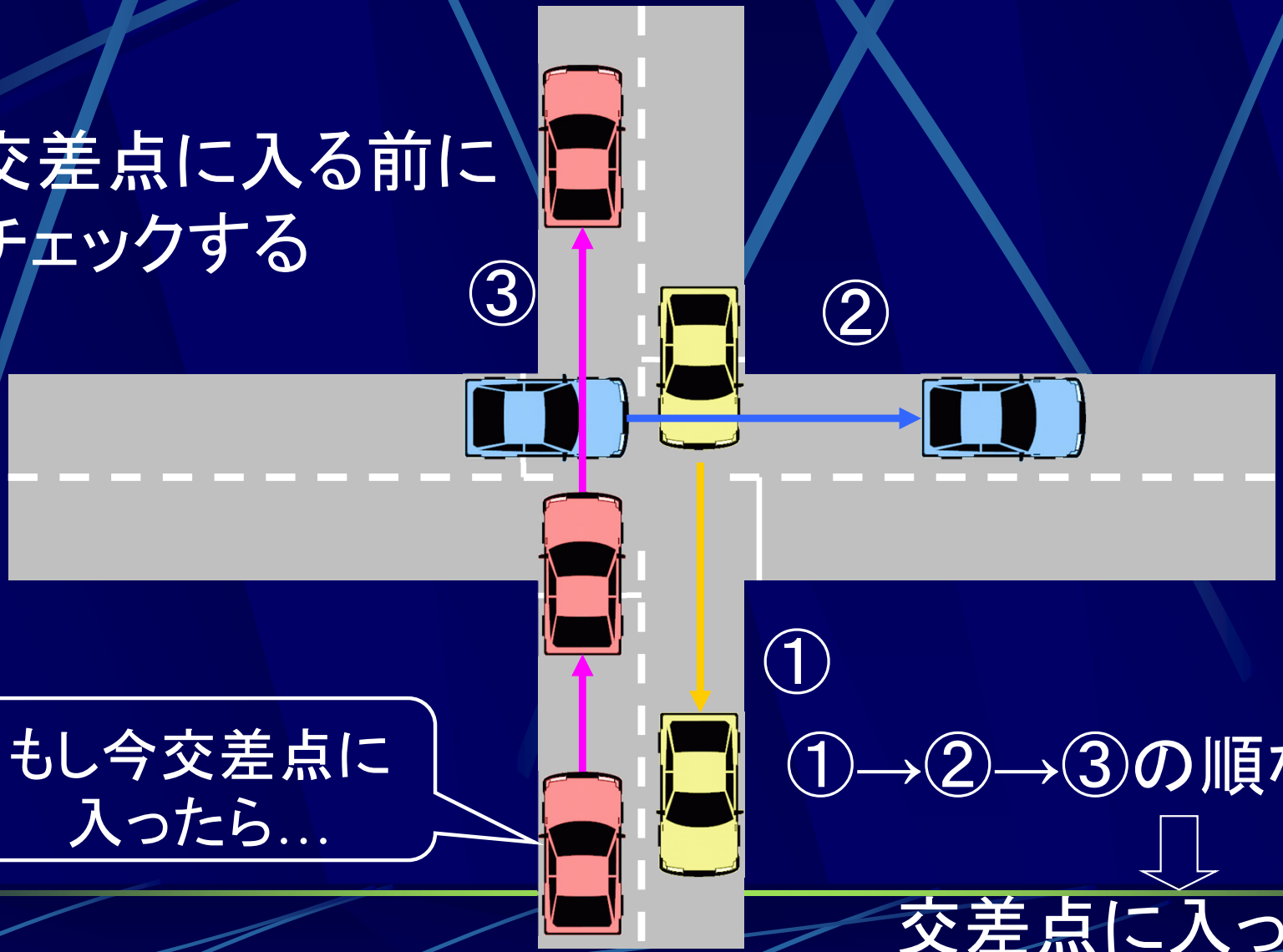
- 待機条件の回避
  - 必要な資源は全て同時に要求する
  - ✓ 必要な資源が予めわかるとは限らない
- 横取り不能条件の回避
  - 必要な資源を全て得られない場合、保持する資源を解放する
  - ✓ 横取り不能資源の存在
- 循環待機条件の回避
  - 資源を獲得する順番を決めておく
  - ✓ 全てのプロセスに都合のいい順番が存在するとは限らない

デッドロックの回避 (deadlock avoidance)

- デッドロックの回避(deadlock avoidance)
  - 資源を確保する前に、資源を確保したことによりデッドロックが起きる可能性が無いかチェックする

デッドロックの回避

交差点に入る前に
チェックする



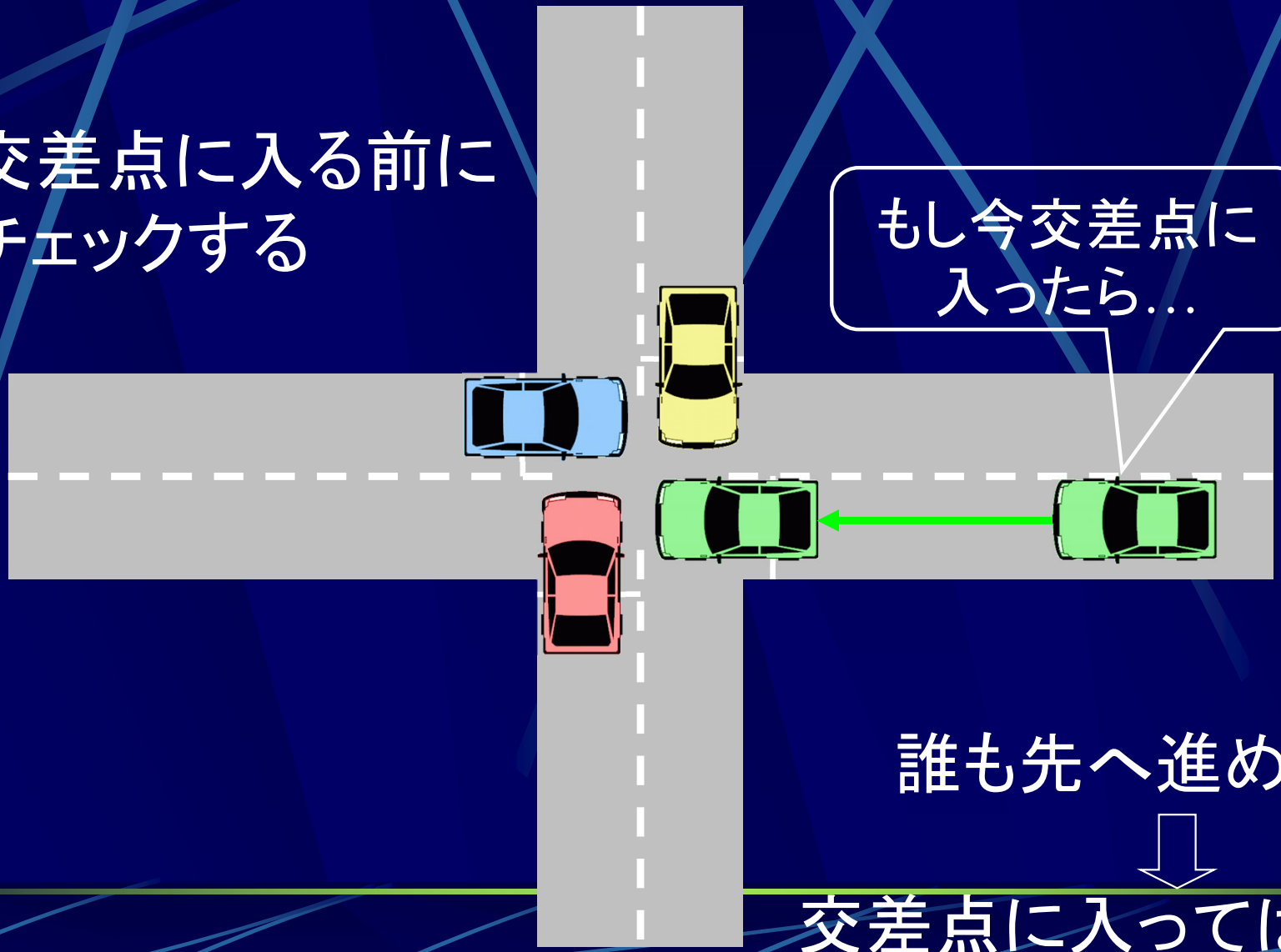
もし今交差点に
入ったら…

①→②→③の順なら通れる

交差点に入っていい

デッドロックの回避

交差点に入る前に
チェックする



誰も先へ進めない

⇩

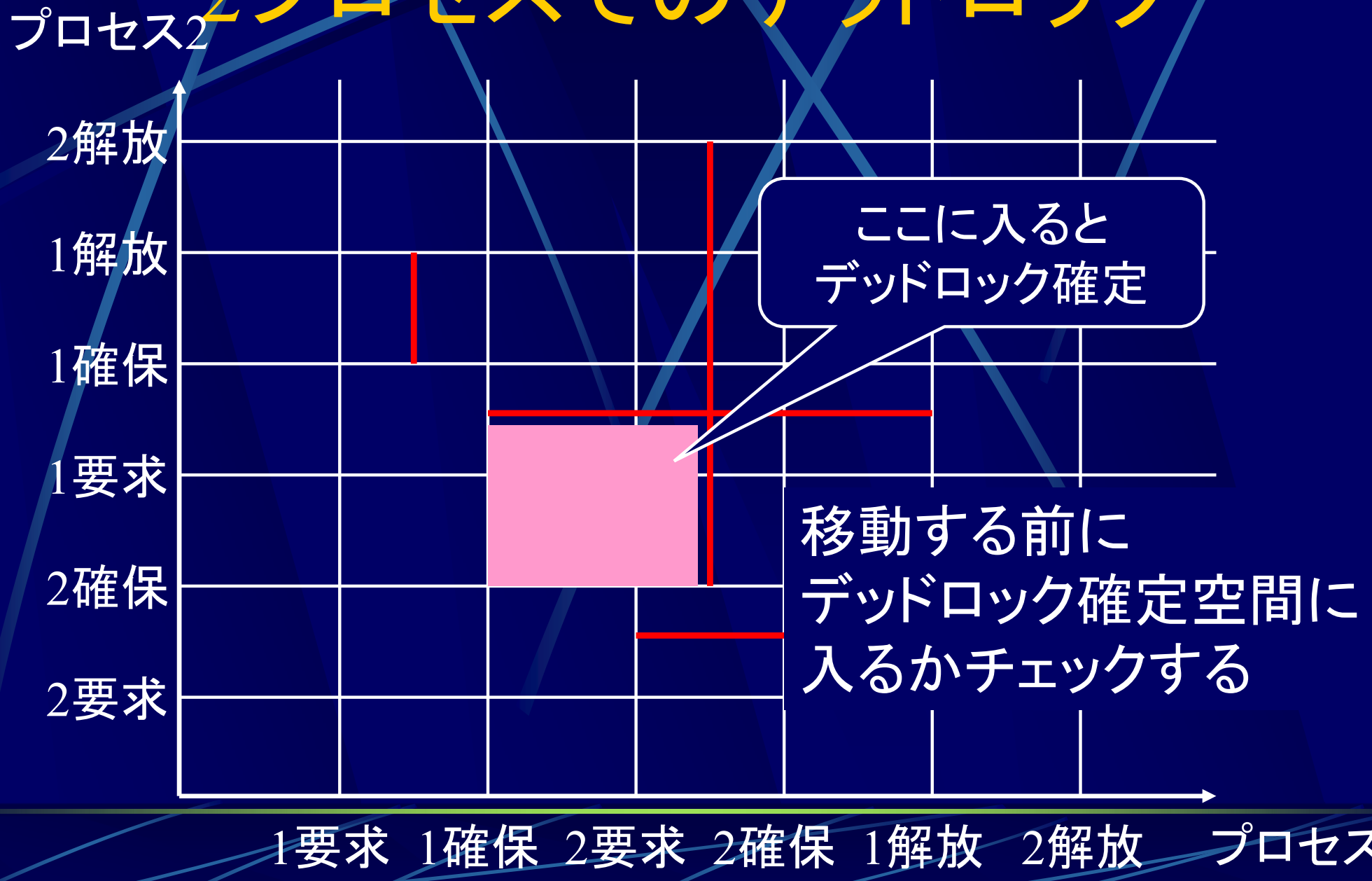
交差点に入ってはいけない

デッドロックの防止と デッドロックの回避

デッドロックする可能性のある資源要求に対して

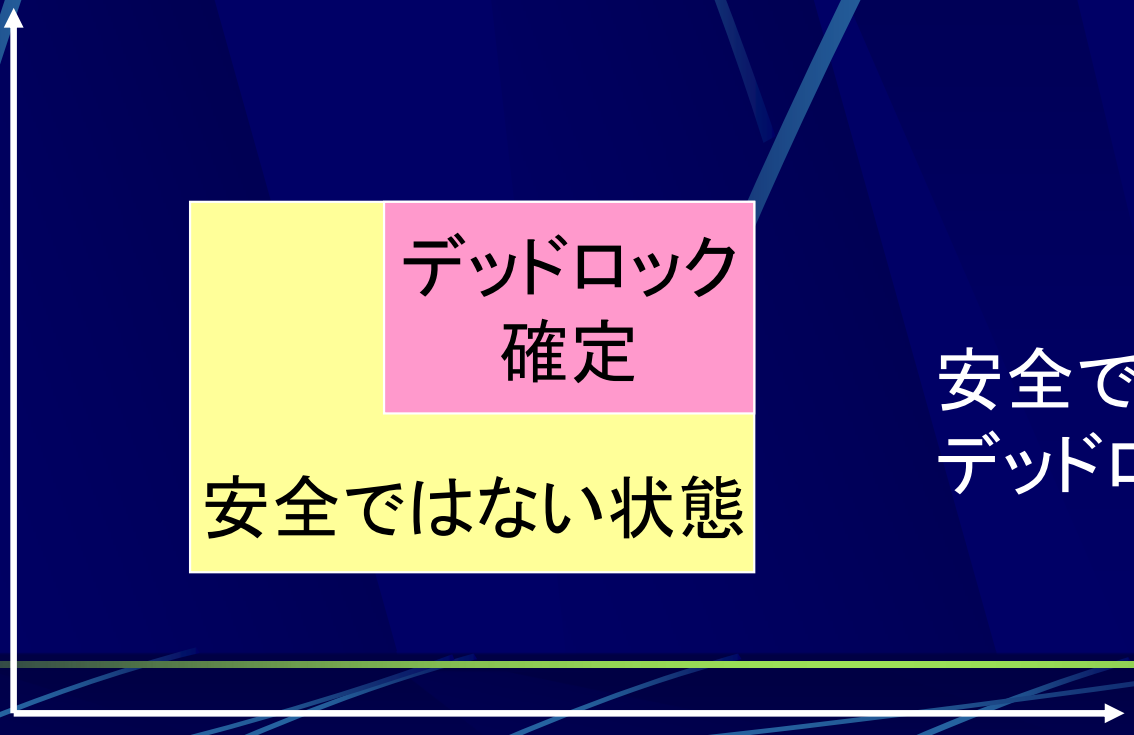
- デッドロックの防止
  - 資源要求そのものを禁止する
- デッドロックの回避
  - 資源を渡す前に渡したときにデッドロックにならないかチェックをする

2プロセスでのデッドロック



安全(safe)な状態

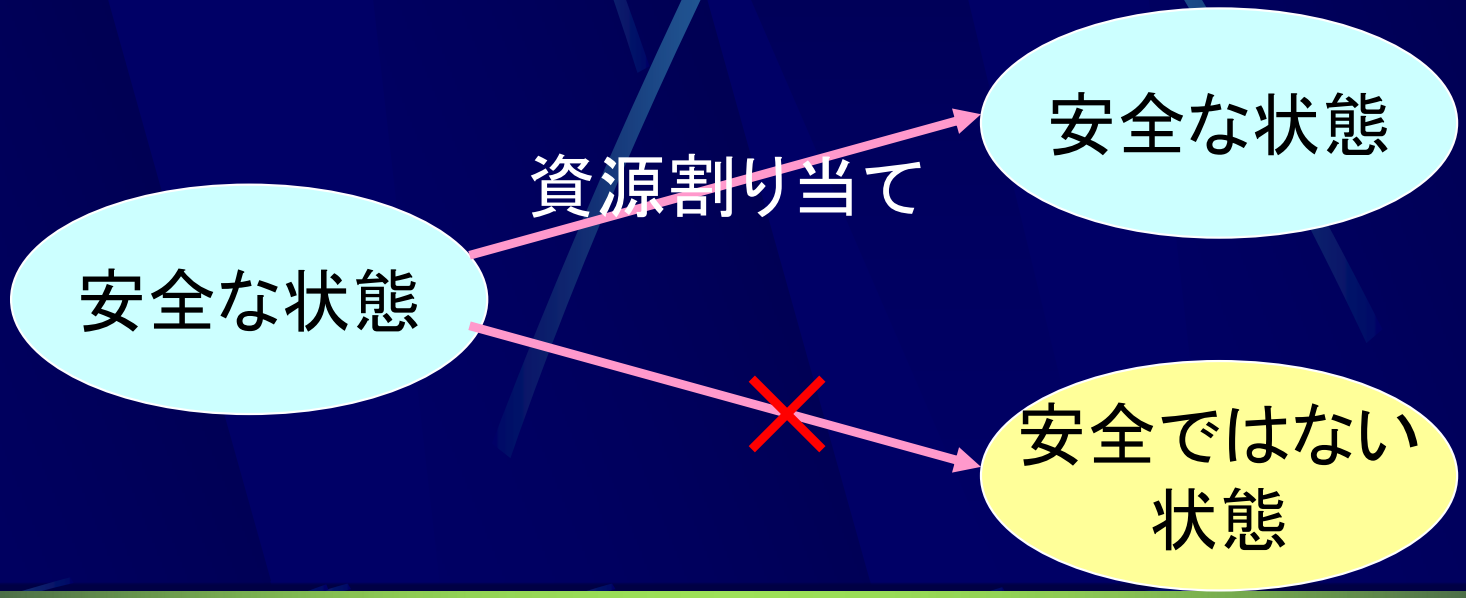
- 安全(safe)な状態
  - ある順序で資源を割り付け可能, かつデッドロック回避可能な状態



安全ではない状態になると
デッドロックの可能性がある

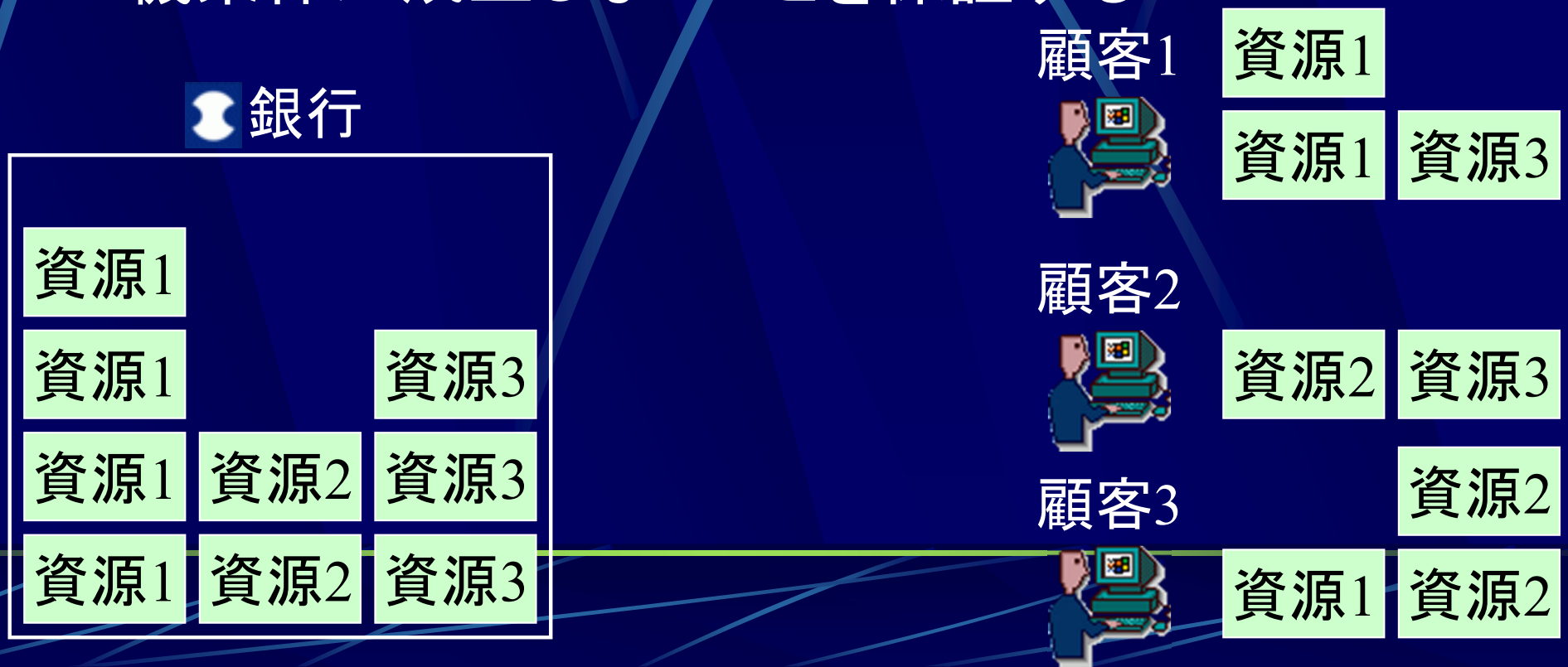
安全な状態と資源割り当て

- 資源を要求された場合
  - 割り当て後も安全な状態であれば資源を割り当てる
  - 割り当て後に安全な状態で無くなるならば要求を拒絶する



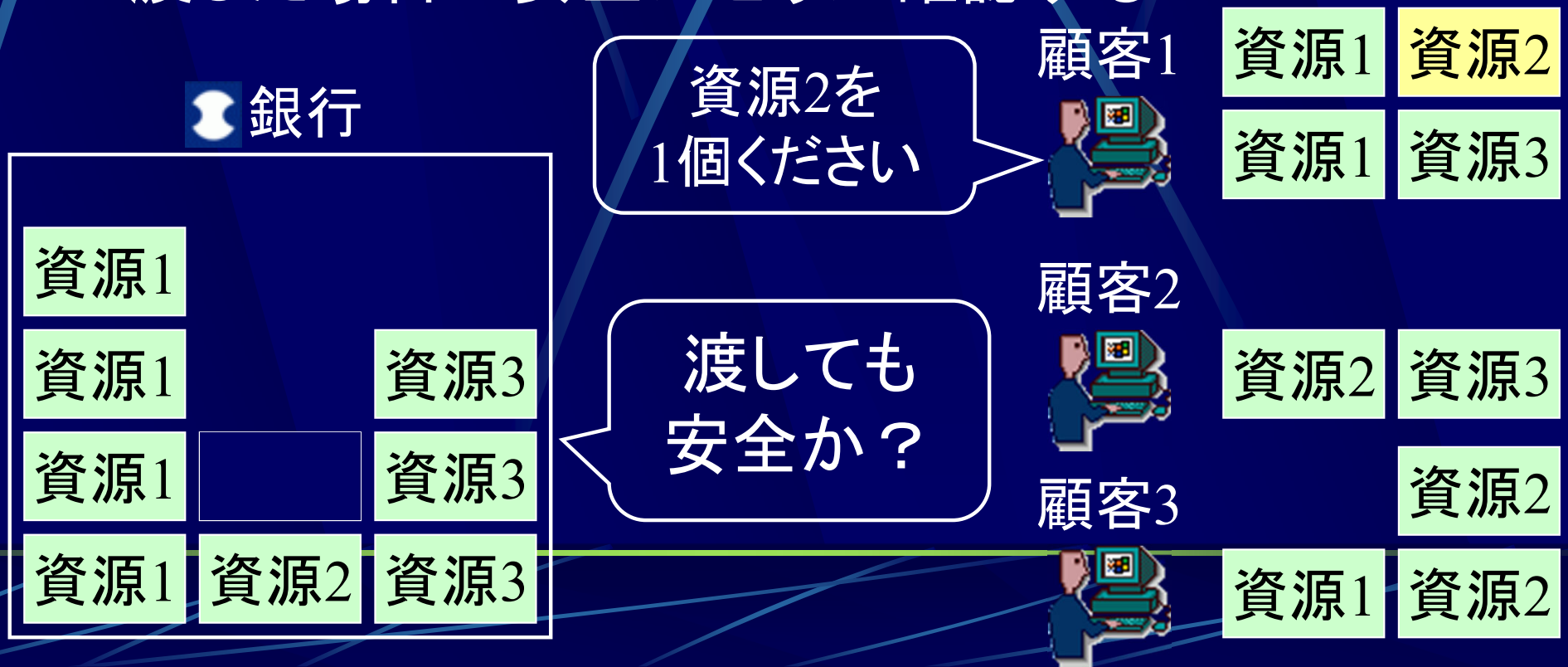
銀行家のアルゴリズム (banker's algorithm)

- 銀行家のアルゴリズム(banker's algorithm)
  - 資源の割付を動的に調べることにより循環待機条件が成立しないことを保証する

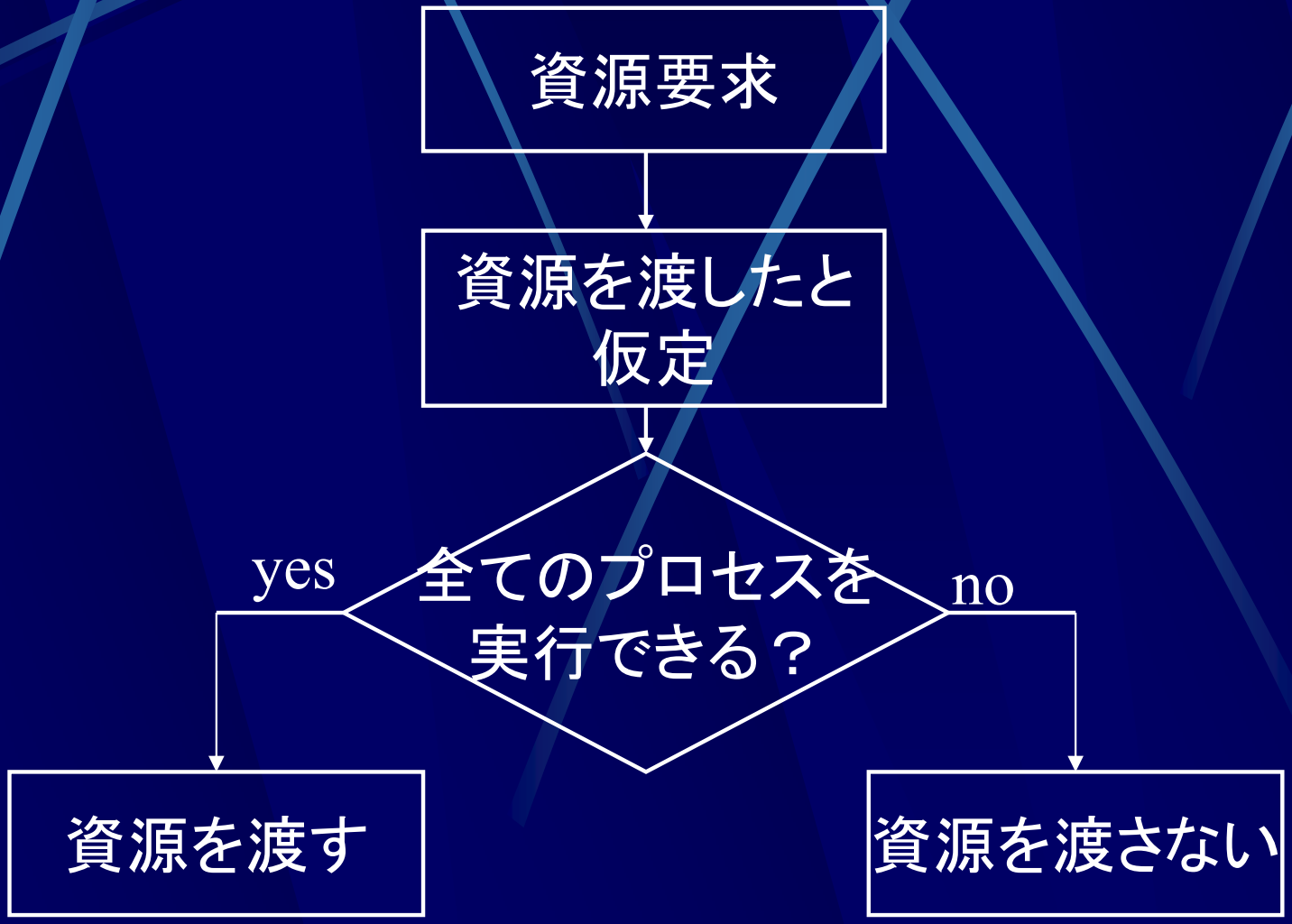


銀行家のアルゴリズム

- 資源を渡す前に安全性を確認する
  - 資源を要求されたときに、要求通りに資源を渡した場合に安全かどうか確認する

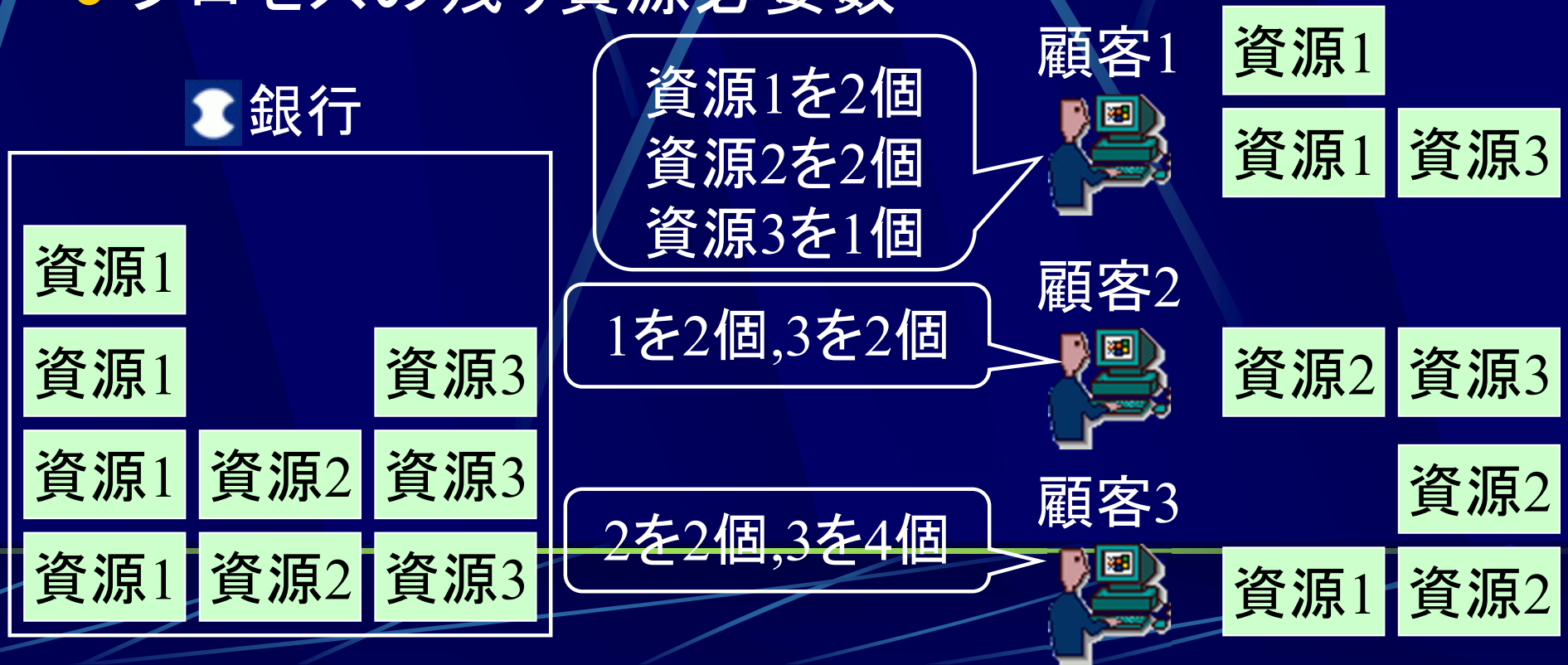


銀行家のアルゴリズム



銀行家のアルゴリズム

- 資源割付の状況を管理
  - 空き資源の数
  - プロセスに割り付けられている資源の数
  - プロセスの残り資源必要数



銀行家のアルゴリズム

- プロセス1～n, 資源1～m
  - F_j $(1 \leqq j \leqq m)$
    - 空き資源の数
  - U_{ij} $(1 \leqq i \leqq n, 1 \leqq j \leqq m)$
    - プロセスに割り付けられている資源の数
  - R_{ij} $(1 \leqq i \leqq n, 1 \leqq j \leqq m)$
    - プロセスの残りの資源の必要数
  - N_{ij} $(1 \leqq i \leqq n, 1 \leqq j \leqq m)$
    - プロセスが要求する資源の数

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	3	1	1	2

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	1,2	2,1	2,0	0,1
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	0,1

この状態は安全か?　　(保有資源数, 残り必要資源数)

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	3	1	1	2

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	1,2	2,1	2,0	0,1
2	1,0	0,**3**	3,0	3,2
3	1,0	1,1	1,**4**	0,1

資源2が不足

資源3が不足

(保有資源数, 残り必要資源数)

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	3	1	1	2

U,R

資源 プロセス	1			
1	1,2	2,1	2,0	0,1
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	0,1

プロセス1の全ての資源が
必要資源数 ≦ 空き資源数

資源2が不足

資源3が不足

まずプロセス1に必要な資源を全て渡す

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	1	0	1	1

資源確保

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	3,0	3,0	2,0	1,0
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	0,1

プロセス1は資源を全て得たので実行

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	4	3	3	2

資源解放

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	0,1

終了

プロセス1は実行終了後資源を解放する

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	4	3	3	2

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	0,1

終了

資源3が不足

(保有資源数, 残り必要資源数)

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	4	3	3	2

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0		0,0	0,0
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	0,1

今後はプロセス2の全ての資源が
必要資源数 ≦ 空き資源数

終了

資源3が不足

次はプロセス2に必要な資源を全て渡す

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	4	0	3	0

資源確保

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	1,0	3,0	3,0	5,0
3	1,0	1,1	1,4	0,1

終了

プロセス2は資源を全て得たので実行

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	5	3	6	5

資源解放

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	0,0	0,0	0,0	0,0
3	1,0	1,1	1,4	0,1

終了

終了

プロセス2は実行後資源を返却する

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	5	3	6	5

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0			
2	0,0		0,0	0,0
3	1,0	1,1	1,4	0,1

終了

終了

プロセス3も全ての資源が
必要資源数 ≦ 空き資源数

最後にプロセス3に必要な資源を全て渡す

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	5	2	2	4

資源確保

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	0,0	0,0	0,0	0,0
3	1,0	2,0	5,0	1,0

終了

終了

プロセス3は資源を全て得たので実行

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	6	4	7	5

資源解放

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	0,0	0,0	0,0	0,0
3	0,0	0,0	0,0	0,0

終了

終了

終了

プロセス3は実行後資源を返却する

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4

プロセス1→2→3 の順序で実行すれば
全て実行できる

(全て資源数)

U,R

プロセス			3	4
1	1,2	2,1	2,0	0,1
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	0,1

＝ 現在は安全な状態

(保有資源数, 残り必要資源数)

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	3	1	**0**	2

資源割り当て数)

U,R

資源 プロセス	1	2	3	4
1	1,2	2,1	2,0	0,1
2	1,0	0,3	3,0	3,2
3	1,0	1,1	**2,3**	0,1

ここでプロセス3が資源3を1個要求すると？

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	3	1	**0**	2

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	1,2	2,1	2,0	0,1
2	1,0	0,**3**	3,0	3,2
3	1,0	1,1	**2,3**	0,1

実行可能

資源2が不足

資源3が不足

(保有資源数, 残り必要資源数)

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	4	3	2	2

資源解放

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	1,0	0,3	3,0	3,2
3	1,0	1,1	**2,3**	0,1

終了

まずプロセス1が実行, 実行後資源解放

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	4	3	2	2

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	1,0	0,3	3,0	3,2
3	1,0	1,1	**2,3**	0,1

終了

実行可能

資源3が不足

(保有資源数, 残り必要資源数)

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	5	3	5	5

(空き資源数)

資源解放

U,R

プロセス＼資源	1	2	3	4
1	0,0	0,0	0,0	0,0
2	0,0	0,0	0,0	0,0
3	1,0	1,1	**2,3**	0,1

終了

終了

次にプロセス2が実行, 実行後資源解放

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	5	3	5	5

(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	0,0	0,0	0,0	0,0
3	1,0	1,1	**2,3**	0,1

終了

終了

実行可能

(保有資源数, 残り必要資源数)

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	6	4	7	5

資源解放　　(空き資源数)

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	0,0	0,0	0,0	0,0
3	0,0	0,0	0,0	0,0

終了

終了

終了

最後にプロセス3が実行, 実行後資源解放

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4

プロセス1→2→3 の順序で実行すれば
全て実行できる

資源割り当て数)

U,R

プロセス				4
2	1,0	0,3	3,0	3,2
3	1,0	1,1	**2,3**	0,1

= 割り当て後も安全な状態

プロセス3への資源3割り当てを**許可**

プロセス3が資源3を1個要求

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	3	1	1	**1**

資源割り当て

U,R

資源 プロセス	1	2	3	4
1	1,2	2,1	2,0	0,1
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	**1,0**

ここでプロセス3が資源4を1個要求すると？

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	3	1	1	**1**

U,R

資源 プロセス	1	2	3	4
1	1,2	2,1	2,0	0,1
2	1,0	0,**3**	3,0	3,**2**
3	1,0	1,1	1,**4**	**1,0**

実行可能

資源2,4が不足

資源3が不足

(保有資源数, 残り必要資源数)

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
数	4	3	3	1

資源解放

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	**1,0**

終了

まずプロセス1が実行, 実行後資源解放

銀行家のアルゴリズム

例：3プロセス 4資源の場合

F

資源	1	2	3	4
数	4	3	3	1

U,R

資源 プロセス	1	2	3	4
1	0,0	0,0	0,0	0,0
2	1,0	0,3	3,0	3,**2**
3	1,0	1,1	1,**4**	**1,0**

終了

資源4が不足

資源3が不足

プロセス2,3共に実行不可能

銀行家のアルゴリズム

例 : 3プロセス 4資源の場合

F

資源	1	2	3	4
				1

プロセス2,3が実行不可能になる

資源割り当て

U,R

= 割り当て後は安全な状態ではない

プロセス				
2	1,0	0,3	3,0	3,2
3	1,0	1,1	1,4	**1,0**

プロセス3への資源4割り当てを**拒絶**

プロセス3が資源4を1個要求

銀行家のアルゴリズム

- デットロックの回避アルゴリズム
  - 長所
    - デッドロックの防止よりも柔軟
  - 短所
    - 資源を得る前にチェックが必要
      - プロセス数，資源数が多いと計算が増える
    - 必要な資源の最大数の予測が必要
      - 必要資源数が予測できないと使えない

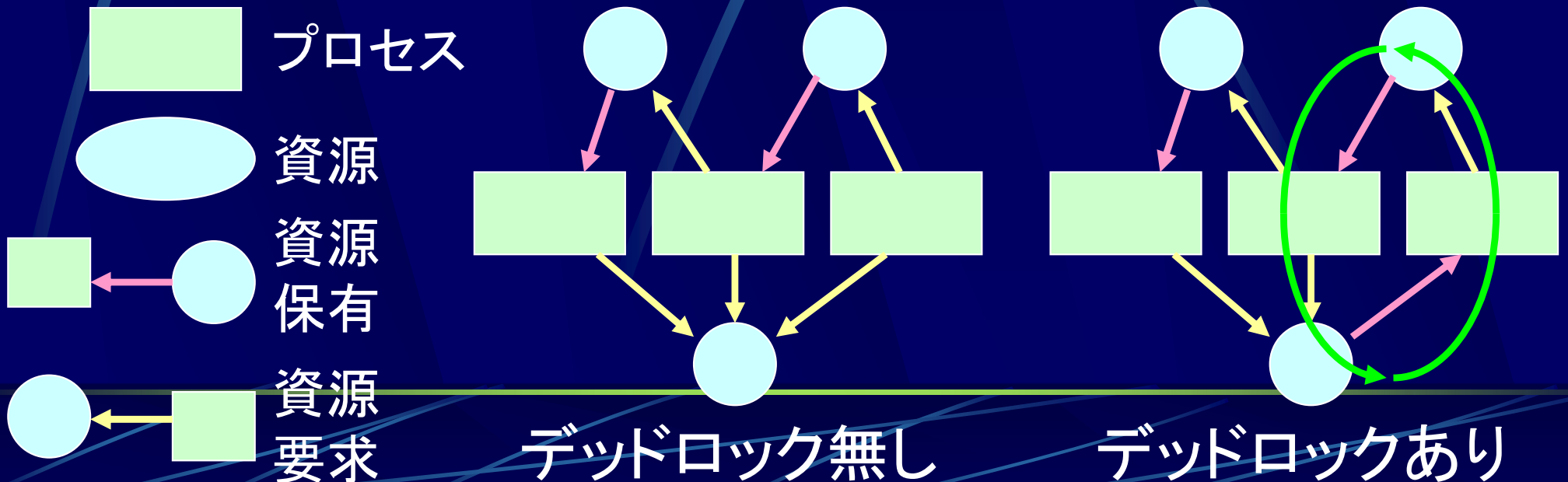
デッドロックの防止・回避の長所と短所

	長所	短所
防止	• 判定が簡単	• 資源割り当てが許可される条件が厳しい
回避	• デッドロックの防止よりも柔軟に対応可能	• 判定に時間がかかる • 必要な資源の最大数の予測が必要

デッドロックの検出
(deadlock detection)

- デッドロック検出(deadlock detection)
  - デッドロックの発生を検知
  - デッドロックに関係するプロセス，資源を特定

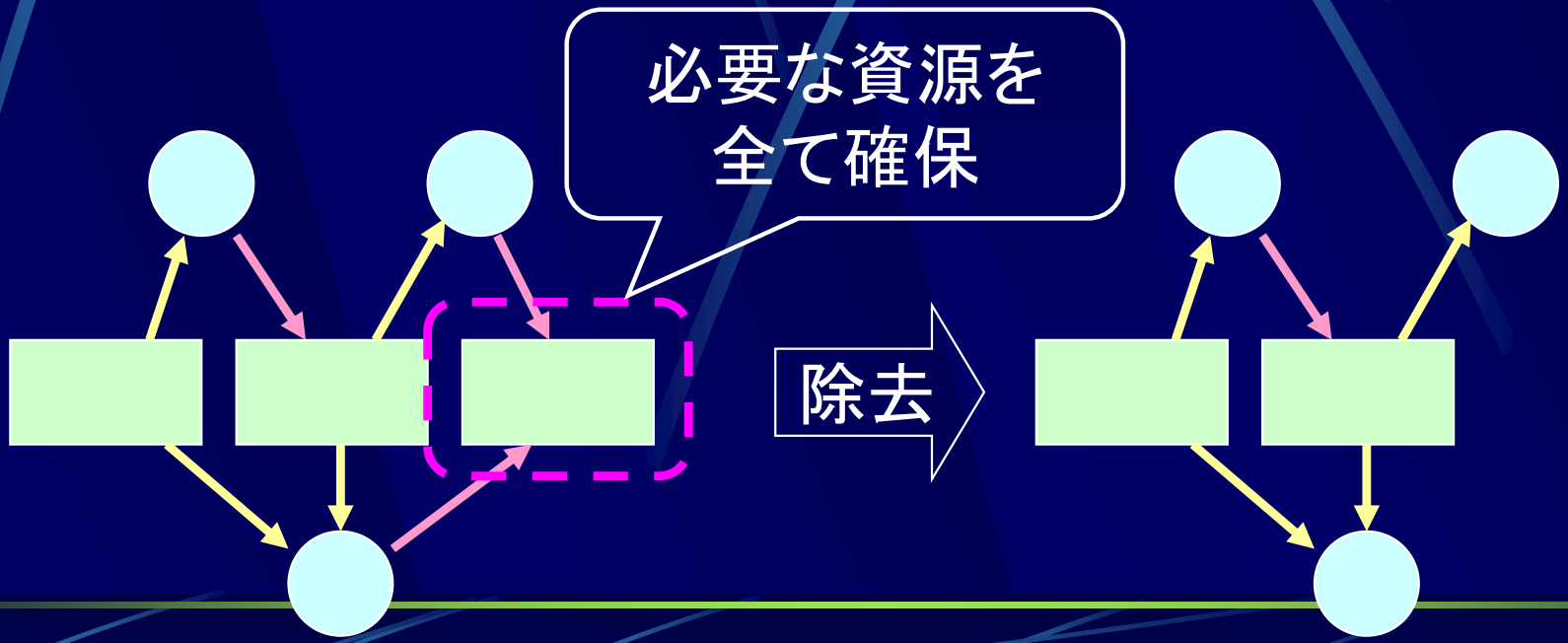
資源割り付けグラフのループを探す



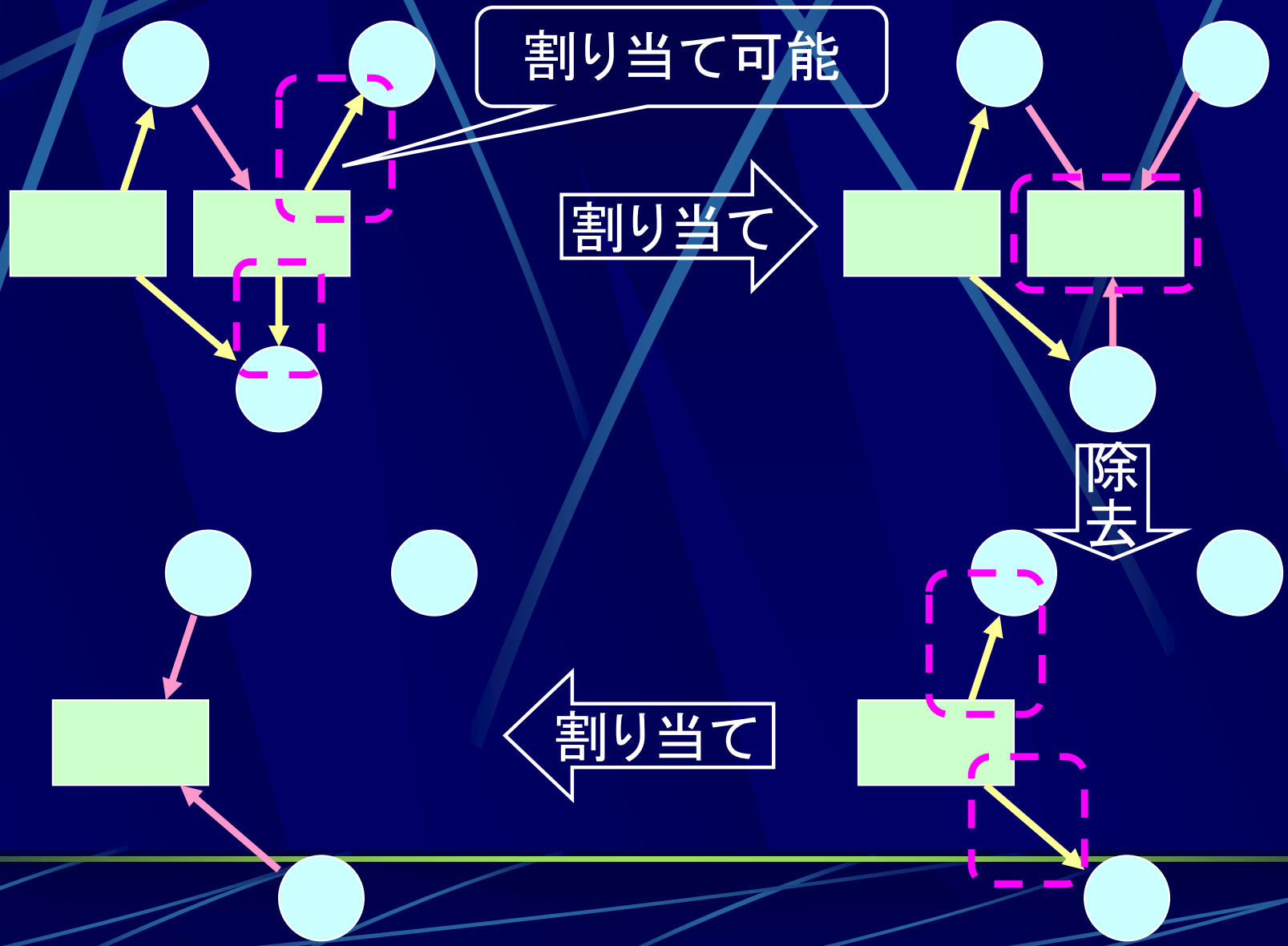
デッドロックの検出

- 簡約可能(reducible)
  - あるプロセスが必要な資源を全て得ている

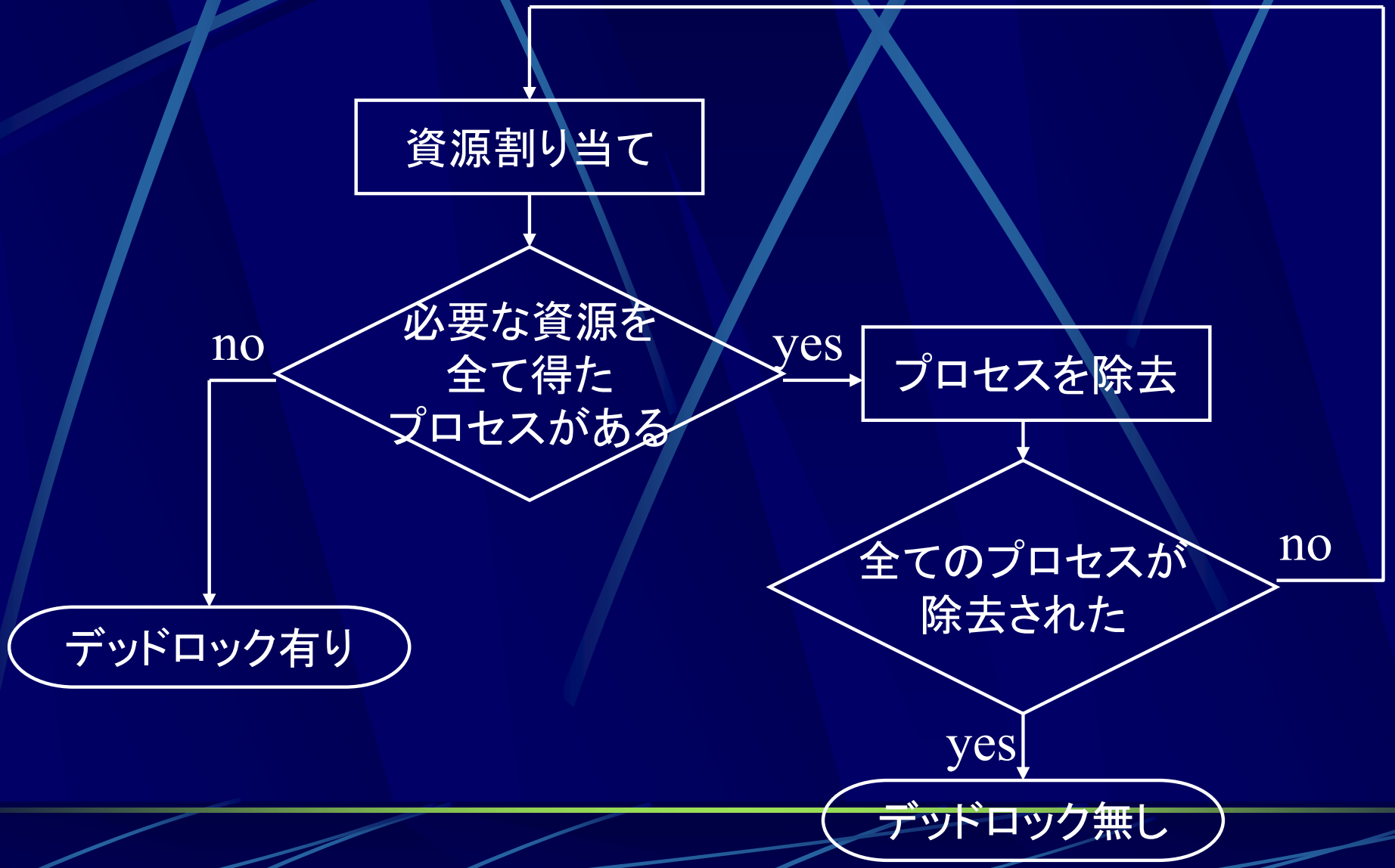
⇒そのプロセスは資源割り付けグラフから除去



デッドロックの検出



デッドロックの検出

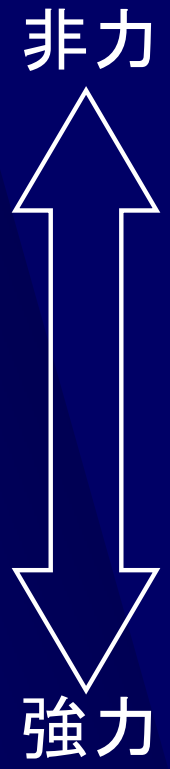

資源割り当て
必要な資源を
全て得た
プロセスがある
no
yes
プロセスを除去
全てのプロセスが
除去された
no
yes
デッドロック有り
デッドロック無し

デッドロックの検出

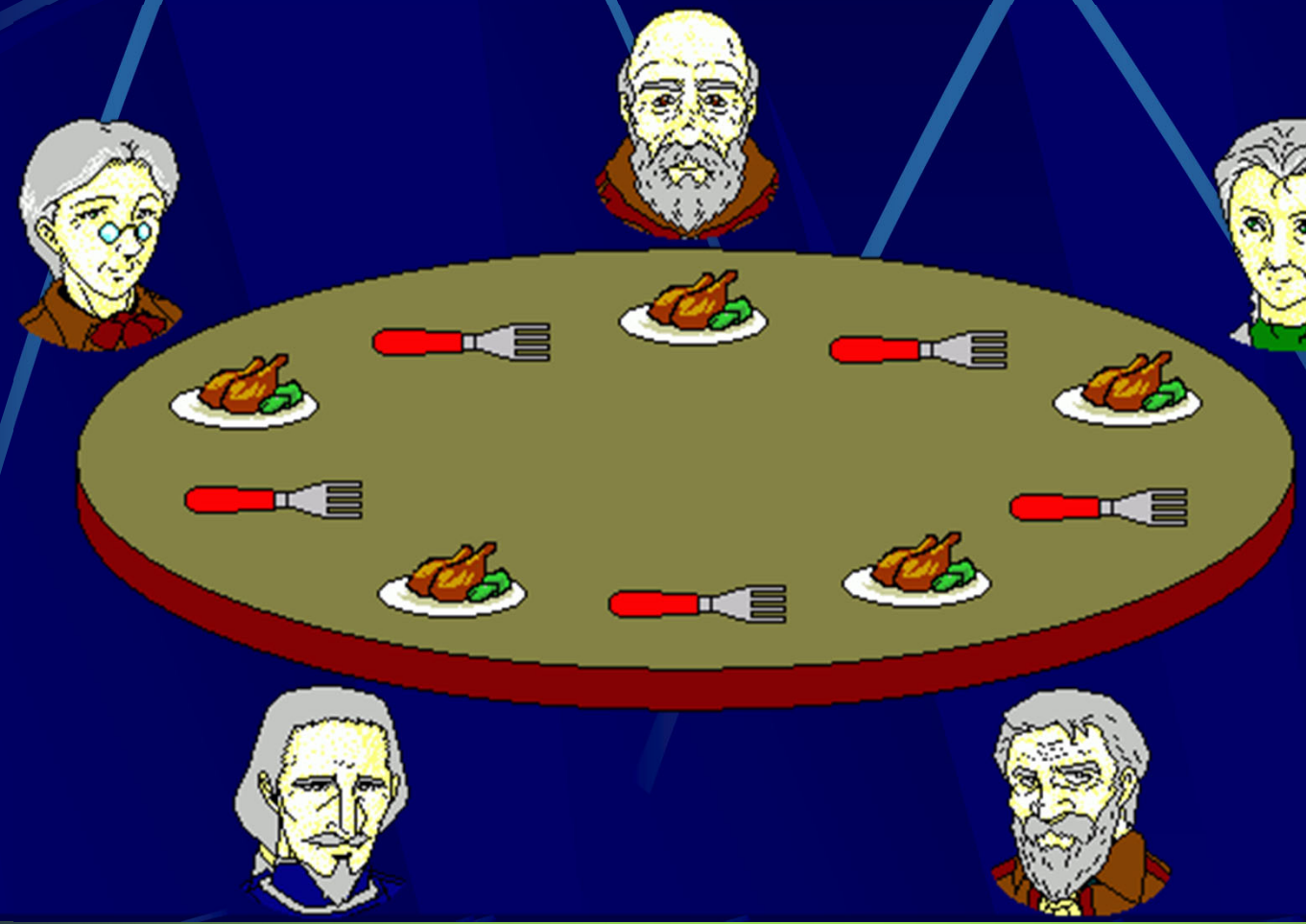
- デッドロック検出した場合
  - デッドロックの通知(deadlock notification)
    - 管理者にデッドロック発生を通知
  - デッドロックの回復(deadlock recovery)
    - デッドロック状態にあるプロセスの1つを終了(異常終了)させる
    - デッドロック状態にあるプロセスの1つを資源獲得前の状態に戻す

デッドロックへの対処法(再掲)

- 無策
- デッドロックの検出(deadlock detection)
  - デッドロックの通知(deadlock notification)
  - デッドロックの回復(deadlock recovery)
- デッドロックの回避(deadlock avoidance)
- デッドロックの防止(deadlock prevention)



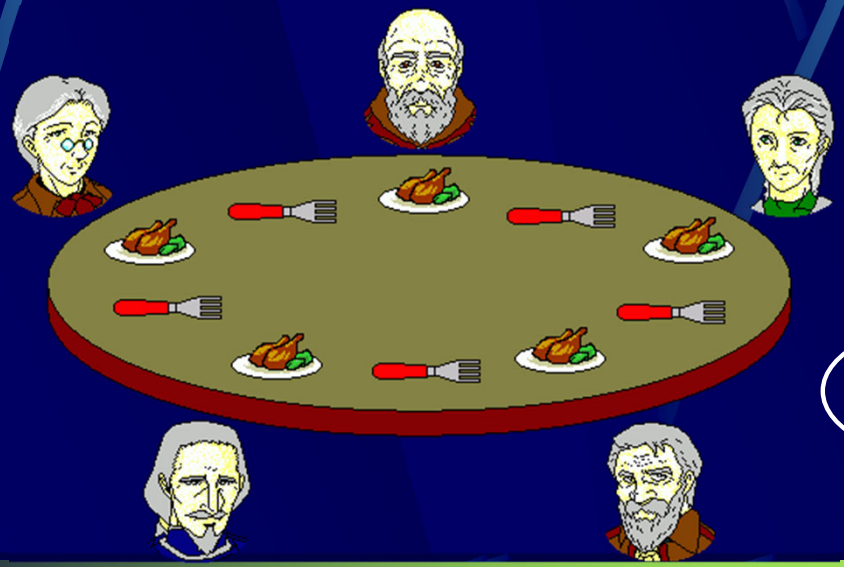
食事をする哲学者問題
Dining Philosophers Problem

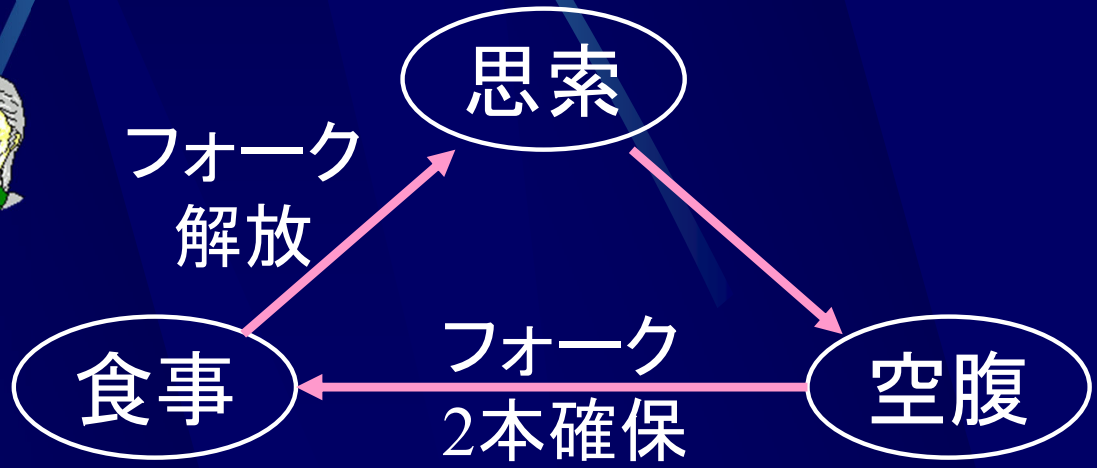


5人の哲学者
5人分の料理
5本のフォーク

食事をする哲学者問題

- 食事をする哲学者の問題
  - 哲学者の状態は 思索→空腹→食事→思索
  - 空腹時に両手にフォークがあれば食事可能
  - 空腹のまま長時間待たされると餓死

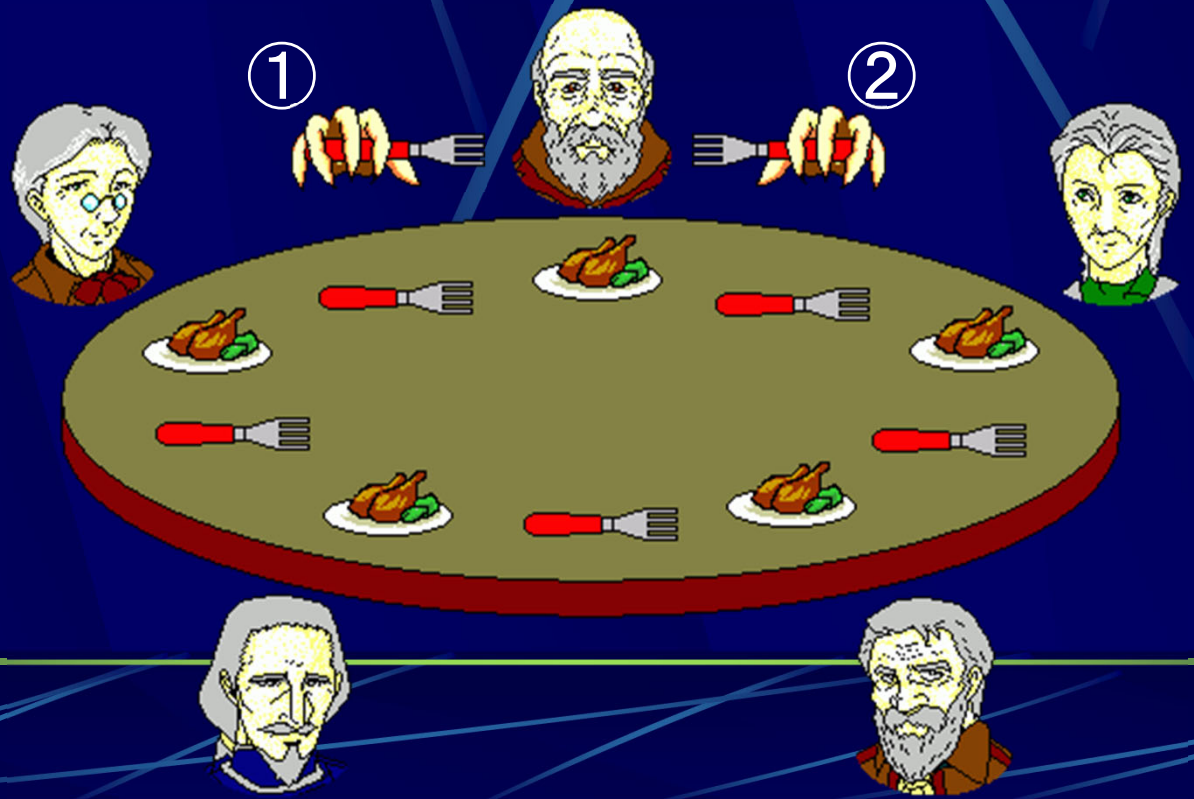




全ての哲学者を餓死させないためには？

食事をする哲学者問題

- 各フォークをセマフォで管理する
  - 空腹時はフォークを 右→左 の順に確保
  - 食後はフォークを 左→右 の順に解放



食事をする哲学者問題

広域変数と初期値

```
semaphore fork[] := {1, 1, 1, 1, 1};
```

哲学者 i のアルゴリズム

```
wait ( fork [i]);                          /* 右のフォークを確保 */
wait ( fork [(i+1)  mod  5]);              /* 左のフォークを確保 */
食事;
signal ( fork [(i+1)  mod  5]);            /* 左のフォークを解放 */
signal ( fork [i]);                        /* 右のフォークを解放 */
思索;
```

これでうまくいく?

食事をする哲学者問題

- 相互排除条件
  - フォークは排他的資源
    - ⇒ 相互排除条件成立
- 待機条件
  - 哲学者はフォークを1本ずつ確保
    - ⇒ 待機条件成立
- 横取り不能条件
  - 哲学者は食事をするまでフォークを手放さない
    - ⇒ 横取り不能条件成立

循環待機条件は?

食事をする哲学者問題

フォーク：①,②,③,④,⑤

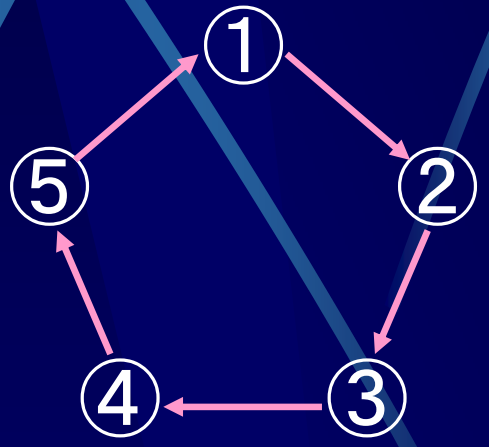
⑤ ① ⑤ → ①

① ② ① → ②

② ③ ② → ③

③ ④ ③ → ④

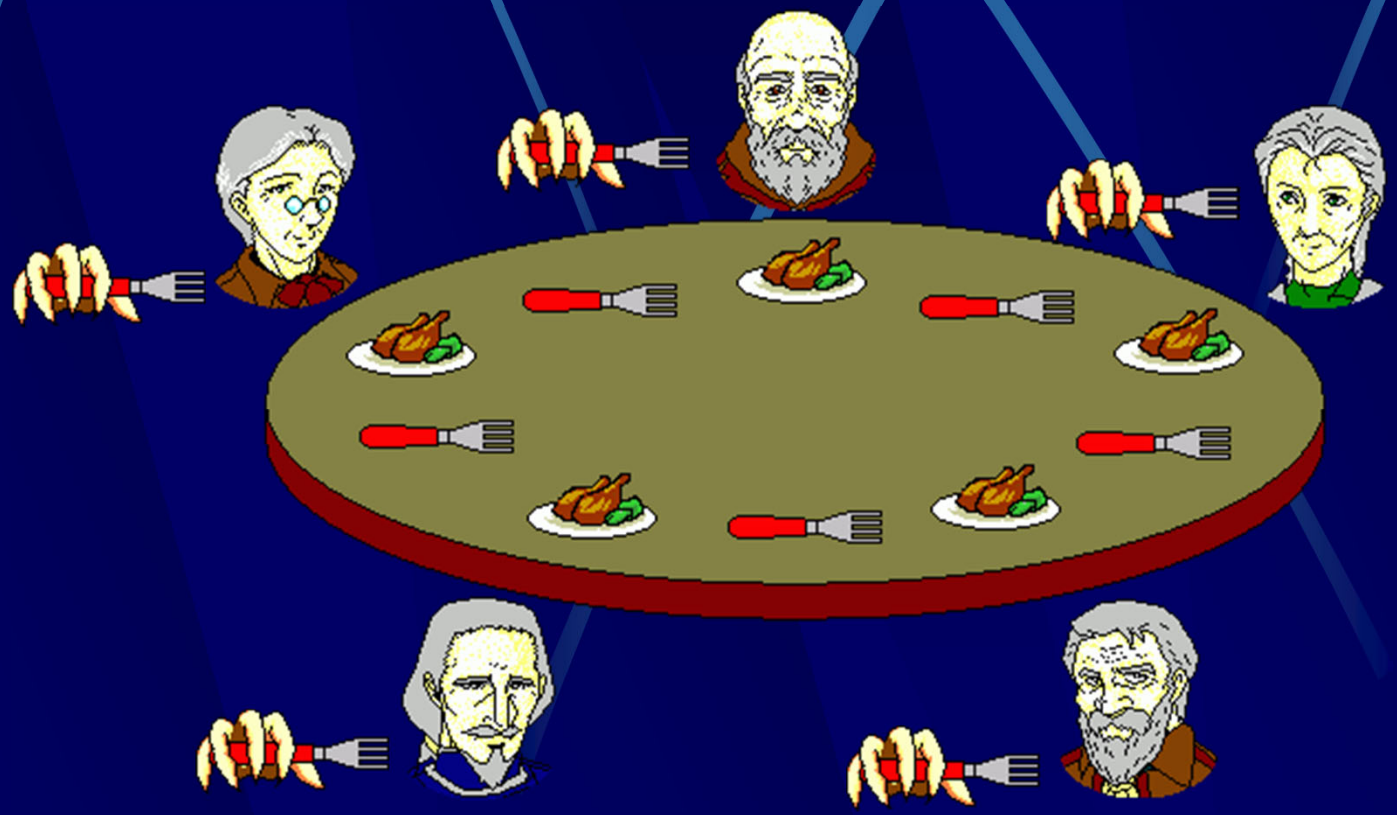
④ ⑤ ④ → ⑤



フォーク獲得順がループ

⇒循環待機条件成立

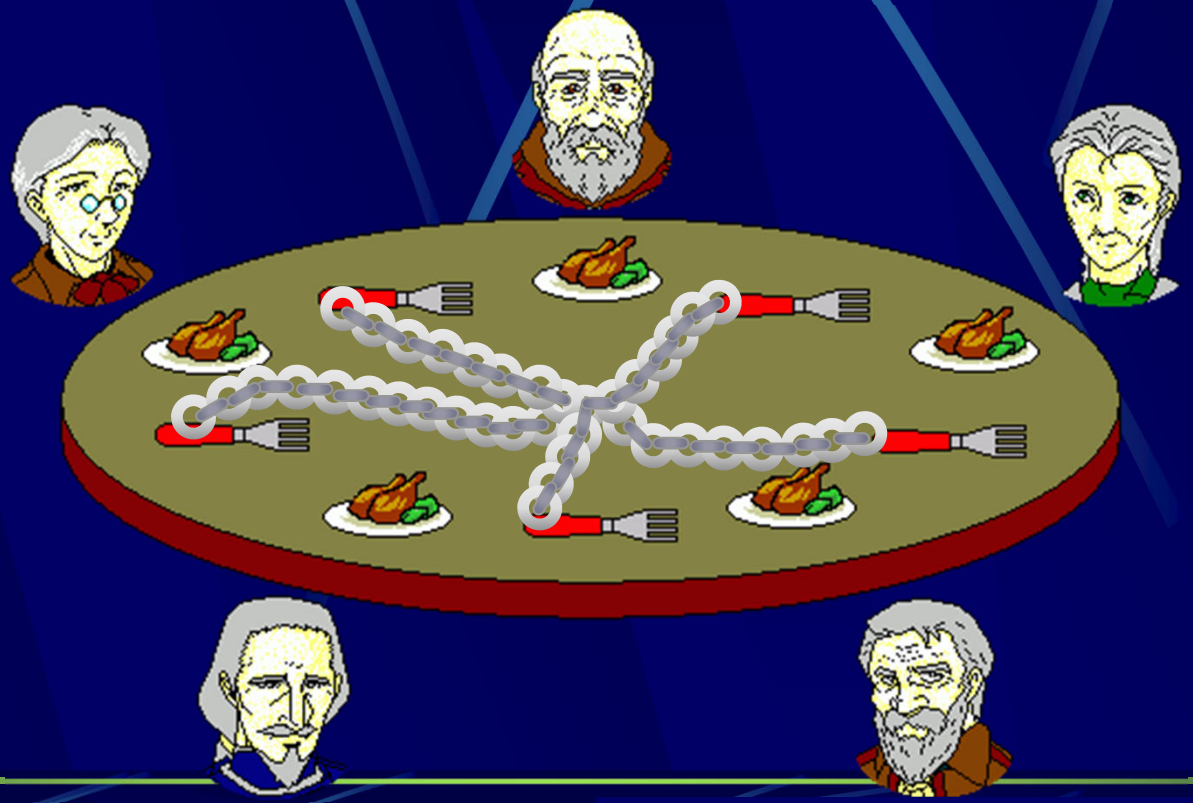
食事をする哲学者問題



全員が片方のフォークを持ったままデッドロックの可能性

食事をする哲学者問題

解法その1：繋がったフォーク
同時に全てのフォークを確保



⇒ 待機条件を回避

食事をする哲学者問題

解法その1：繋がったフォーク
フォーク全体を1つのセマフォで管理
広域変数と初期値

```
semaphore fork := 1;
```

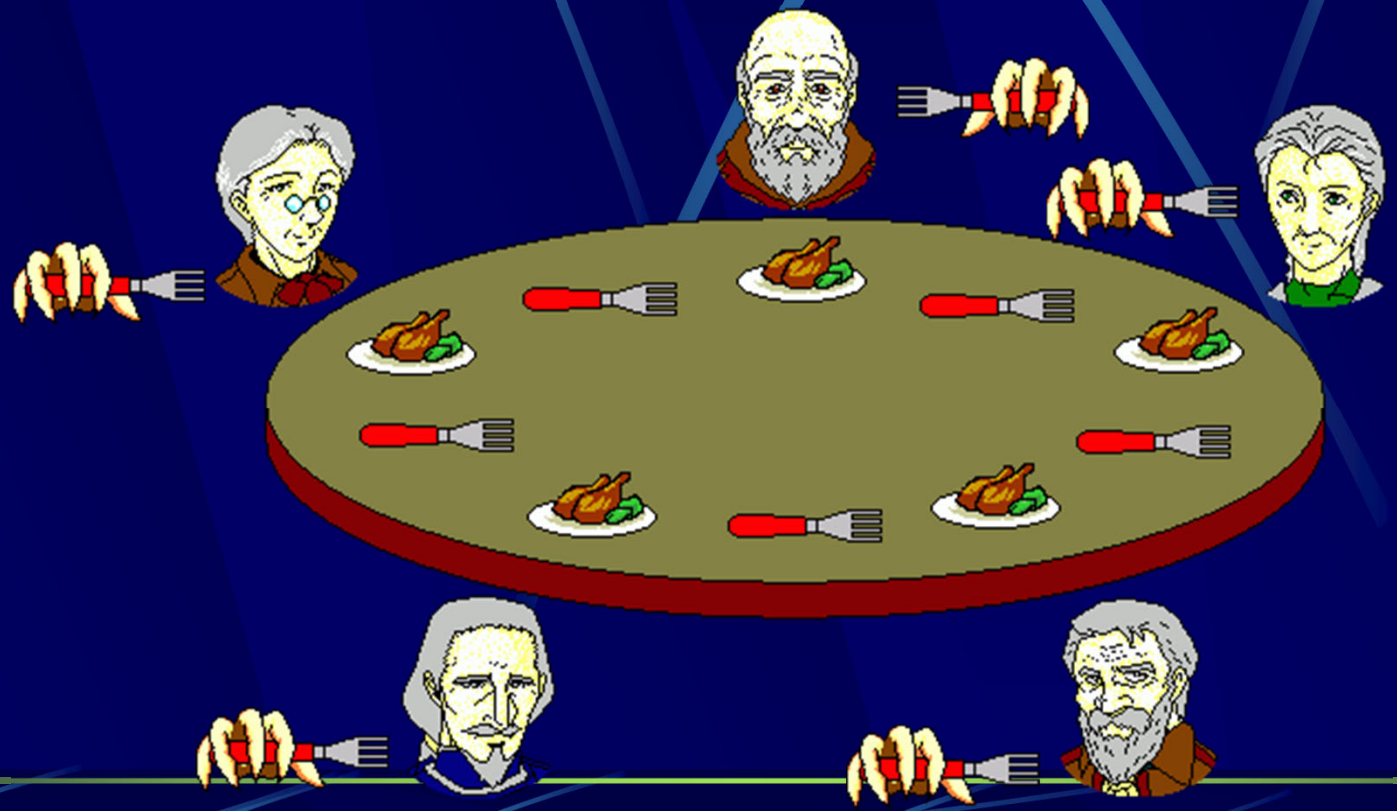
哲学者 *i* のアルゴリズム

```
wait ( fork );                    /* フォークを全て確保 */
食事;
signal ( fork );                  /* フォークを全て解放 */
思索;
```

デッドロックはしないが同時に1人しか食事できない

食事をする哲学者問題

解法その2：左利きの哲学者
1人だけフォークを逆順で確保



食事をする哲学者問題

```
if (i = 0) {
   wait ( fork [(i+1)  mod  5]);   /* 左のフォークを確保 */
   wait ( fork [i]);               /* 右のフォークを確保 */
}  else {
   wait ( fork [i]);               /* 右のフォークを確保 */
   wait ( fork [(i+1)  mod  5]);   /* 左のフォークを確保 */
}
食事;
signal ( fork [(i+1)  mod  5]);  /* 左のフォークを解放 */
signal ( fork [i]);              /* 右のフォークを解放 */
思索;
```

食事をする哲学者問題

フォーク：①,②,③,④,⑤

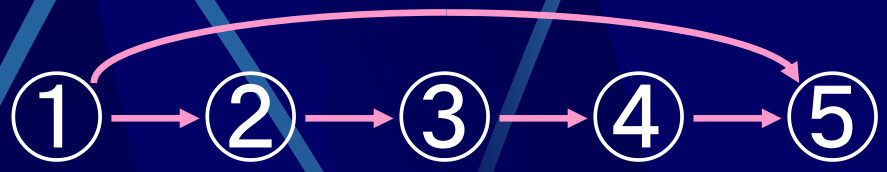
⑤ ① ⑤ ← ①

① ② ① → ②

② ③ ② → ③

③ ④ ③ → ④

④ ⑤ ④ → ⑤



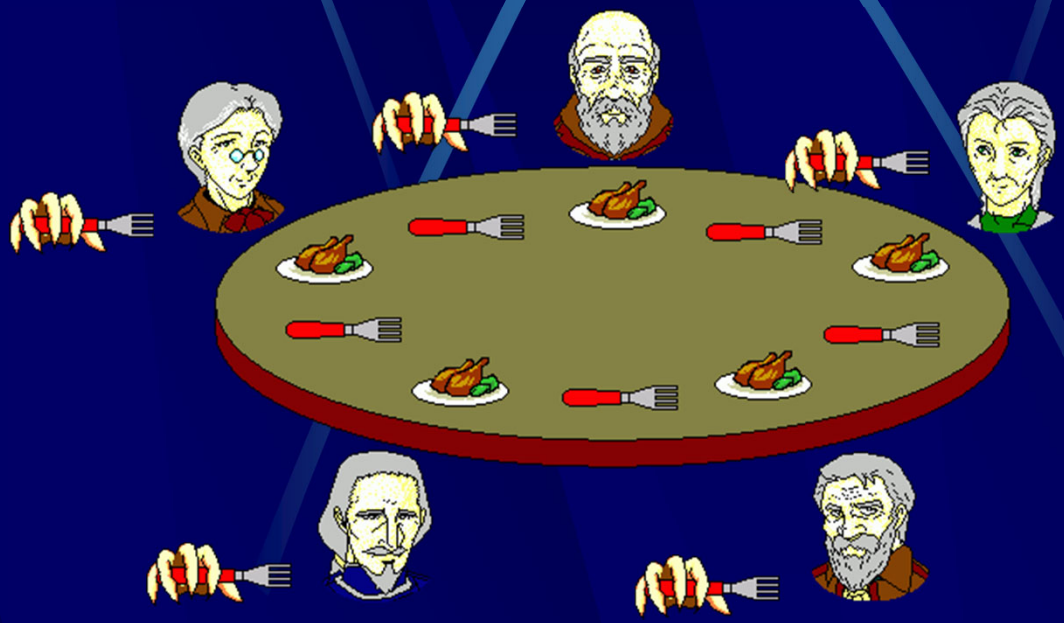
フォーク獲得順に
ループが無い

⇒ 循環待機条件を回避

デッドロックはしないが
公平性を欠く可能性がある

食事をする哲学者問題

解法その3：我慢する哲学者
　右のフォークを確保後、左をフォークを確保できなければ一旦右のフォークを放し、少し待つ



「全員が右フォークを確保したまま」の状態からは抜け出せる

⇒ 横取り不能条件を回避

食事をする哲学者問題

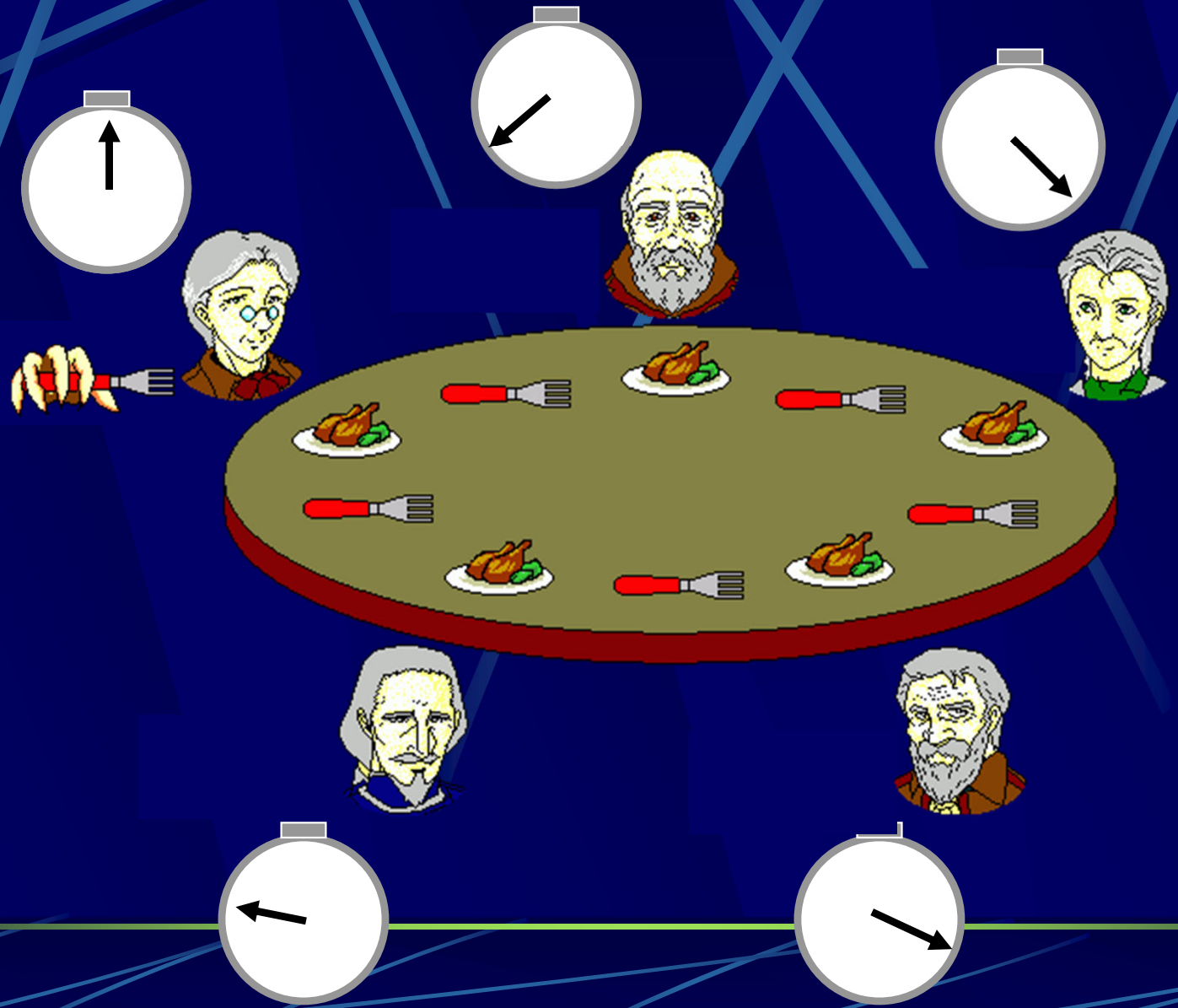
解法その3：我慢する哲学者
　右のフォークを確保後、左をフォークを確保できなければ一旦右のフォークを放し、少し待つ

全員の待ち時間が同じだとが
全員が「右フォーク確保」→「右フォーク解放」を繰り返す可能性がある

待ち時間をランダムに設定する

c.f. イーサネット技術 CSMA/CD

食事をする哲学者問題



try&wait 命令

- wait 命令
  - 資源を獲得できないときは待ち状態に
  - 資源の有無の確認ができない
- try&wait 命令
  - 資源を獲得できれば true を、できなければ false を返す

Java で実装

```
if ( s > 0 ) {
    s := s - 1;
    return true;
} else {
    return false;
}
```

食事をする哲学者問題

```
wait ( fork [i]);                          /* 右のフォークを確保 */
while (try&wait ( fork [(i+1)  mod  5]) = false) {
                          /* 左のフォークを確保できるまで繰り返す */
   signal (fork [i]);                      /* 右のフォークを一旦解放 */
   少し待つ;
   wait ( fork [i]);                       /* 右のフォークを再確保 */
}
食事;
signal ( fork [(i+1)  mod  5]);  /* 左のフォークを解放 */
signal ( fork [i]);                        /* 右のフォークを解放 */
思索;
```

Java でのセマフォ使用

- Semaphore クラスを使用
  - java.util.concurrent.Semaphore

コンストラクタ

```
public Semaphore (int permit)   /* permit : 資源数 */
```

wait 命令

```
public void acquire ()
        throws InterruptedExeption
```

signal 命令

```
public void release ()
```

try&wait 命令

```
public boolean tryAcquire ()
```

参考：食事をする哲学者プログラム(java)

- DiningPhilosophers.java
  - 食事をする哲学者の行動をシミュレート
  - 引数により哲学者の行動型を指定

1: 繋がったフォーク　　2: 左利きの哲学者
3: 我慢する哲学者　　それ以外: デッドロックに無策な哲学者

http://www.info.kindai.ac.jp/OS
からダウンロードし、各自実行してみること

参考：食事をする哲学者プログラム(java)

実行例

```
$ javac DiningPhilosophers.java
$ java DiningPhilosophers 3
  1 2 3 4 : get 0
  1   3 4 :
  1     4 :                    get 3
        4 :          get 1
          :                          get 4
0         : release 0
                                     get 0
```

引数で哲学者の行動型を指定

一旦フォークを解放

両方のフォークを確保