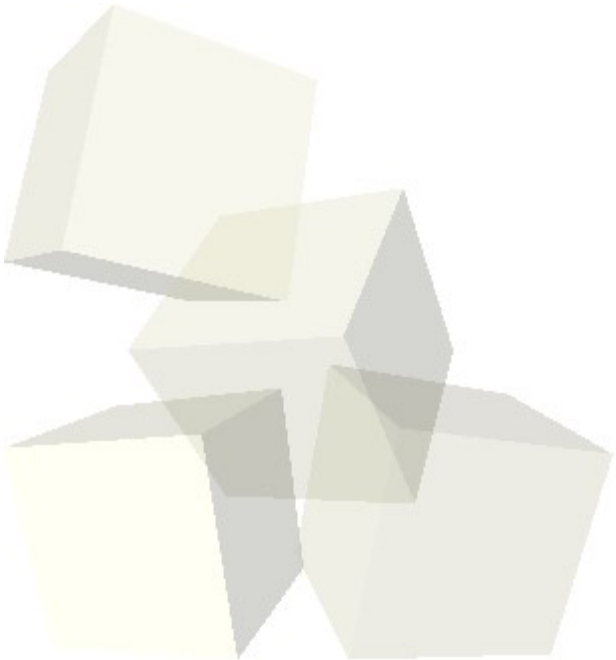
オペレーティングシステム

#5 並行プロセス：セマフォ



復習：排他制御

- クリティカルセクション
  - 共有リソースをプロセス同士が取り合う局面
  - リソース競合を解決する手法が必要

- 排他制御（mutex: MUTual EXclusion）
  - クリティカルセクションに同時に複数プロセスが入らないようにする制御

復習：排他制御

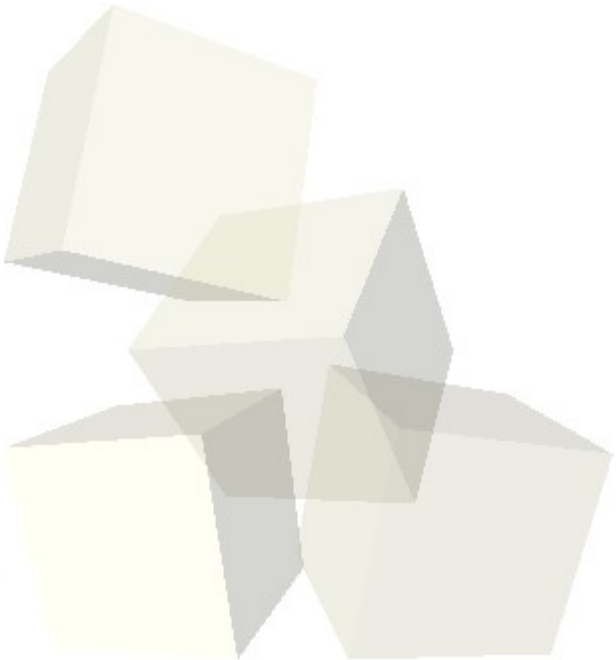
Dekkerのアルゴリズム

- ソフトウェアによる排他制御の基本手法
- 入る前に手を挙げる（Interest）
- 優先権により競合を解決（Priority）

問題点

- ユーザプログラムに依存
  - → ちゃんとプロセスが約束を守ってくれないと破綻
- ビジーウェイト（busy wait）
  - → 一方がクリティカルセクションを実行中，
  - → 待っている方は優先権をひたすらチェックし続ける
  - → CPUリソースの無駄

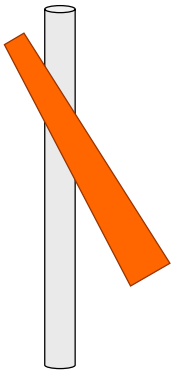
5.1
セマフォ構造体



- Semaphore（セマフォ）
  - プロセス間同期機構
  - Dijkstraにより提案

  - 「腕木式信号機」の意（オランダ語）

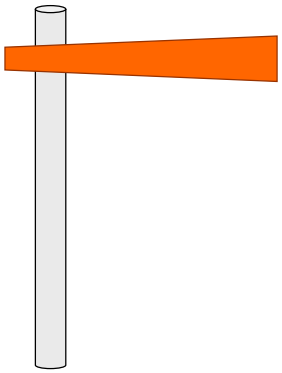
進め



Passeren

（英語のpassと同源）

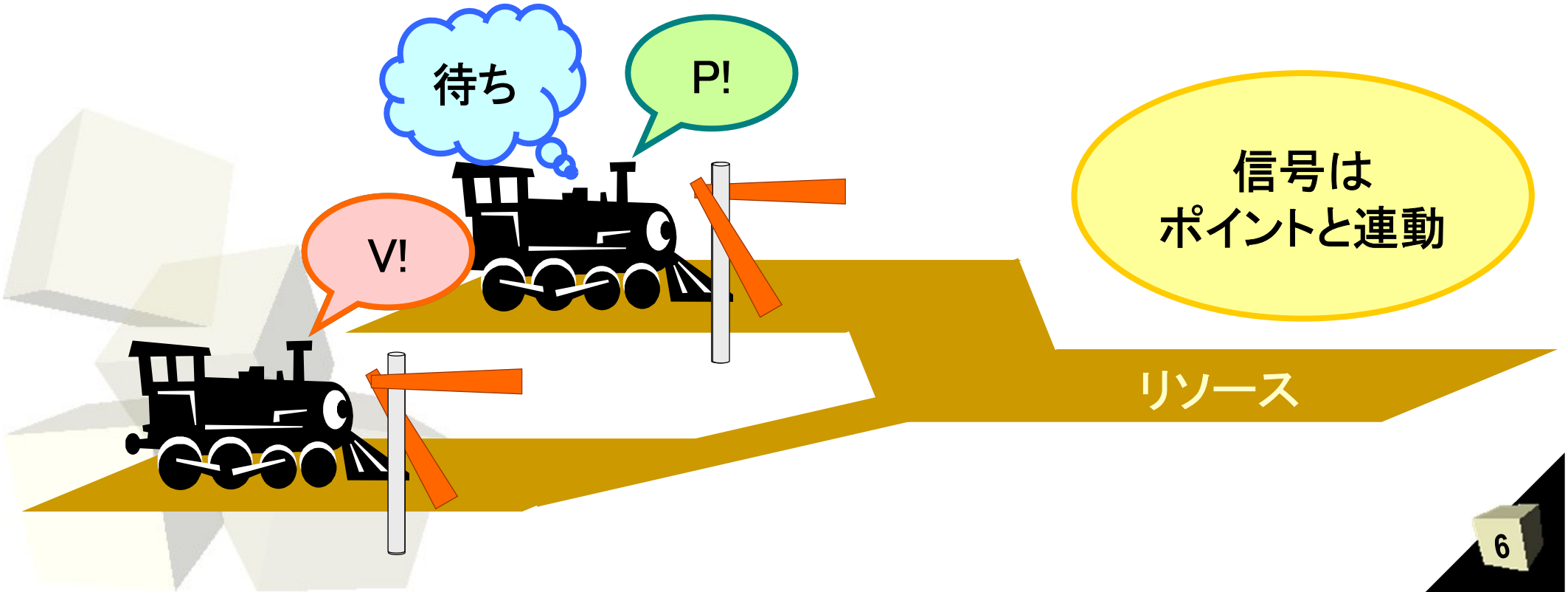
止まれ



Verhoog

セマフォに対する命令

- P命令
  - リソースを要求，許可されない場合は待ち状態へ移行
- V命令
  - リソースを解放，待ちプロセスを実行可能状態へ



P命令とV命令

P命令

- リソースを要求，許可されない場合は待ち状態へ移行

```
P(S){
    if ( S >= 1 ){
        S = S − 1;
     :
    }else{
        wait...
    }
}
```

V命令

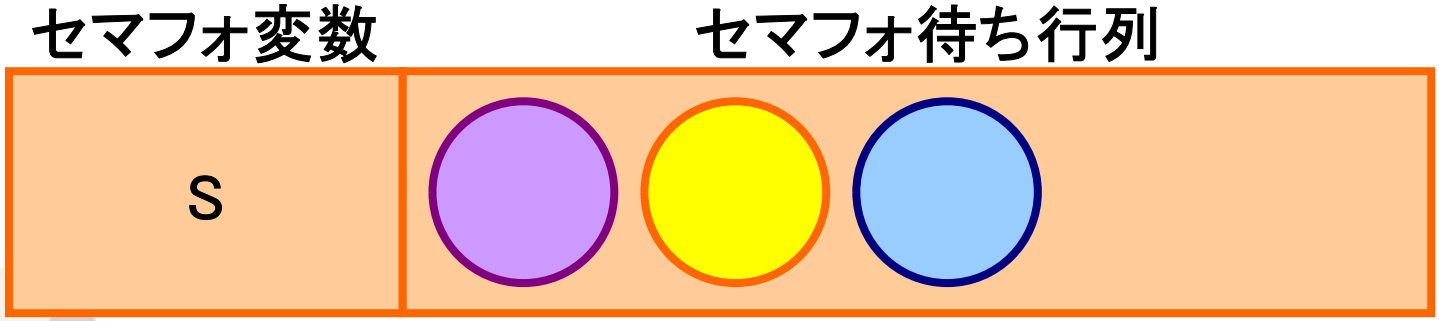
- リソースを解放

```
V(S){
    if ( len(S) >= 1 ){
        待ち行列中のプロセスを
       1つ実行可能状態へ;
    }else{
        S = S + 1;
     :
    }
}
```

これらの命令の実装はTS命令などが用いられている事に注意

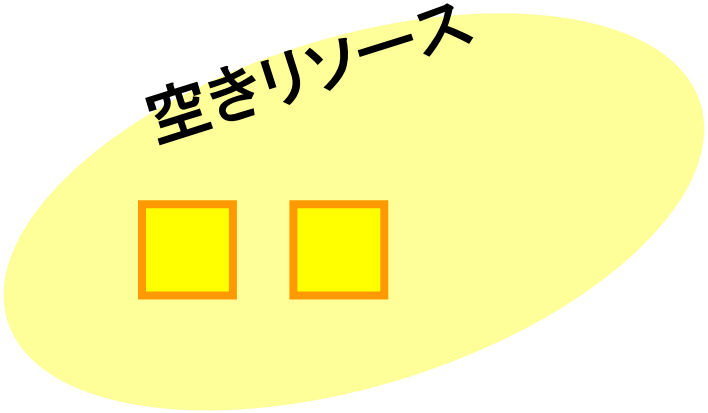
セマフォ構造体

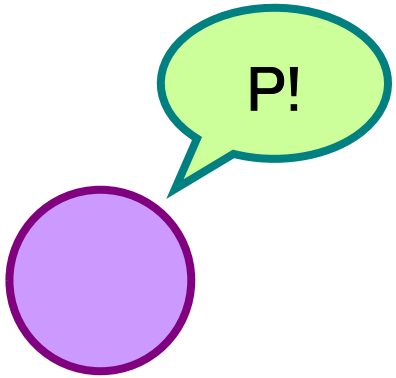
- セマフォ変数
  - リソースの空きを表現する変数
- セマフォ待ち行列
  - リソースの使用待ちプロセスの行列



P命令(その1)

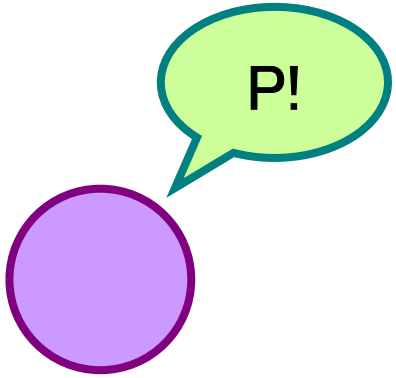
```
P(S){
    if ( S >= 1 ){
        S = S - 1;
     :
    }else{
        wait...
    }
}
```





P命令（その2）

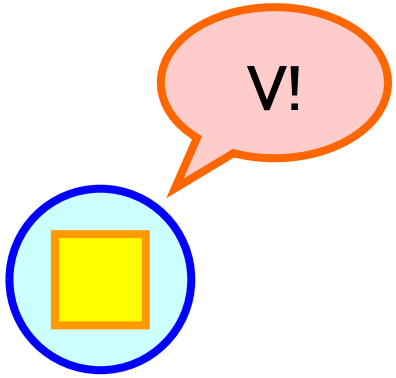
```
P(S){
    if ( S >= 1 ){
        S = S − 1;
     :
    }else{
        wait...
    }
}
```

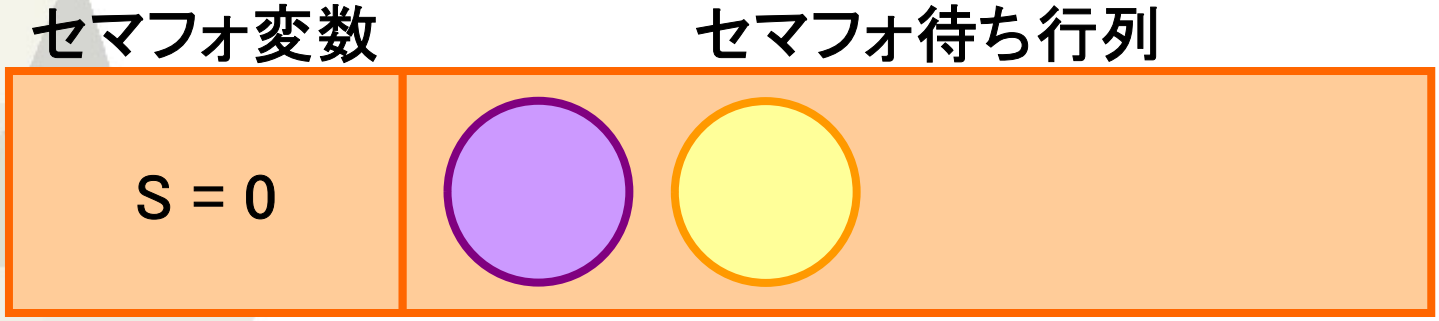


セマフォ変数	セマフォ待ち行列
S = 0	

V命令(その1)

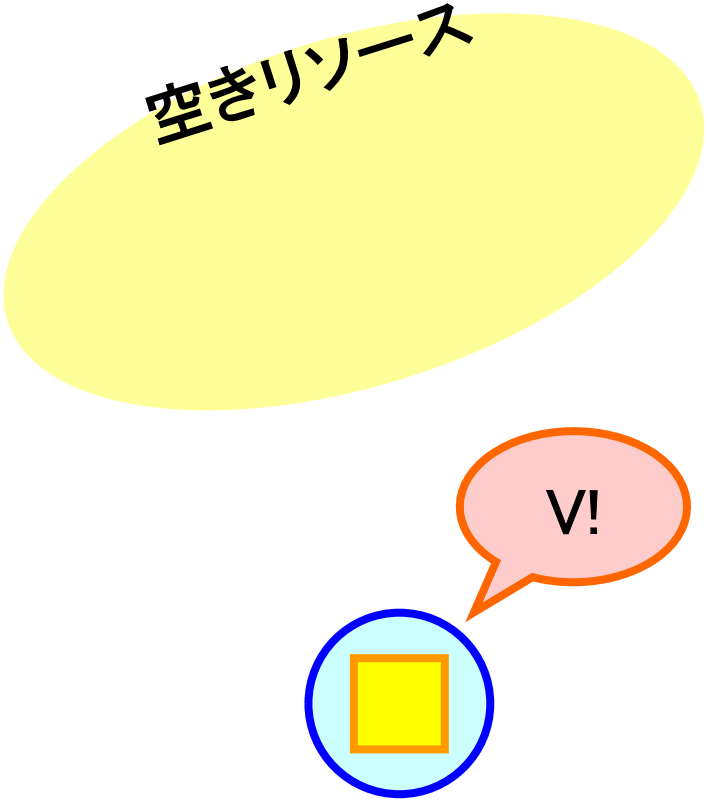
```
V(S){
    if ( len(S) >= 1 ){
        待ち行列中のプロセスを
      1つ実行可能状態へ;
    }else{
        S = S + 1;
     :
    }
}
```





V命令(その2)

```
V(S){
    if ( len(S) >= 1 ){
        待ち行列中のプロセスを
       1つ実行可能状態へ;
    }else{
        S = S + 1;
     :
    }
}
```



セマフォ変数	セマフォ待ち行列
S = 1	

まとめ：セマフォ

- P命令
  - 空きリソースを1つ使用
  - 空きリソース数（セマフォ変数）をデクリメント
  - 空きがない場合、プロセスを待ち状態に
- V命令
  - 空きリソースを1つ解放
  - 待ちプロセスを1つ実行可能状態に
  - 待ちプロセスがない場合、
    空きリソース数（セマフォ変数）をインクリメント

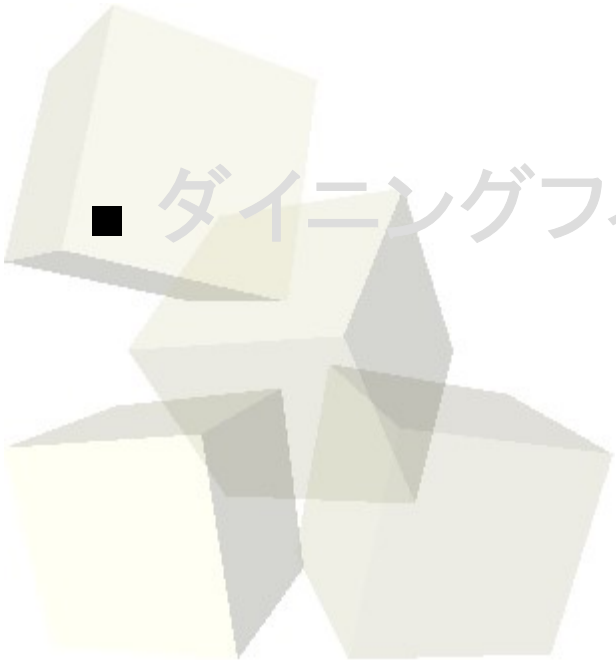
5.2
基本的なプロセス協調問題

セマフォを使った具体例

- 排他制御
- プロデューサコンシューマ
- リーダライタ
- ダイニングフィロソファ

セマフォを使った具体例

- 排他制御
- プロデューサコンシューマ
- リーダライタ
- ダイニングフィロソファ



排他制御

S = 1

P(S)
クリティカルセクション
V(S)

P(S)
クリティカルセクション
V(S)

復習：Dekkerのアルゴリズム

```
Interest[A] = TRUE;
while( Interest[B] ){
  if( Priority == B ){
    Interest[A] = FALSE;
    while( Priority == B ){};
    Interest[A] = TRUE;
  }
}

      :
クリティカルセクション
      :

Priority = B;
Interest[A] = FALSE;
```

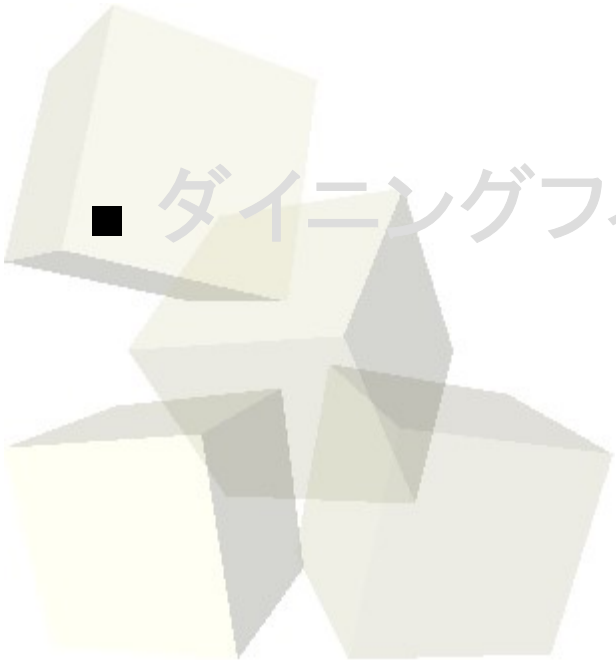
```
Interest[B] = TRUE;
while( Interest[A] ){
  if( Priority == A ){
    Interest[B] = FALSE;
    while( Priority == A ){};
    Interest[B] = TRUE;
  }
}

      :
クリティカルセクション
      :

Priority = A;
Interest[B] = FALSE;
```

セマフォを使った具体例

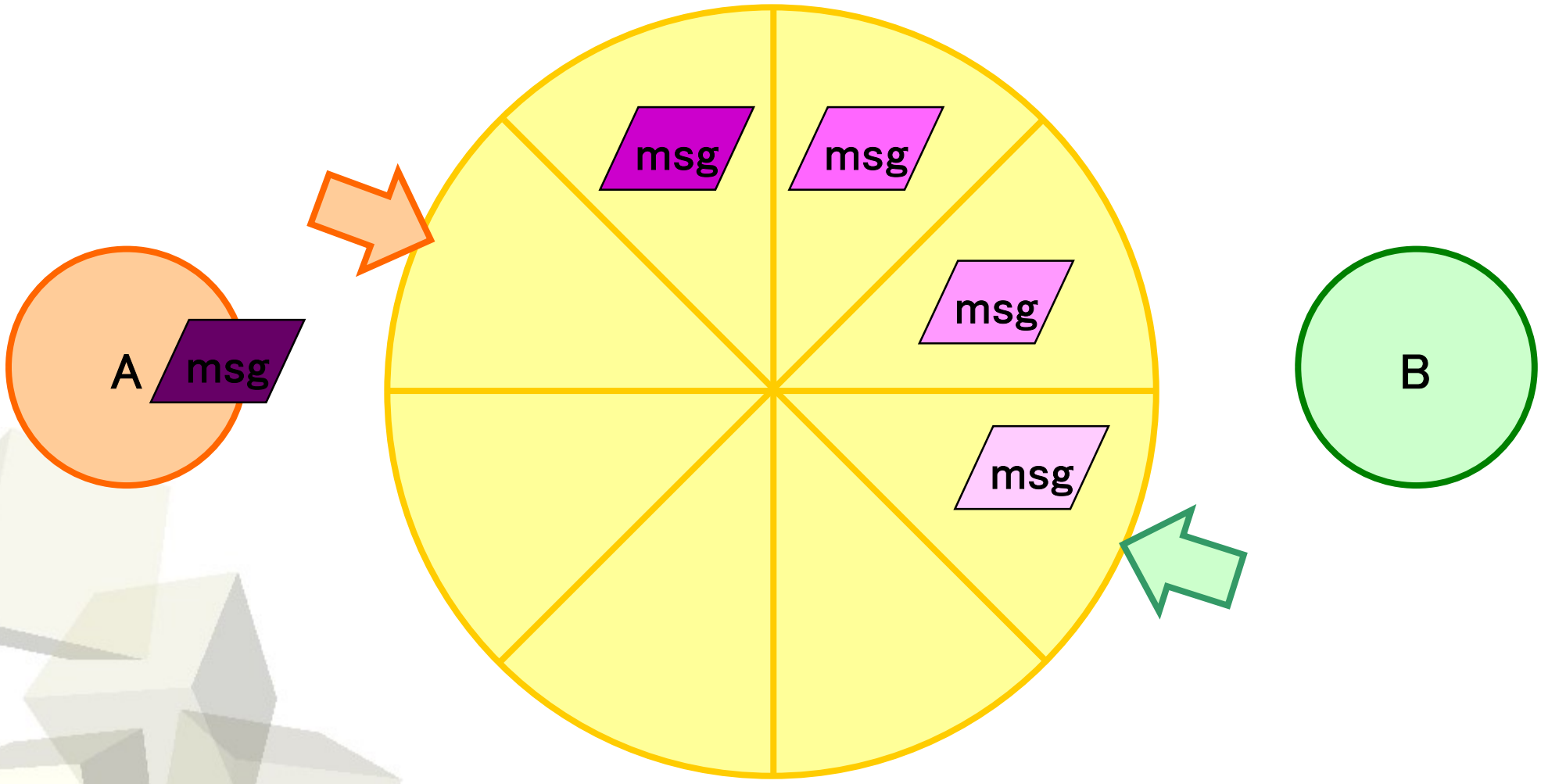
- 排他制御
- プロデューサコンシューマ
- リーダライタ
- ダイニングフィロソファ



プロデューサコンシューマ

- 送信側
  - 受信側の状態（受信可能か否か）は分からない
- 受信側
  - いつ送信されてくるか分からない
  - 通信を常時待つ必要 ⇒ ほかの処理ができない
- 送受信バッファ

参考：リングバッファ



セマフォによる プロデューサコンシューマ

```
Semaphor S = N;        // 空きバッファ数
Semaphor M = 0;        // メッセージ数
Messag
```

受信可能なメッセージが
あるか否かチェック

使用可能なバッファ数を
増やしたことを通知

送信待ちプロセスが存在する場合
そのプロセスを実行可能に

```
}
```

受信可能なメッセージ数を
増やしたことを通知

送信待ちプロセスが存在する場合、
そのプロセスを実行可能状態へ

```
        recv_msgの処理;
}
```

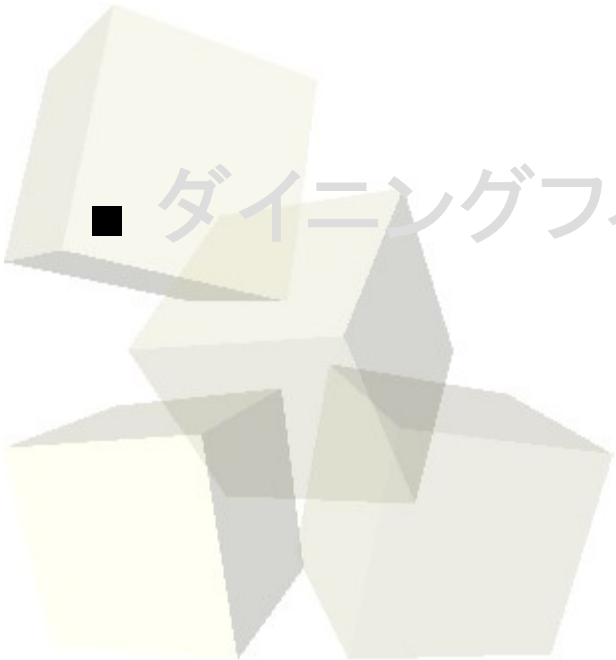
セマフォによる プロデューサコンシューマ

- リングバッファ

- セマフォ1：空きバッファ数
  - 送信側がメッセージをバッファに格納可能か
  - 格納不可の場合、送信側を待ち状態に
  - メッセージの上書きを回避
- セマフォ2：メッセージ数
  - 受信側が受信すべきメッセージがあるか
  - ない場合、受信側を待ち状態に
  - メッセージの重複受け取りを回避

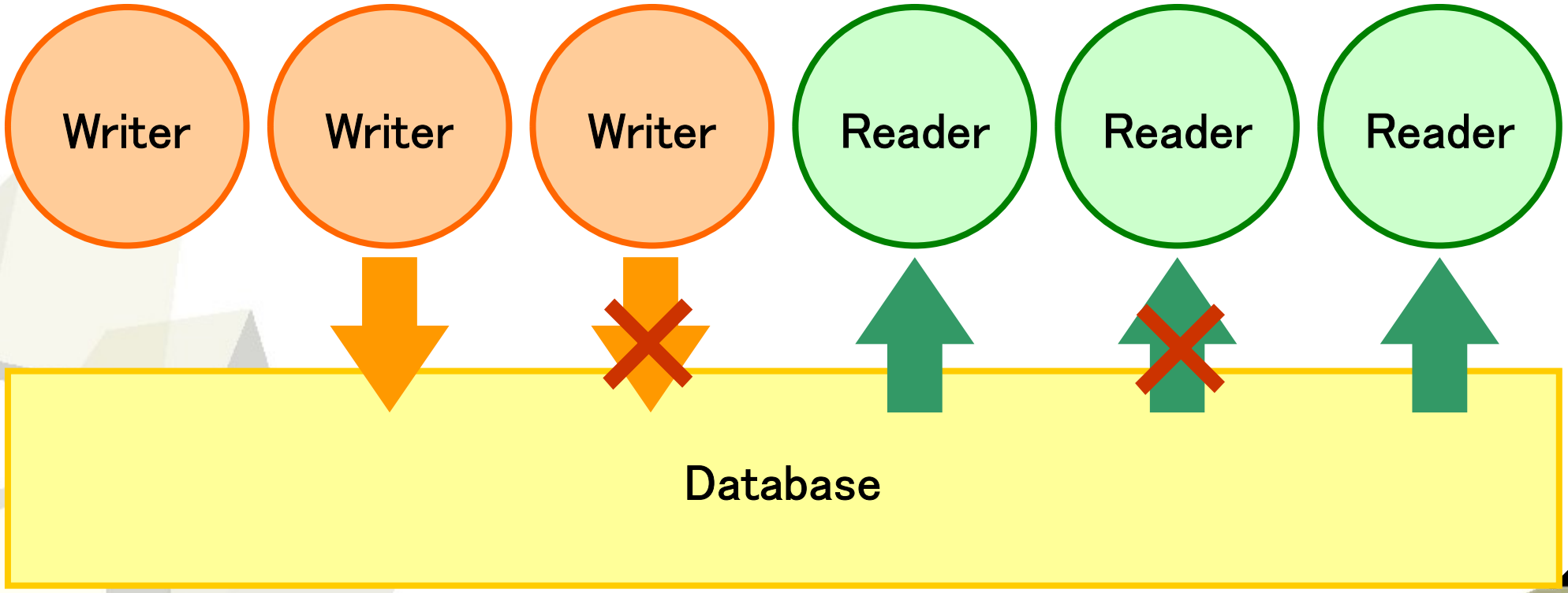
セマフォを使った具体例

- 排他制御
- プロデューサコンシューマ
- リーダライタ
- ダイニングフィロソファ



リーダライタ問題

- ライタによる書き込み中は読み出し不可
- 同時には1ライタのみ書き込み可
- 同時に複数リーダが読み出し可



セマフォによる リーダライタ

```
Semaphor W = 1;          // 書き込みプロセスの制御
Semaphor M = 1;          // RおよびWに対する操作を制御
int R = 0;               // 同時読み出しプロセス数
```

```
while(1){
        データ生
        P(W);
        書き
        V(W
}
```

読み出し中は
書き込みプロセスを
ブロック

変数Rの操作を
他のリーダプロセスと
排他制御

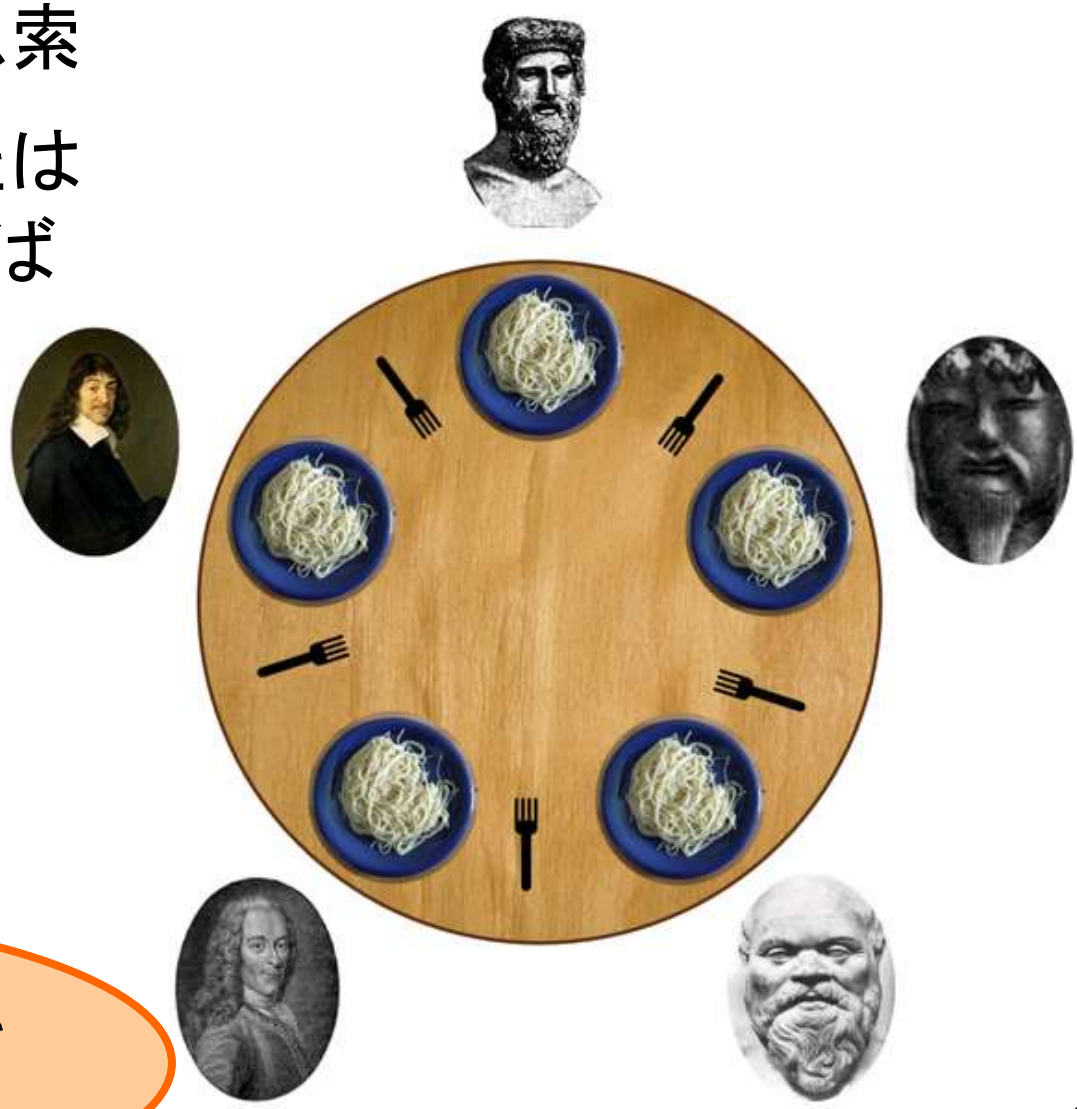
```
while(1){
        P(M);
        if( R == 0 ) P(W);
        R += 1;
        V(M);
        読み出し;
        P(M);
        R -= 1;
        if( R == 0 ) V(W);
        V(M);
}
```

セマフォを使った具体例

- 排他制御
- プロデューサコンシューマ
- リーダライタ
- ダイニングフィロソファ

Dining Philosophers

- 「食事をする哲学者」問題
  - 思索→空腹→食事→思索
  - 空腹時、両脇の箸（またはフォーク）が使用できれば食事可能
  - 空腹のまま長時間待たされると餓死
  - すべての哲学者を死なせない方法を考える問題

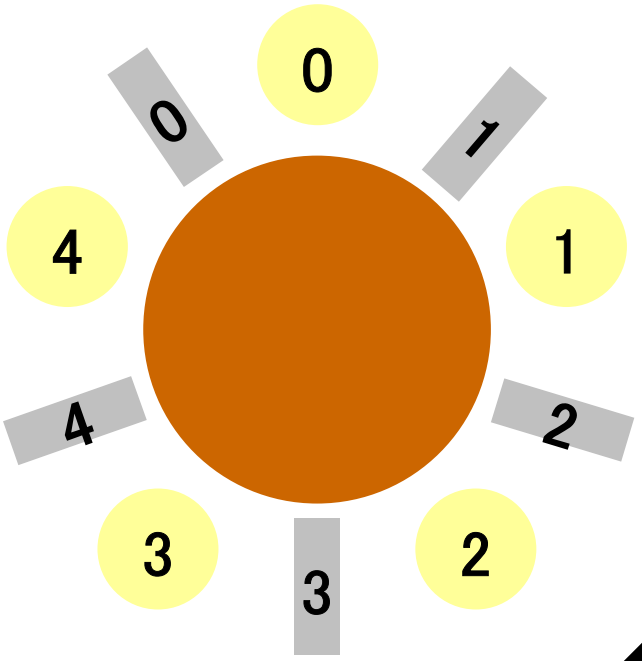


プロセスが複数リソースを要求する場合

うまくいかない例

- 各フォークをセマフォで管理
  - 右のフォークを確保し、左のフォークを確保
  - 食事後、左のフォークを解放し、右のフォークを解放

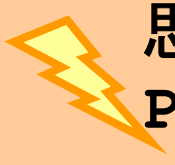
```
Semaphore fork[5] = {1,1,1,1,1};
philosopher( i ){
   while(1){
      思索;
      P( fork[i] );                    //右
      P( fork[ (i+1) mod 5 ] );        //左
      食事;
      V( fork[ (i+1) mod 5 ] );        //左
      V( fork[i] );                    //右
   }
}
```



うまくいかない例

- 一方のフォークを確保した段階で中断する可能性
  - 全員が片方のフォークを持ったまま動けなくなることがある

```
Semaphore fork[5] = {1,1,1,1,1};
philosopher( i ){
  while(1){
    思索;
    P( fork[i] );
    P( fork[ (i+1) mod 5 ] );
    食事;
    V( fork[ (i+1) mod 5 ] );
    V( fork[i] );
  }
}
```



デッドロック
（deadlock）

どのプロセスも資源獲得の処理が進まず資源が確保できない状態

解法その1

- フォーク1本1本ではなく，
  「フォーク全体を使う権利」をセマフォで管理

```
Semaphore forks = 1;

philosopher( i ){
   while(1){
      思索;
      P( forks );
      食事;
      V( forks );
   }
}
```

うまくいくが，
同時に1人しか食事できない

実際は2人同時に
食事可能な場合があるはず

リソースが有効利用できていない

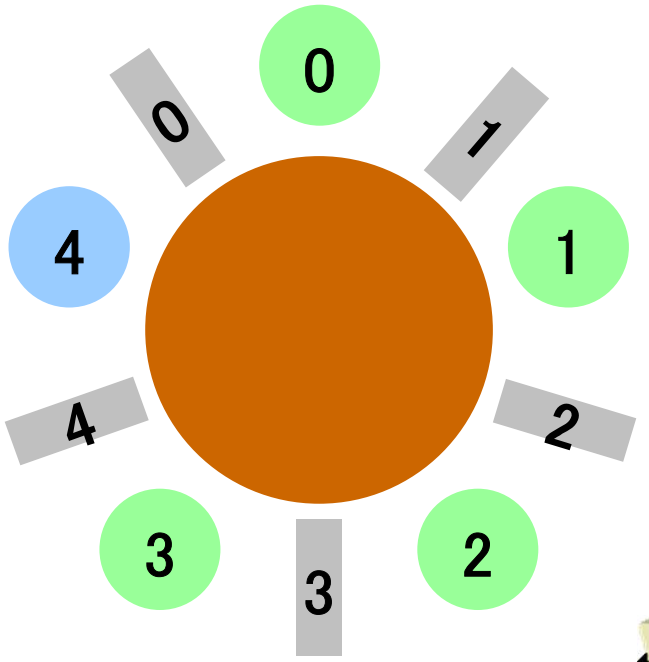
解法その2

- 1つのプロセスだけが逆順でフォークを要求

```
Semaphore fork[5] = {1,1,1,1,1};
philosopher( i ){
  while(1){
    思索;
    if( i != 4 ){
      P( fork[i] );                    //右
      P( fork[ (i+1) mod 5 ] );        //左
    }else{
      P( fork[ (i+1) mod 5 ] );        //左
      P( fork[i] );                    //右
    }
    食事;
    if( i != 4 ){
      V( fork[ (i+1) mod 5 ] );        //左
      V( fork[i] );                    //右
    }else{
      V( fork[i] );                    //右
      V( fork[ (i+1) mod 5 ] );        //左
    }
  }
}
```

4: 左，右

他: 右，左



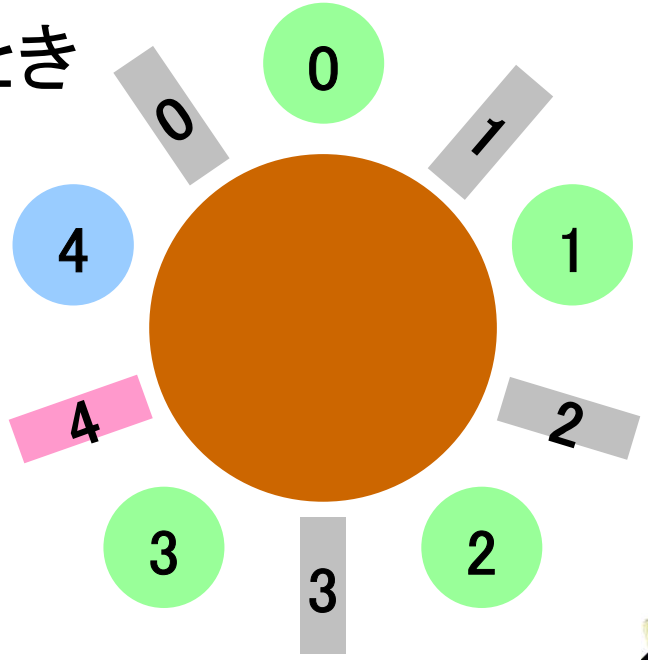
解法その2

- 1つのプロセスだけが逆順でフォークを要求
  - philosopher(3)が左を確保できないとき
    - → philosopher(4)は右を確保ずみ
    - → philos[illegible]食事中

4: 左，右

他: 右，左

うまくいくが，
哲学者4が特殊であるため，
公平性を欠いている可能性がある



解法その3

- 少し我慢をする哲学者
  - 右のフォークを確保後，
    左のフォークが確保できなければ，
    いったん右のフォークを解放して少し待つ
  - これによって，
    「全員右フォークを確保した状態」から抜け出せる

- 問題点
  - 全員が同時に「右フォーク確保，右フォーク解放」を繰り返すと，やはりデッドロック

解法その4

- 不定時間だけ我慢をする哲学者
  - 右のフォークを確保後，
    左のフォークが確保できなければ，
    いったん右のフォークを解放して少し待つ
  - 待つ時間はランダムに決定する
  - これによって，全員が同時に「右フォーク確保，右フォーク解放」を繰り返すことがなくなる
- 問題点
  - デッドロックが発生しないことを証明できない
  - フォークを解放して「ゆずった」哲学者は，
    次に優先されるしくみがないと公平性に欠ける

理想的な解法

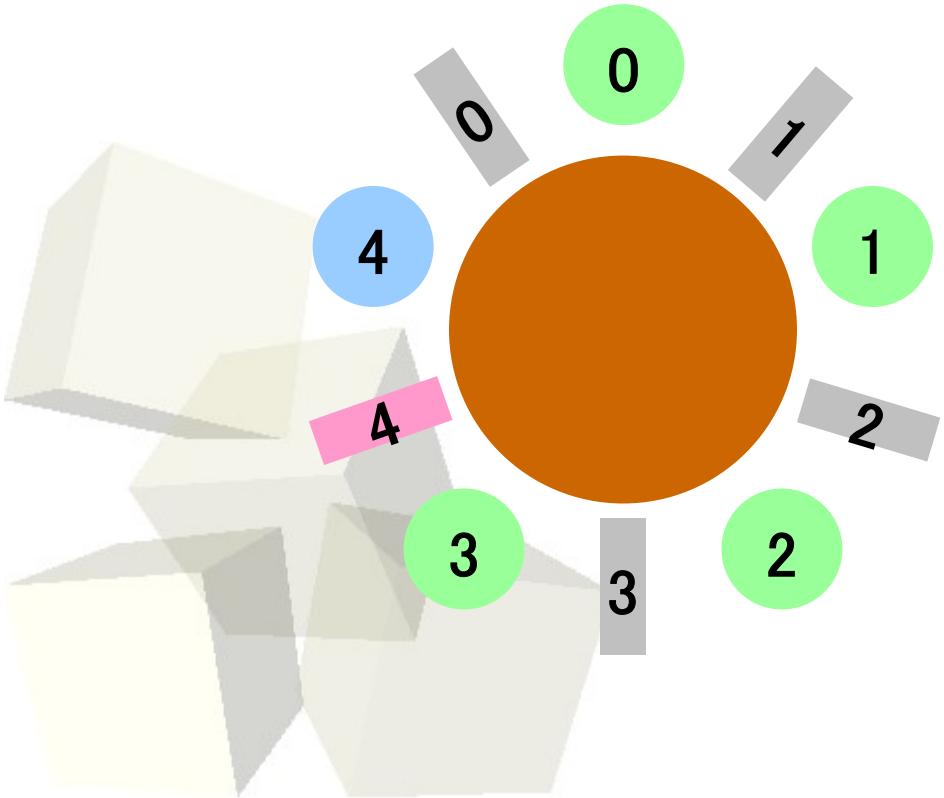
求められる条件

- リソース確保に失敗した場合，
  当該リソースを確保するための
  待ち行列に並ぶことができること

- 全てのプロセスがリソースを
  平等に確保できることを保証すること

Chandy/Misraの解法

- 1984年に提案
- 任意人のエージェント(P1,,, Pn)(哲学者)が任意個のリソース(R1,,,Rm)(フォーク)を獲得する状況に対して適用可能な方式



哲学者＝＞エージェント

フォーク＝＞リソース
実際はフォークを管理する変数で管理する
THINKING(初期値):未使用
HUNGRY　　:利用許可待ち
EATING　　:利用中

利用する変数

- 配列state[ ]:哲学者の3状態(THINKING,HUNGRY,EATING)を示す。両隣の哲学者がEATING状態でないときに、哲学者はEATING状態に遷移可能である。
- セマフォア配列 s[ ](初期値0):フォークの取得時の同期用。フォークを獲得できないときのwait処理に用いる。
- セマフォア変数　mutex: state[ ]操作への排他制御に用いる。

Candy/Misra解法

```
#Define N  5
#Define THINKING 0
#Define HUNGRY 1
#Define EATING 2
int     state[N];
Semaphore mutex = 1;
Semaphore s[N]={0};

void philosopher( int i ){
  while(1){
    思索;
    take_forks(i);
    食事;
    put_forks(i);
  }
}
void take_forks(int i){
    P(mutex);
    state[i]=HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}
```

```
void put_forks(int i) {
    P(mutex);
    state[i]=THINKING;
    test((i-1) mod N);
    test((i+1) mod N);
    V(mutex);
}
void test(int i){
    if(state[i] == HUNGRY &&
    state[(i-1) mod N] !=EATING &&
    state[(i+1) mod N] !=EATING) {
       state[i] =EATING;
       V(s[i]);
    }
}
```

Chandy/Misraの解法（1）

- take_forks(i)
  - Step1-1: 自身の状態state[i]をHUNGRYに設定する。
  - Step1-2: 両側の哲学者の状態がEATINGでない場合は、自身の状態をEATINGにすると同時に、V(s[i])を実行しEATING状態になったことを示す。
  - Step1-3: P(s[i])を実行し、もし前ステップでEATING状態にならなかった場合はwait状態に移行する(P(s[i]), i=0)。
- put_forks(i)
  - Step2-1: 自身の状態state[i]をTHINKINGに設定する
  - Step2-2: 両側の哲学者(A,B)の状態がHUNGRYであり、さらにそれぞれA,Bの両端の哲学者の状態がEATINGでない場合、哲学者A,Bの状態をEATINGにすると同時にV(s[i])を実行する。

Chandy/Misraの解法(2)

- ５人の哲学者全員が同時にHUNGRY状態
  - 哲学者0がEATING状態に移行(state[0]=EATING)した場合は、P(s[i])を通過し食事をすることができる。
- もし哲学者0がstate[i]=EATINGを実行した後、中断(プリエンプション)が起こった場合
  - 次に実行するプロセスが、哲学者1もしくは4の場合:当該プロセスはStep1-3のP(s[1])もしくはP(s[4])を実行することによりwait状態に移行する。
  - 次に実行するプロセスが、哲学者2もしくは3の場合：Step1-2の実行によりEATING状態に遷移し、食事をすることができる。

Candy/Misra解法

```
#Define N  5
#Define THINKING 0
#Define HUNGRY 1
#Define EATING 2
int     state[N];
Semaphore mutex = 1;
Semaphore s[N]={0};

void philosopher( int i ){
  while(1){
    思索;
    take_forks(i);
    食事;
    put_forks(i);
  }
}
void take_forks(int i){
    P(mutex);
    state[i]=HUNGRY;
    test(i);
    V(mutex);
    P(s[i]);
}
```

```
void put_forks(int i) {
    P(mutex);
    state[i]=THINKING;
    test((i-1) mod N);
    test((i+1) mod N);
    V(mutex);
}
void test(int i){
    if(state[i] == HUNGRY &&
    state[(i-1) mod N] !=EATING &&
    state[(i+1) mod N] !=EATING) {
       state[i] =EATING;
       V(s[i]);
    }
}
```

まとめ

- セマフォ
  - 排他制御の枠組み
  - P命令
    - → リソース獲得要求，失敗時には待ち状態に移行
  - V命令
    - → リソース解放，待ちプロセスを実行可能状態に
  - リソースを獲得できなかったプロセスは待ち状態に
    - → ビジーウェイトが発生しない

まとめ

プロセス協調問題

- Producer–Consumer
  - → プロセス間通信，計算機間通信
- Reader–Writer
  - → データベースアクセス制御
- Dining Philosophers
  - → 複数リソースを要求する場合
  - → デッドロックの考慮が重要