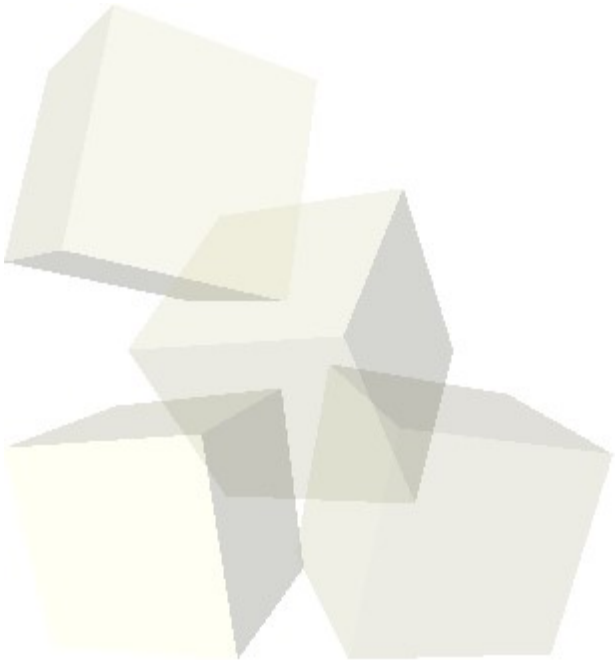
オペレーティングシステム

#4 並行プロセス:排他制御基礎



4.1
プロセスの
競合，協調，干渉

プロセスの同時実行

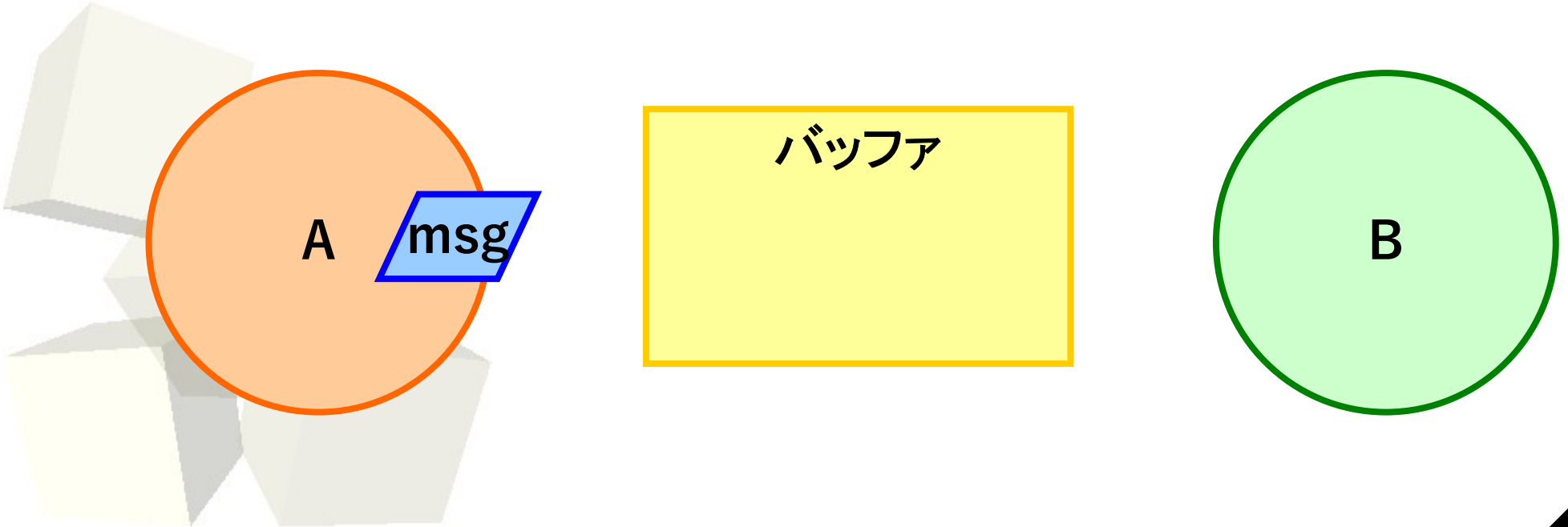
- 複数プロセスが同時に動作する際に...

- プロセス協調
  - 仕事の分担や通信など，複数プロセスが助け合う
- プロセス競合
  - 複数プロセスで有限リソースを取り合う
  - 調停し，各プロセスに適切にリソース割当
- プロセス干渉
  - 他プロセスの影響で異常が発生すること
  - 原因はプログラムのバグ

プロセス協調

例）プロセス間通信

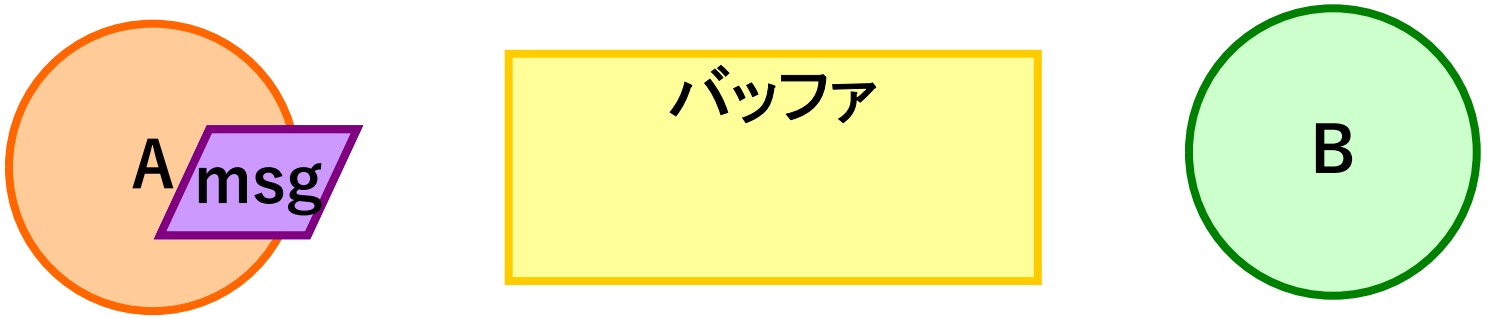
- 何も通信のための仕組みがない場合…
  - → 送信側と受信側でタイミングを合わせる必要
  - → 受信側は，常にメッセージが来ないかを
    チェックしていなければならない
- 通信バッファ



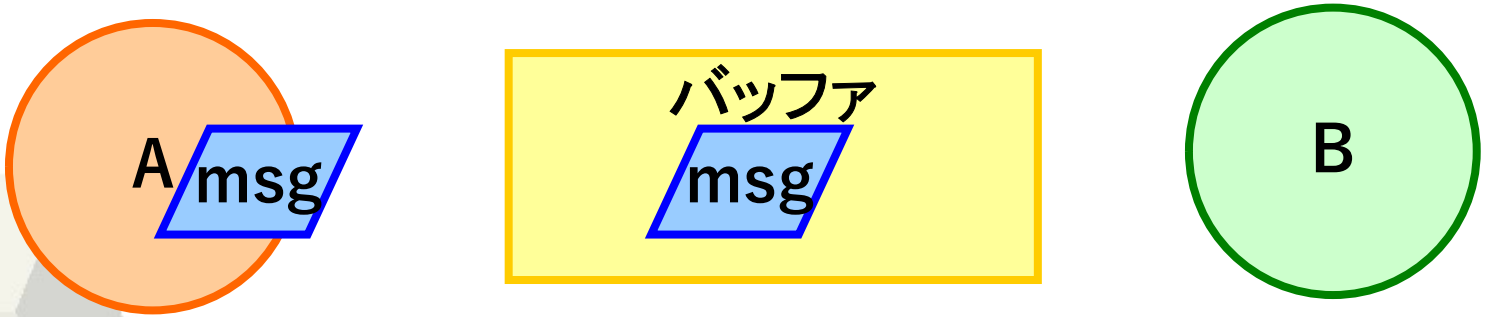
プロセス協調

問題

- 読み出す前に上書き…とりこぼし

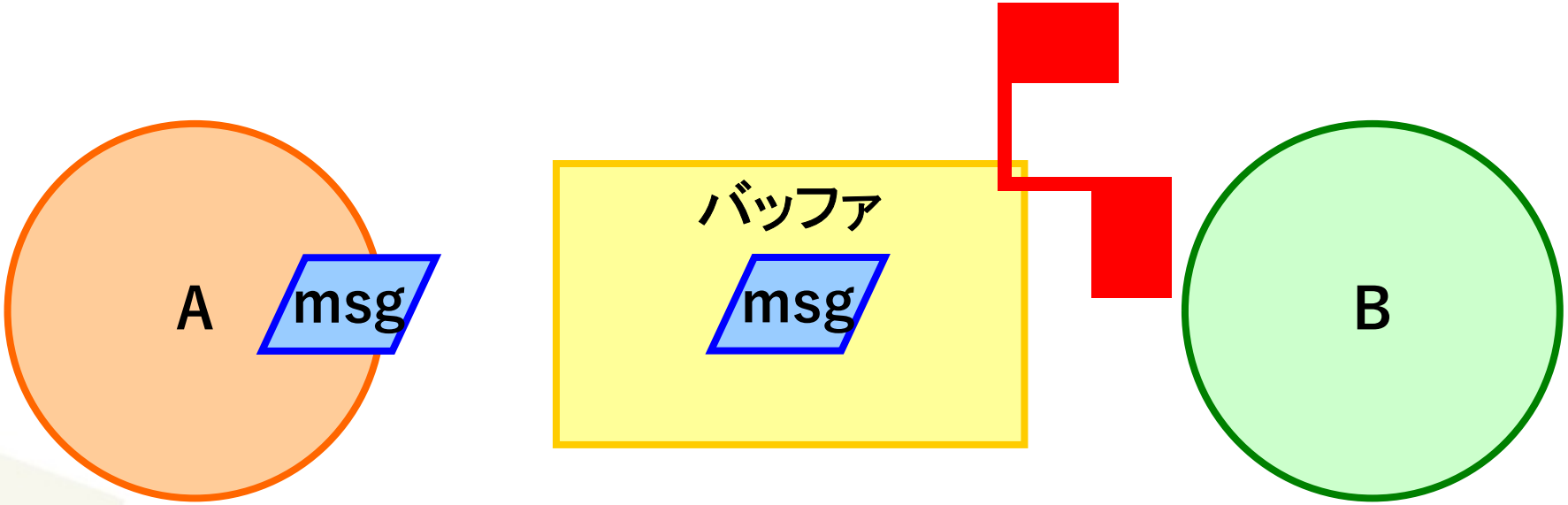


- 格納する前に再読込…だぶって受け取り



フラグによる管理

- 受信すべきメッセージが
  バッファ内に存在するか否かをフラグで判断



- フラグが立っている間,
  送信側は新たに送信を行わない → 上書き回避
- フラグが降りている間,
  受信側は新たに順を行わない → 再受信回避

プロセス競合

- 例）磁気テープの利用

NUM: 空きテープ数

プログラム例

```
LOAD        NUM,   R
DEC         R, 1
STORE       NUM, R
        :
        :テープ使用
        :
LOAD        NUM, R
INC         R, 1
STORE       NUM, R
```

テープを確保

テープを解放

プログラムの意味

- テープの確保
  - → 変数NUMの値をレジスタにロード
  - → レジスタから1減算
  - → レジスタの値を変数NUMにストア（格納）
- テープの解放
  - → 変数NUMの値をレジスタにロード
  - → レジスタから1加算
  - → レジスタの値を変数NUMにストア

```
LOAD        NUM, R
DEC         R, 1
STORE       NUM, R
      :
      :テープ使用
      :
LOAD        NUM, R
INC         R, 1
STORE       NUM, R
```

- プロセスAとプロセスBがテープにアクセス
  - プロセスBが確保中
  - プロセスAによる確保とプロセスBによる解放が発生
  - 結果，テープの空き数は変化しないはず

ケース1

- LOAD NUM, R R=1
- DEC R, 1 R=0
- STORE NUM, R R=0

- LOAD NUM, R R=0
- INC R, 1 R=1
- STORE NUM, R R=1

NUM

0

テープの空きは
1でOK

ケース2

- LOAD NUM, R R=1
- DEC R, 1 R=0
- STORE NUM, R R=0

- LOAD NUM, R R=1
- INC R, 1 R=2
- STORE NUM, R R=2

NUM

0

テープの空きは
1しかないはず
(実際より多いと勘違い)

ケース3

- LOAD NUM, R R=1
- DEC R, 1 R=0
- STORE NUM, R R=0

- LOAD NUM, R R=1
- INC R, 1 R=2
- STORE NUM, R R=2

NUM

1

テープの空きは
1あるはず
（実際より少ないと勘違い）

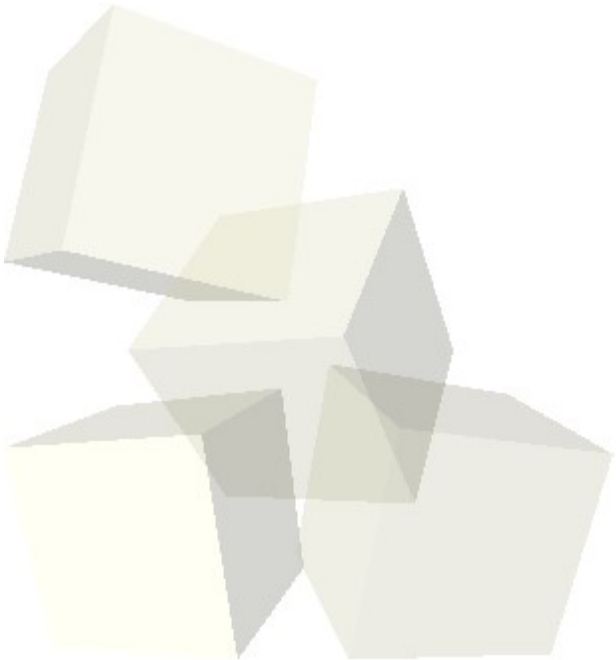
原因

- 変数NUMから値を読んで，変数NUMに値を書くまでの間に，他のプロセスが変数NUMを読んでしまう
  - → 他のプロセスからみて，変数NUMは変化していない
  - → 実際は変化させるための手続きが始まっている

対処法

- LOADからSTOREまでの一連の処理を不可分に
- クリティカルセクション：
  このような分割してはいけない一連の処理
- 排他制御（mutual exclusion）：
  クリティカルセクションなどを他のプロセスと
  排他的に実行するための制御

4.2
排他制御



排他制御

- プロセスがクリティカルセクションを
  他のプロセスと排他的に実行できるように

- 重要な性質
  - 即時性
  - デッドロック防止
  - 公平性

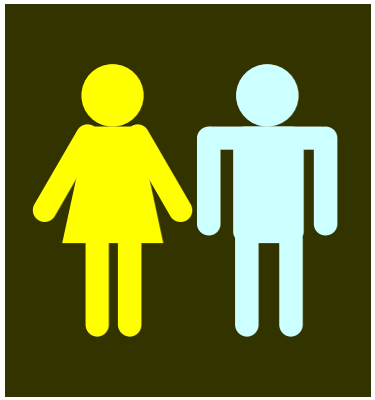
排他制御の性質

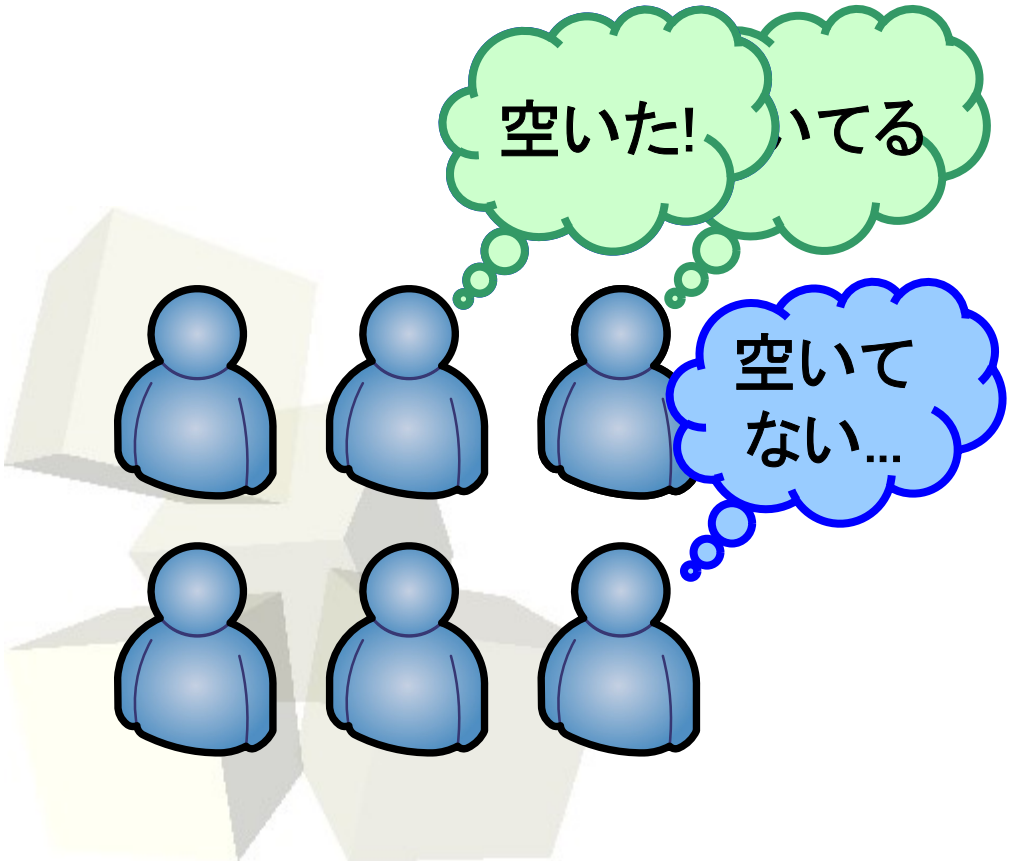
- 即時性
  - クリティカルセクションの実行に競合するプロセスがほかにない場合，プロセスはクリティカルセクションの実行を即座に許可される。
- デッドロック防止
  - 競合するプロセスがある場合でも，許可されるまで永久に待たされてはいけない。
- 公平性
  - どのプロセスも，他のプロセスがクリティカルセクションを実行することを妨げられない。

クリティカルセクション

- エントリーシーケンス
  - クリティカルセクションに入る権利を獲得する処理
- イグジットシーケンス
  - クリティカルセクションから出るための処理
- 例）フラグによる制御

例）フラグによる制御

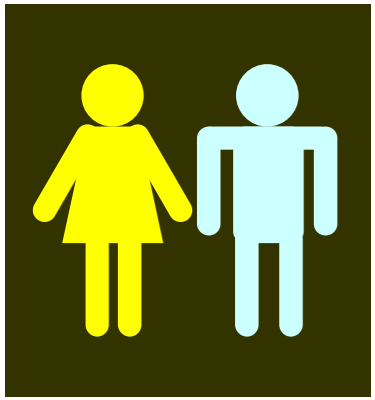




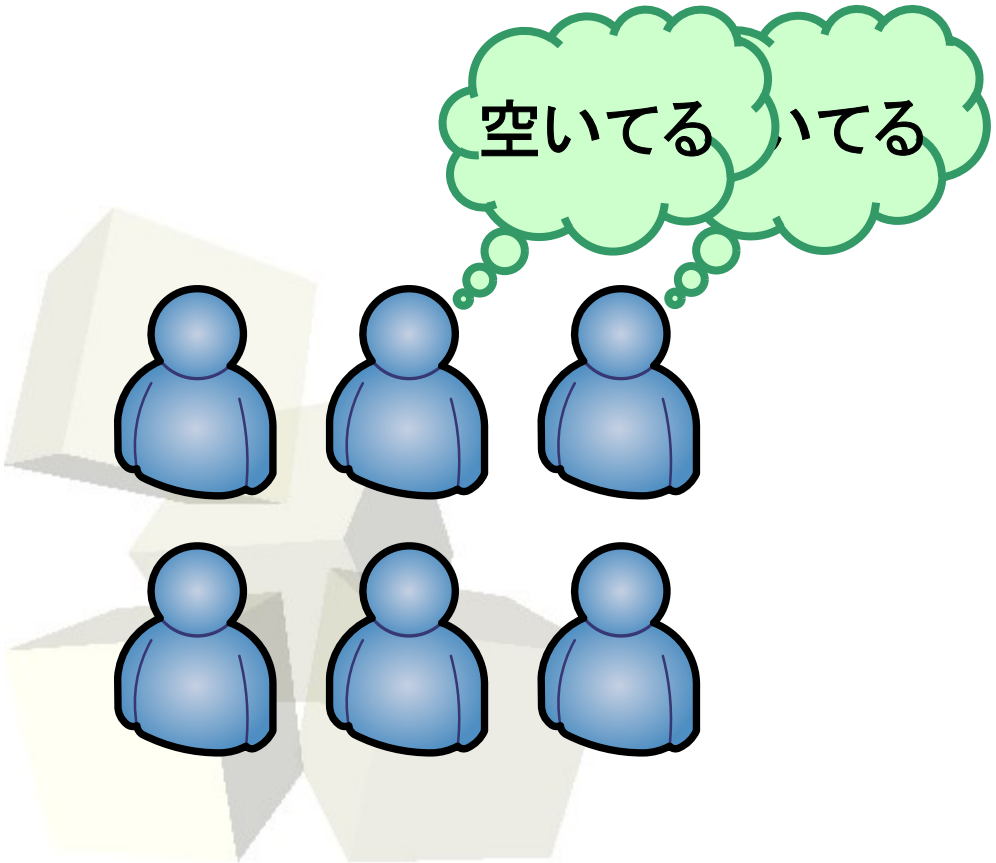
- 新幹線のトイレと同じ
  - クリティカルセクション（トイレ）に入ろうとするプロセス（乗客）は，フラグ（インジケータ）を確認し，入れるかどうかを決定
  - 入ると同時にフラグを下げる（インジケータが点く）

一見うまくいく

例）フラグによる制御



- うまくいかない場合
  - フラグを見て確認
  - 入る



- という2つの処理は不可分

この方式では
排他制御を実現できない

4.3
Dekkerのアルゴリズム

Dekkerのアルゴリズム

- 2プロセスの排他制御を行うことを可能とする

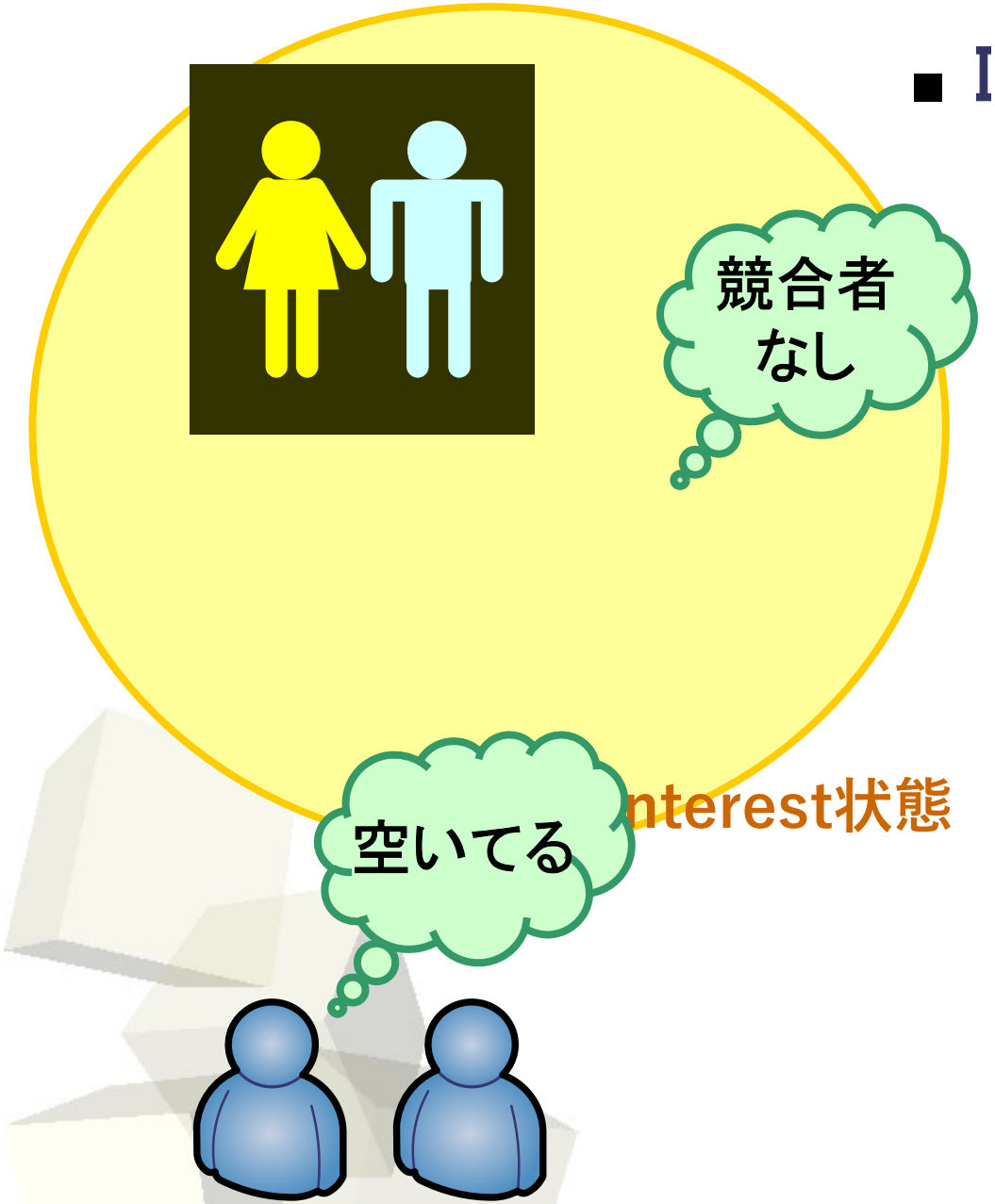
- Interest
  - プロセスA,Bがクリティカルセクションに興味があるか否かを示す
- Priority
  - プロセスA,Bがクリティカルセクションに同時に興味を持った場合，どちらを優先するかを決定する

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;
```

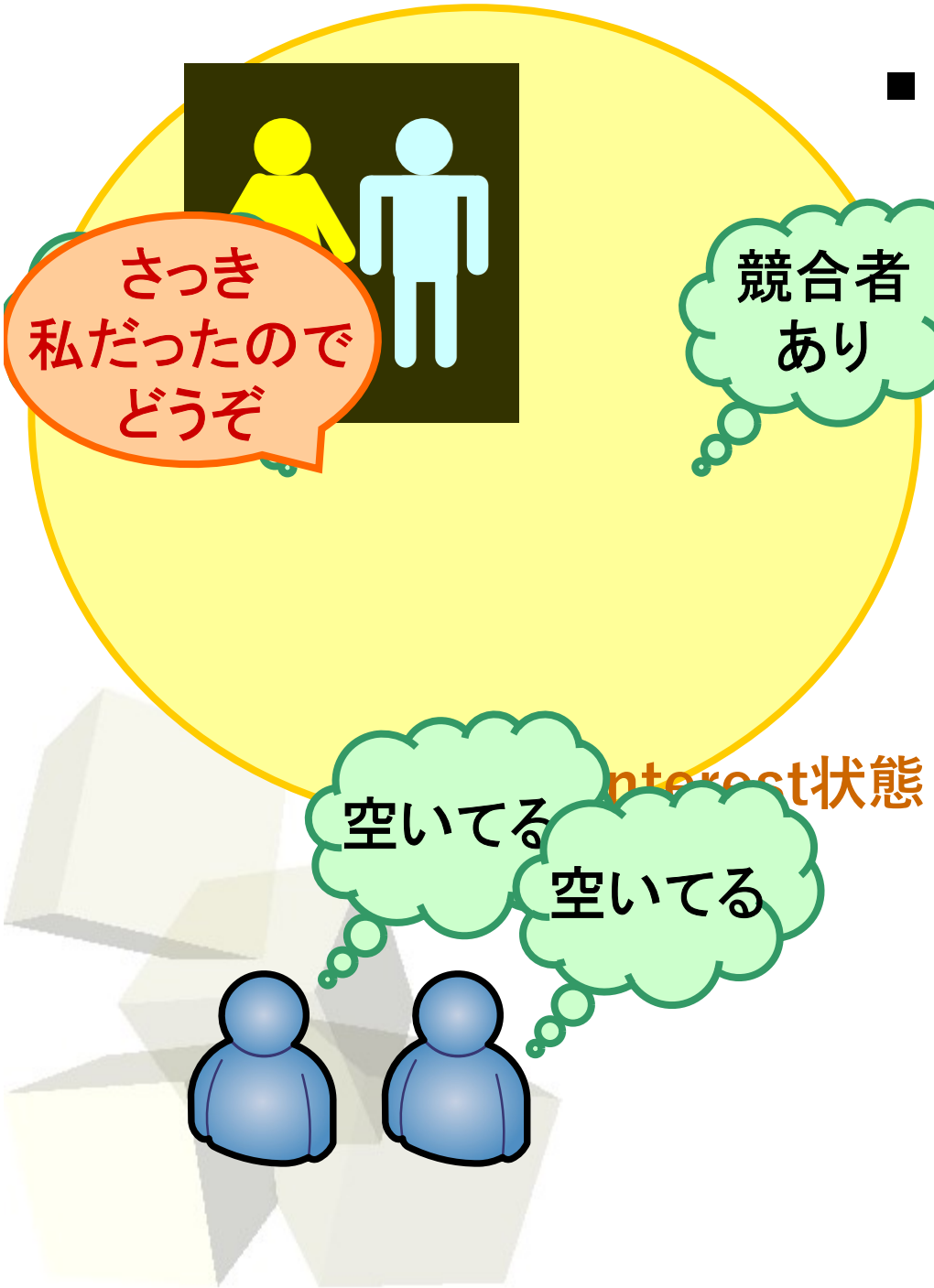
Dekkerのアルゴリズム



Interest

- まずクリティカルセクションに入る前に,
- クリティカルセクションに入りたい旨を宣言
- 競合者がいなければ
  入れる

Dekkerのアルゴリズム



- Priority
  - 競合者がいた場合
  - Priorityが示す優先度で
    どちらが入るか決定
  - 自分に優先度が回ってくるまで
    Interest状態を解除

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;
```

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;
```

BがInterestの間

Bに優先権がある間

優先権を得たら
再びInterest状態オン

終わったら優先権を

Interest状態オフ

Dekkerのアルゴリズム

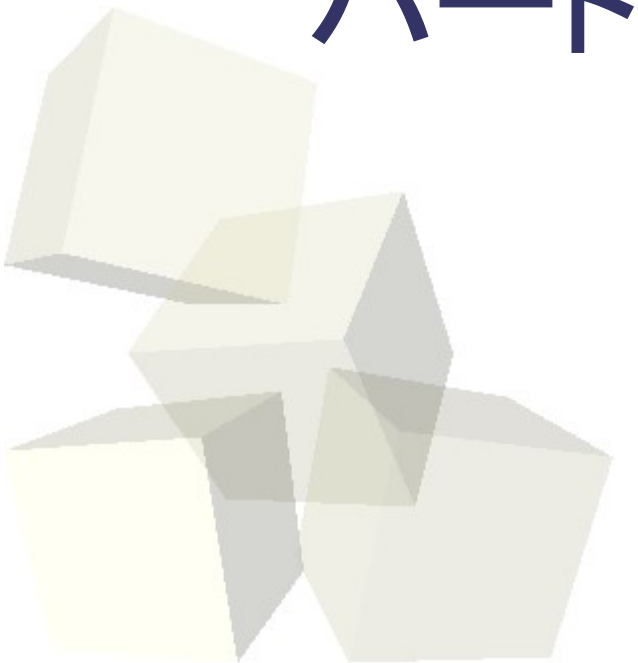
- ポイント
  - 入る前に手を挙げる
  - 優先権により競合を解決
- 問題点
  - ユーザプログラムに依存
    - → ちゃんとプロセスが約束を守ってくれないと破綻
  - ビジーウェイト（busy wait）
    - → 一方がクリティカルセクションを実行中，
    - → 待っている方は優先権をひたすらチェックし続ける
    - → CPUリソースの無駄
- 最近はメニーコアが主流。CPUリソースが余っているためビジーウエイトが必ずしも悪ではない状況が発生
  - → ビジーウエイトの利点
    - リソースが空いたときの、反応が早い
      - （名前が変わって）スピンロック

4.4
割り込み制御による排他制御

ユニプロセッサの場合

- 割り込みのみがプロセス中断を発生させる
  - エントリーシーケンスで
    割り込み禁止命令を実行しておけばよい
  - 同様にイグジットシーケンスで
    割り込み禁止を解除
- 利点
  - 簡単
  - 軽い（OSの負荷が少ない）
- ただし…
  - 割り込み禁止時間の増加はシステムの性能に影響

4.5
ハードウェアによる排他制御



ハードウェアによる排他制御

- 対話処理の重要性から排他制御の必要性が認識

- テストアンドセット命令
  - ハードウェア自体に，排他制御のための仕組みを
  - v = test_and_set(x)
    - → v = x と x = 0 を同時に実行する命令
  - 競合者フラグのチェックとセットを同時に行える

TEST AND SETによる排他制御

```
Flag X = 1;
```

```
Int v;
Repeat
  v = test_and_set(X);
Until v == 1;

       :
クリティカルセクション
       :

X = 1;
```

```
Int u;
Repeat
  u = test_and_set(X);
Until u == 1;

       :
クリティカルセクション
       :

X = 1;
```

今日のまとめ

- プロセス競合
  - クリティカルセクション:
    - → リソース競合が発生する可能性のある部分
  - 排他制御(MUTEX):
    - → 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
  - 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
    - → ビジーウェイトという問題点
- プロセス協調