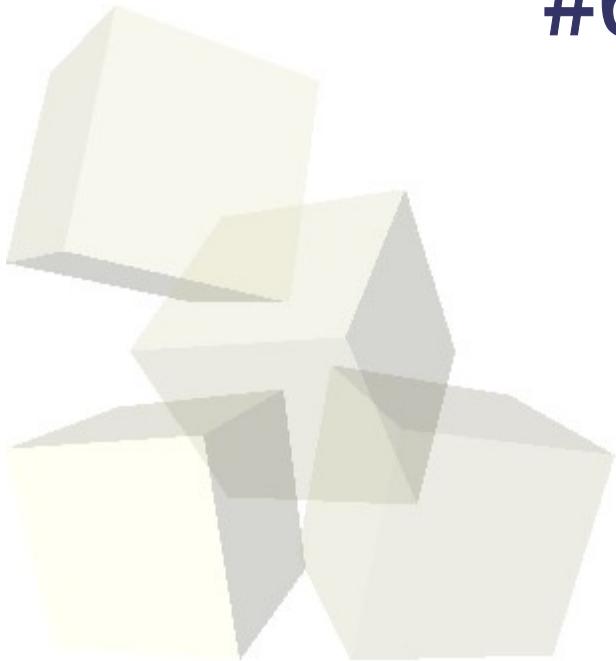




オペレーティングシステム

#6 並行プロセス: モニタ



■ 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

■ 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

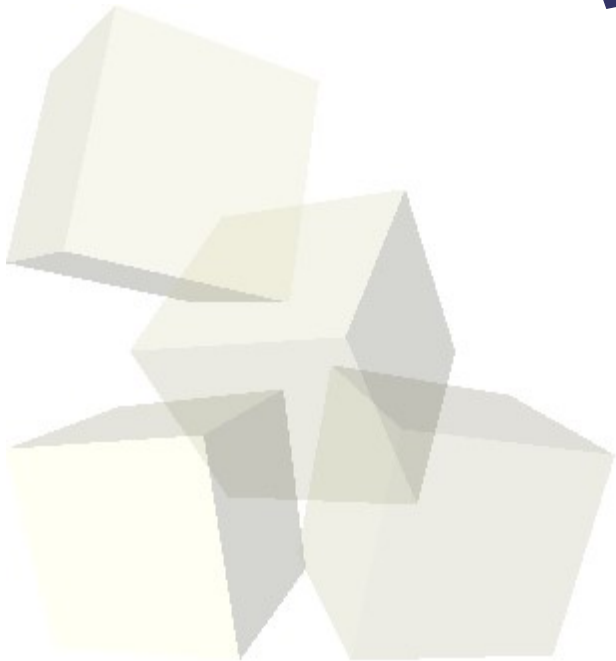
■ Dining Philosophers

- 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



6.1

セマフォの問題点



■ ユーザ依存

- P命令

- 発行せずにクリティカルセクションにアクセス可能
- 他のプロセスからリソースの状況が判断不可能に

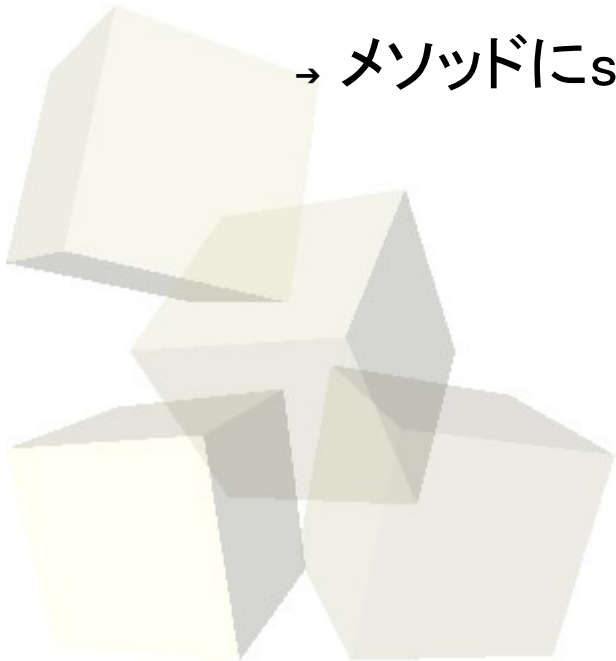
- V命令

- P命令でリソースへのアクセス権を獲得し、
アクセス終了後にV命令を実行しないことも可能
- 他のプロセスは永遠にリソースへのアクセス不可能

- 全てユーザ(プログラマ)の良識・責任に
任せられている

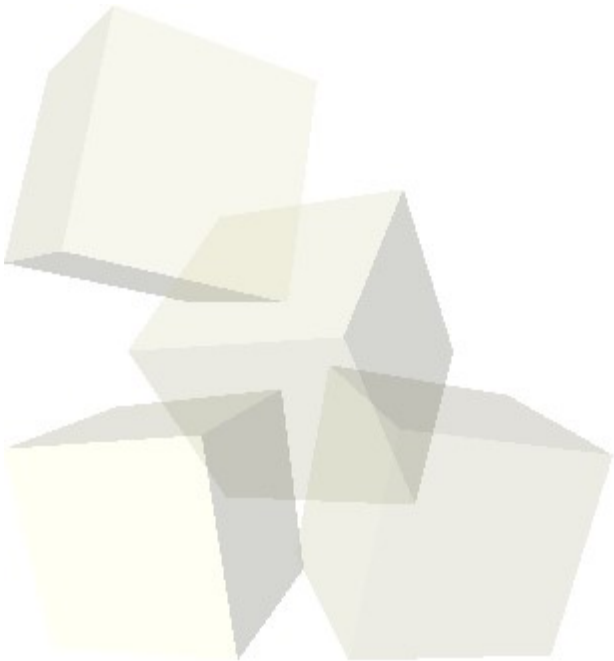
■ モニタ

- セマフォより洗練された排他制御の仕組み
- オブジェクト指向の枠組み
- Java の初期から、同期機構として採用されている
 - 共有データに `private` 修飾子
 - メソッドに `synchronized` 修飾子



■ モニタの詳細

■ ...の前に





6.2

オブジェクト指向プログラミングとは？



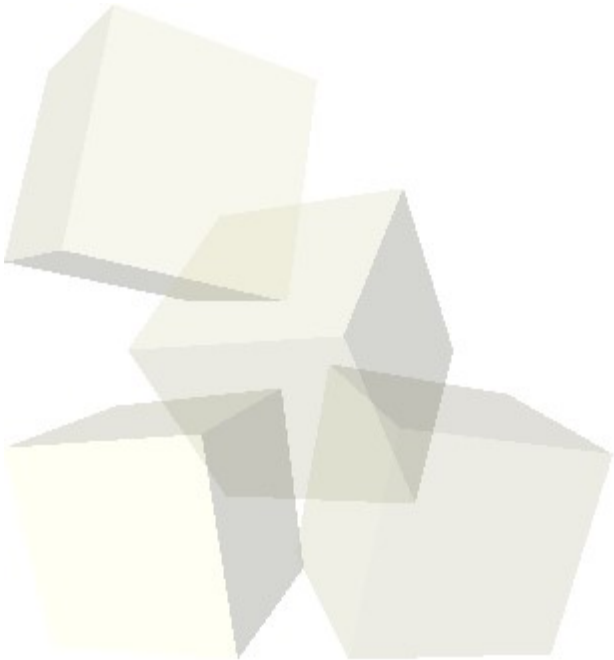
■ オブジェクト指向 (プログラミング)

- **オブジェクト** (物体) 同士の相互作用としてシステムの振る舞いをとらえる考え方
- プログラムの構造を, オブジェクト群の相互作用およびその雛形である**クラス**の関係として捉える

■ オブジェクト指向言語

- C++
- Java
- Smalltalk
- Objective-C
- Python(?)
- Ruby(?)

- クラス (class)
- メソッド (method)
- インスタンス (instance)



■ クラス

- あるオブジェクトの
特徴や機能を定義する

■ たとえば自動車

- さまざまな内部状態の
集まりで表現できる



```
class 自動車{  
  
    投入燃料量;  
  
    制動力;  
  
    タイヤ角度;  
  
    ブレーキランプ;  
  
    :  
    :  
}
```



■ 減速させたいとき

- 投入燃料量 = 0;
- 制動力 += n;
- ブレーキランプ = on;
- if (スリップ){ ... };
- 物体に対する深い知識が必要
- 手順が面倒
- システムとして異常な動作も許してしまう
 - 例) 減速中でないのに
ブレーキランプ = on;

```
class 自動車{  
  
    投入燃料量;  
  
    制動力;  
  
    タイヤ角度;  
  
    ブレーキランプ;  
  
    :  
    :  
}
```



■ メソッド

- オブジェクトに対する操作をメソッドとして外に提供
- 操作を系統的に

■ カプセル化

- 外からタッチ可能な範囲を限定
- メソッドを通じてのみオブジェクトの内部状態を変更可能に
- 誤操作が減る

```
class 自動車{  
public:  
    減速(n){  
        投入燃料量 = 0;  
        制動力 += n;  
        ブレーキランプ = on;  
        if( スリップ ){ ... }  
    }  
private:  
    投入燃料量;  
    制動力;  
    タイヤ角度;  
    ブレーキランプ;  
    :  
    :  
}
```

■ クラス

- あくまで
オブジェクトの定義

■ インスタンス

- あるクラスの実例である
オブジェクト
- メソッド呼びは、イン
スタンス(実体)に
対して行う

```
class 自動車{
```

```
public:
```

```
    減速( n ){
```

```
        :
```

```
    }
```

```
    :
```

```
private:
```

```
    :
```

```
}
```

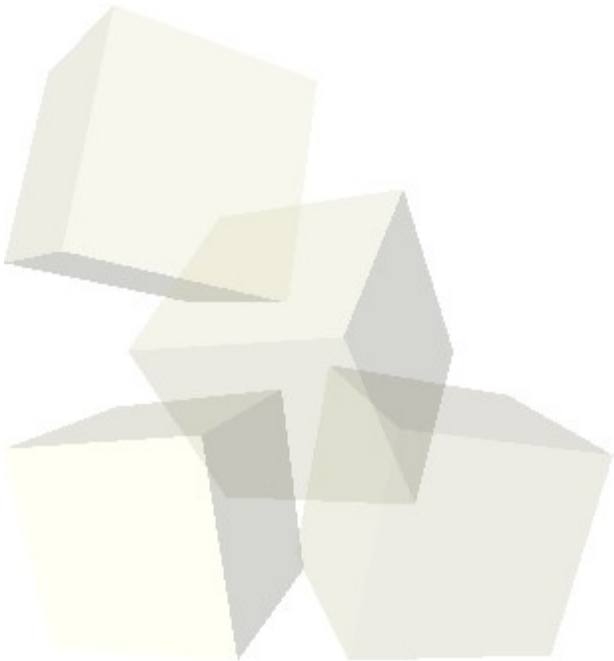
```
自動車 A氏の自動車;
```

```
main(){
```

```
    A氏の自動車.減速(10);
```

```
}
```

6.3 モニタ



	セマフォ	モニタ
提唱者	Dijkstra (1965)	Hoare (1978)
形態	手続き (サブルーチン)	構造化 (オブジェクト指向)
可読性・保守性	低	高
提供形態	ライブラリ等	言語仕様の一部

■ カプセル化

■ wait / signal / queue

■ 以下, 実例を通じて...

■ 共有リソース

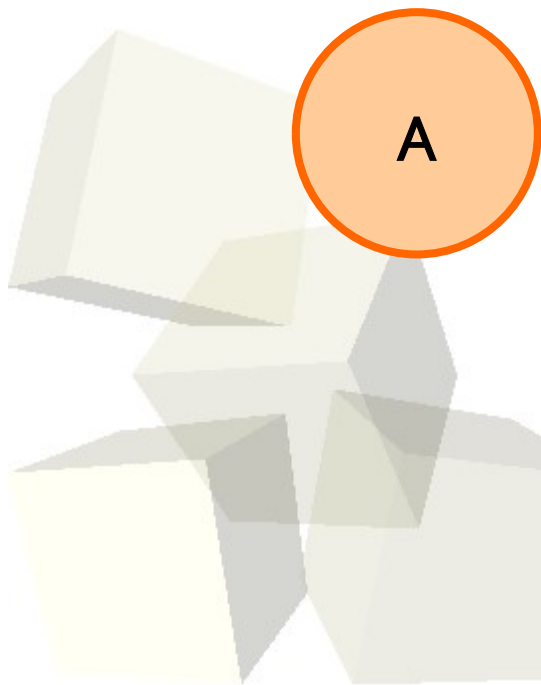
- 直接アクセス禁止
- メソッドを通じてのみアクセス可能
- 不正な直接アクセスは
コンパイル時に検出

■ メソッド

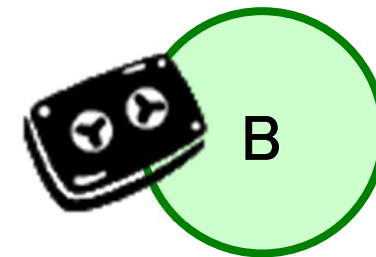
- 排他的
- 他のプロセスがメソッド呼び出し中は、待ち状態へ

```
monitor{  
public:  
    use_resource1(n){  
        リソース1 += n;  
    }  
    use_resource2(){  
        :  
    }  
    :  
private:  
    リソース1;  
    リソース2;  
    :  
    :  
}
```

- 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
 - ・ テープの空き数: 共有変数



空きテープ





共有変数へのアクセス

```
monitor Shared_I {  
  private:  
    int I;  
  public:  
    increment( amount ) {  
      I += amount;  
    }  
    decrement( amount ) {  
      I -= amount;  
    }  
}
```

```
Shared_I num;
```

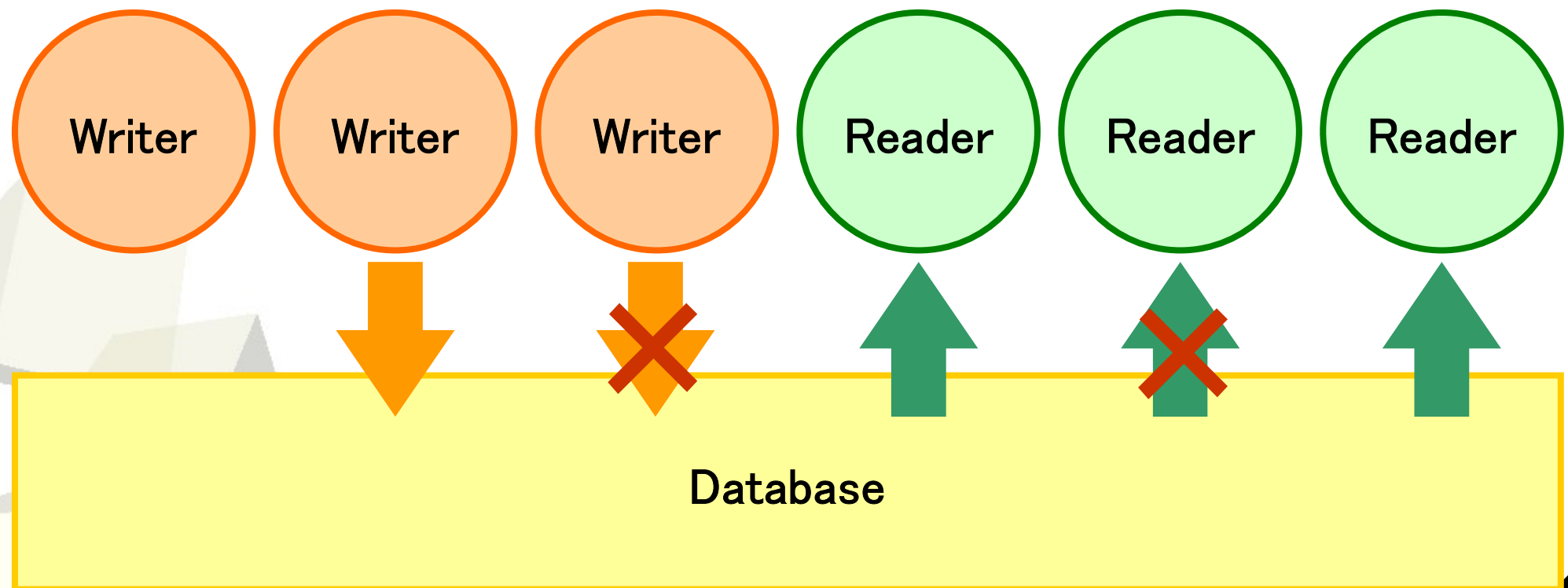
```
num.decrement(1);  
テープ利用;  
num.increment(1);
```

```
num.decrement(1);  
テープ利用;  
num.increment(1);
```

共有変数へのアクセスは
メソッドを通じてのみ

プログラマは排他制御のための
記述の必要なし

- 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





リーダライタ問題(その1)

```
monitor Reader_Writer_1{
private:
    int Readers = 0;
    int Writers = 0;
    boolean busy = FALSE;
public:
    start_read(){
        while(Writers!=0){};
        Readers += 1;
    }
    finish_read(){
        Readers -= 1;
    }
    start_write(){
        Writers += 1;
        while( busy ||
            Readers!=0 ){};
        busy = TRUE;
    }
    finish_write(){
        Writers -= 1;
        busy = FALSE;
    }
}
```

```
Reader_Writer_1 lock;
```

```
lock.start_write();
DBに書く;
lock.finish_write();
```

```
lock.start_read();
DBから読む;
lock.finish_read();
```

無限ループによる待機中,
他プロセスはモニタにアクセスできない

■ 条件変数

- ある条件が成立するまでプロセス・スレッドを待機させる仕組み

■ 条件変数操作のためのメソッド

- wait
 - 待ち状態に移行
- signal (javaでは notify(),notifyAll())
 - 待ち状態にあるプロセスのうち1つを実行可能に
- queue
 - 待ち状態のプロセスの有無を返す

■ wait

- ・ 待ち状態に移行

■ signal

- ・ 待ち状態にあるプロセスのうち1つを実行可能に

■ queue

- ・ 待ち状態のプロセスの有無を返す

```
class condition{  
  
private:  
    待ち行列;  
  
public:  
    wait() {  
        プロセスを待ち行列に追加;  
    }  
    signal() {  
        待ち行列からプロセスを  
        選択し, 実行可能に;  
    }  
    queue() {  
        if( 待ち行列の長さ > 0 )  
            return TRUE;  
        else  
            return FALSE;  
    }  
  
}
```



```
monitor Reader_Writer_2{

private:
    int Readers = 0;
    int Writers = 0;
    condition OK_read, OK_write;

public:
    start_read(){
        if( busy || OK_write.queue() )
            OK_read.wait();
        Readers += 1;
        while( OK_read.queue() )
            OK_read.signal();
    }

    finish_read(){
        Readers -= 1;
        if( Readers == 0 )
            OK_write.signal();
    }
}
```

```
start_write(){
    if( Readers != 0 || busy )
        OK_write.wait();
    busy = TRUE;
}

finish_write(){
    busy = FALSE;
    if( OK_read.queue() )
        OK_read.signal();
    else
        OK_write.signal();
}
}
```

```
class condition{
public:
    wait(){ ... }
    signal(){ ... }
    queue(){ ... }
}
```

■ セマフォ

■ P命令

- 共有リソースの取得
トライ
- 失敗時, 待ち行列へ

■ V命令

- 共有リソース返却
- 待ちプロセスを1つ
実行可能状態へ

■ モニタ

■ wait

- 待ち行列へ

■ signal

- 待ちプロセスを1つ
実行可能状態へ

■ queue

- 待ちプロセスの有無

■ セマフォ

- P命令を実行しないとセマフォの状態が分からない
- リソースを取ろうとしない
と、空いてるかどうか不明
- 取れなかったら、いきなり
待ち行列に待たされる

■ モニタ

- メソッドにより、
共有リソースの状態を
排他的に調べられる
- リソースの空きの
確認とリソース待ちが
分離
- 条件変数への wait に
より、自由度の高い
「待ち」が可能

■ リソース確認と「待ち」の分離

- ・ リソースに空きがない場合, 「待ち」に入るかどうか自由に選べる

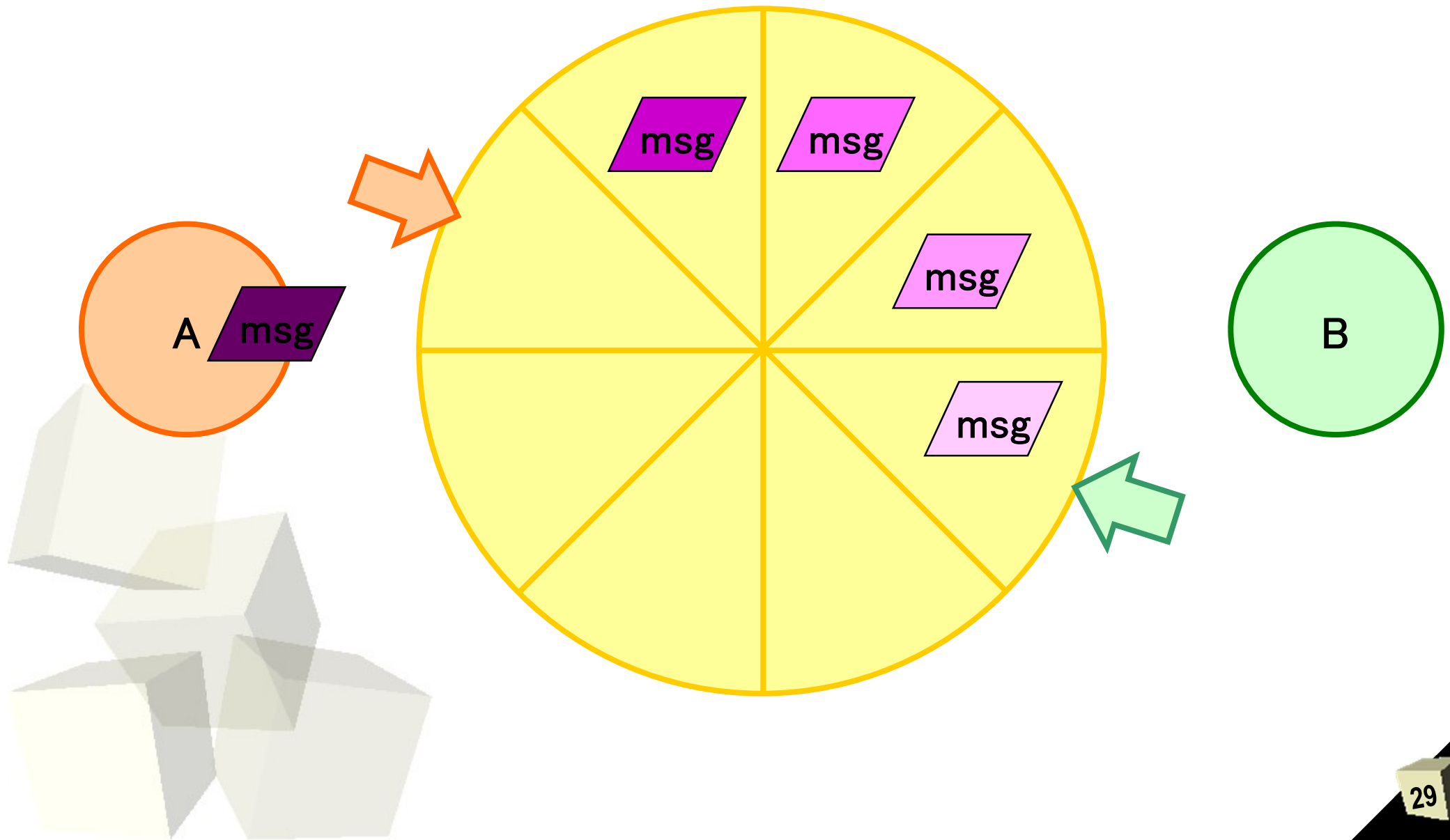
■ 排他制御すべきリソースの明示

- ・ モニタ内に記述されるため明示的
- ・ 他の一般的な変数と判別しやすい
- ・ 排他的メソッドを通じた処理により, 保護

プログラマに
安全で扱いやすい枠組みを提供



■ リングバッファ





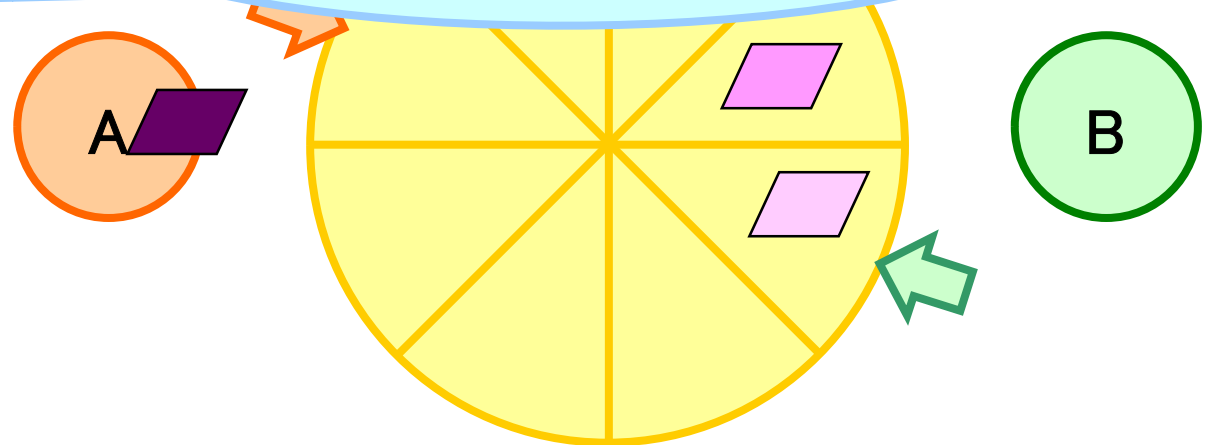
プロデューサコンシューマ

```
monitor Producer_Consumer{
private:
    Messages Buffer[N];
    int S, M;
    condition Full, Empty;
    int Count;
public:
    producer( word ){
        if( Count == N )
            Full.wait();
        Buffer[S] = word;
        S = (S + 1) mod N;
        Count += 1;
        Empty.signal();
    }
    consumer( word ){
        if( Count == 0 )
            Empty.wait();
        word = Buffer[M];
        M = (M + 1) mod N;
        Count = Count - 1;
        Full.signal();
    }
}
```

メッセージでいっぱい

空いたら書いて、
ポインタを進める

Emptyで読み出し待ち
しているかもしれないので





Dining Philosophers

```
enum{ EATING, HUNGRY, THINKING };
monitor Dining_Philosophers{
private:
    state[N];
    condition self[N];
    test( i ){
        if( (state[(i-1) mod N] != EATING)
            && (state[i] == HUNGRY)
            && (state[(i+1) mod N] != EATING) ){
            state[i] = EATING;
            self[i].signal();
        }
    }
public:
    up( i ){
        state[i] = HUNGRY;
        test( i );
        if( state[i] != EATING )
            self[i].wait();
    }
    down( i ){
        state[i] = THINKING;
        test( (i-1) mod N );
        test( (i+1) mod N );
    }
}
```

食べようとして、
成功すると
EATINGに

食事トライ
失敗すると
waitで待ち

食事終了
両隣の人がHUNGRY
ならEATINGにして
signalで起こす

■ 利点

- カプセル化による, 排他制御の簡単化
- 可読性・保守性の高さ
- 非公開リソースへのアクセス, `signal()/wait()`のセット使用など, コンパイル時にある程度の不具合検出が可能

■ 欠点

- コンパイル時チェックは完全ではない
 - デッドロックの可能性を完全に排除しているわけではない
- サポートしている言語が少ない
 - Javaによる採用で, 再び脚光?

- `public final void notify()`
 - ・ オブジェクトを待っている1つのスレッドを起こす
- `public final void notifyAll()`
 - ・ オブジェクトを待っているすべてのスレッドを起こす
- `public final void wait()`
`throws InterruptedException`
 - ・ 他のスレッドに起こされるのを待つ。モニタに関する Synchronization Lock はすべて解除
 - ・ 起こされたら、モニタを再獲得する必要あり



```
public class Counter {
    private int count;
    private int N;
    public Counter(int max){
        count = 0;
        N = max;
    }
    public int get(){return count; }
    synchronized void inc() throws InterruptedException{
        while(count==N) wait();
        count++;
        notifyAll();
    }
    synchronized void dec() throws InterruptedException{
        while(count==0)wait();
        count--;
        notifyAll();
    }
}

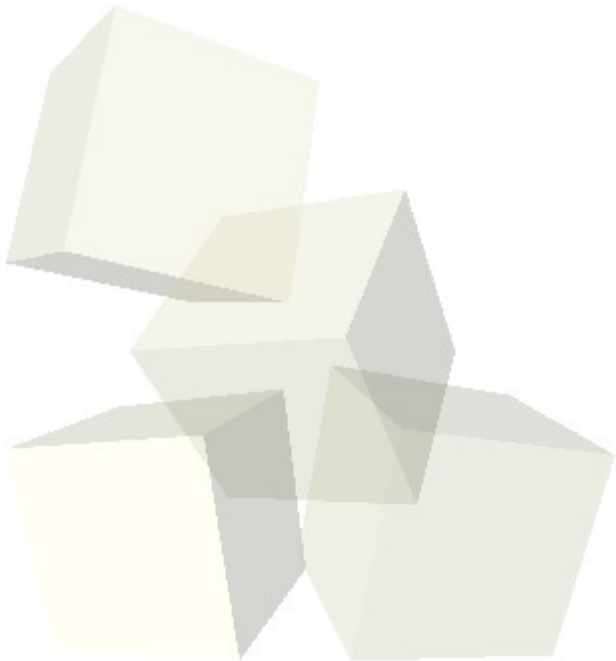
public class IncRunnable implements Runnable{
    private Counter counter;
    public IncRunnable(Counter c){counter = c;}

    @Override
    public void run() {
        while(true){
            try {
                counter.inc();
            } catch (InterruptedException e){}
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter(10);
        new Thread(new IncRunnable(counter)).start();
        new Thread(new DecRunnable(counter)).start();
        for(int i=0;i<10000;i++)
            System.out.println(counter.get());
    }
}
```

■ モニタ

- オブジェクト指向プログラミングを排他制御に適用
- リソース操作を, オブジェクトのメソッドとして提供
- メソッドは排他的にのみ実行可能であり, プログラマはリソース排他制御に注意を払う必要がない
- デッドロックの可能性を, コンパイル時にある程度排除可能



■ 並行プロセス

- 機器上では一般に複数プログラムが**並行動作**
- **プロセス**並列, さらに細かい**スレッド**並列など
- これに対し, システムのリソースはOSにより**仮想化**(無限)されているが**実際は有限**
- 有限リソースの使用権を, プロセス/スレッド間で**調停する必要アリ**
- つまりリソース使用が競合しうる場所で何らかの対処が必要

■ クリティカルセクション

- プログラム中で,
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、複数プロセスが
同時に入らないようにするための制御

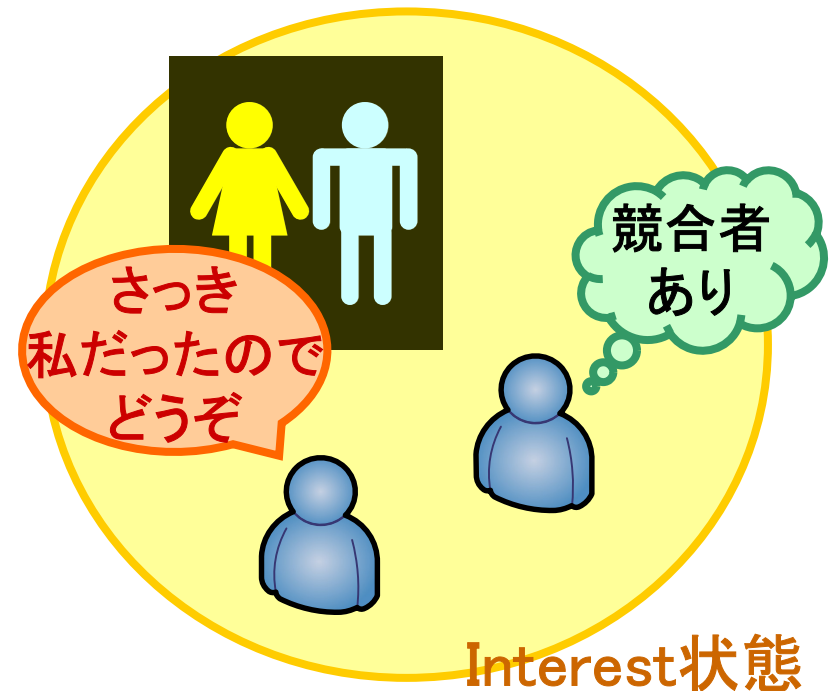
■ さまざまなアプローチ

- Dekkerのアルゴリズム
- セマフォ
- モニタ

■ Dekkerのアルゴリズム

- **Interest変数**により, 各プロセスがクリティカルセクションに興味があるか否かを表現
- **Priority変数**により, 複数プロセスが同時にクリティカルセクションに興味を持った場合にどちらを優先するかを決定

- ○ : ソフトウェアのみで実現可能
- × : ビジーウェイト



■ セマフォ

• P命令

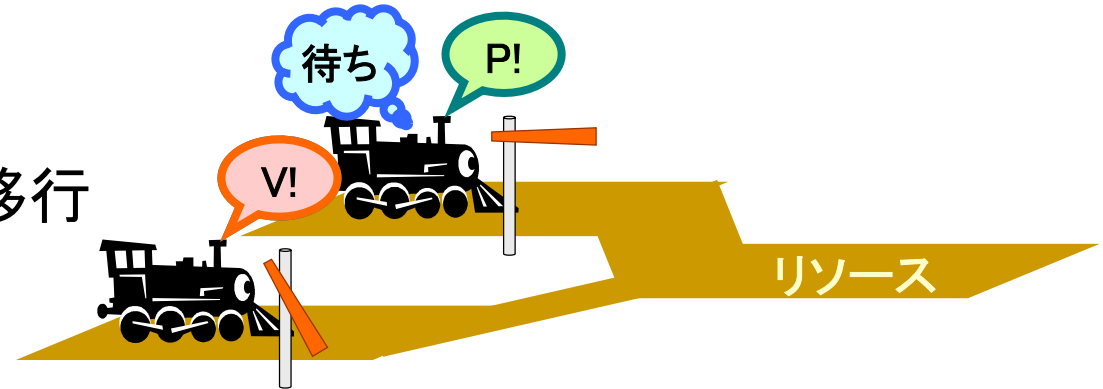
- リソース獲得要求,
失敗時には待ち状態に移行

• V命令

- リソース解放,
待ちプロセスを実行可能状態に

• リソース獲得が失敗すると、待ち行列へ

- V命令が起こしてくれるまで待てばよい
- ビジーウェイト不要
- プログラマ依存
- デッドロックの可能性 (Dining Philosopher)



■ モニタ

- オブジェクト指向
プログラミング
- カプセル化による
リソースの保護
- メソッド排他実行による
排他制御の簡潔化
- デッドロック可能性の
コンパイル時検出
(部分的)

```
class 自動車{  
public:  
    減速(n){  
        投入燃料量 = 0;  
        制動力 += n;  
        ブレーキランプ = on;  
        if( スリップ ){ ... }  
    }  
private:  
    投入燃料量;  
    制動力;  
    タイヤ角度;  
    ブレーキランプ;  
    :  
    :  
}
```



- 並行プロセスの話は今日で終わり
- 次回からは主記憶(メモリ)管理の話題

