# オペレーティングシステム
# 第7章 モニタ

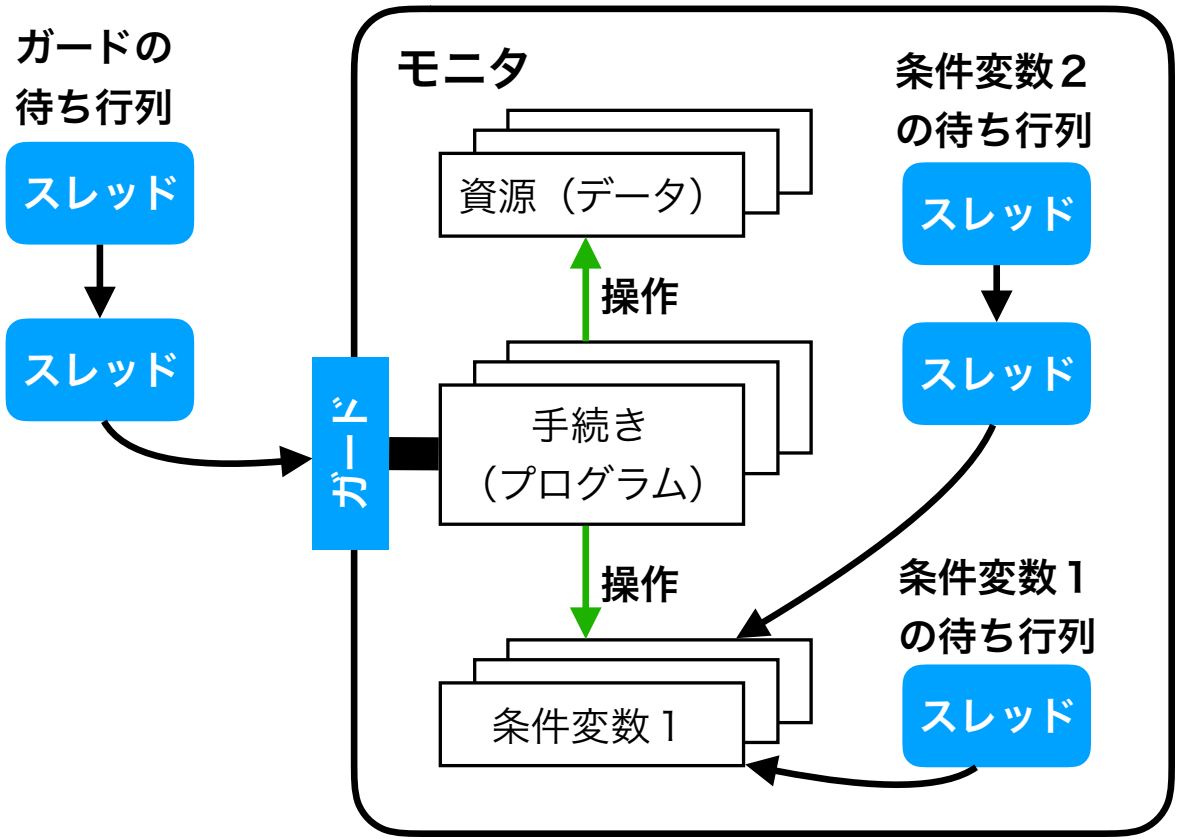
https://github.com/tctsigemura/OSTextBook

# モニタ（Monitor）の概要

プロセス間同期のために使用できる高級言語の機能のこと．
モニタは**抽象データ型**（Java 言語のクラスのようなもの）である．

- プログラマが定義できる型である．（抽象データ型で一般的）
- データと操作をまとめて定義する．（抽象データ型で一般的）
- 同期のための機能が組込まれている．（モニタ独特）

# モニタの模式図




- 資源（データ，変数），手続き（メソッド），ガード，条件変数
- 図では省略してあるが**初期化プログラム**

# モニタの構成要素

**資源（データ、変数）** 複数のスレッドによって共有される変数のことである．モニタの外から直接アクセスすることはできない．

**手続き（操作，メソッド）** 外部から呼び出されるプログラムである．手続きの実行は**ガード**の働きにより排他的に行われ，同時に実行される手続きは必ず一つ以内である．

**ガード** モニタに一つのガードが存在し，手続きを排他的に実行するために用いられる．

**条件変数** 条件変数には wait と signal の二つの操作ができる．wait 操作を行ったスレッドは**ガードを外して**条件変数の待ち行列に入る．signal 操作は条件変数の待ち行列から一つのスレッドを選んで実行可能にする．

**初期化プログラム** モニタのインスタンスを作成する時に，初期化のために実行されるプログラムである．

# モニタによる相互排除の実現（仮想言語）

```
1  // 銀行口座の残高を管理するモニタ
2  monitor MonAccount {
3    // 資源
4    int         money;                          // スレッド間の共有変数(残高)
5    // 初期化プログラム
6    MonAccount(int m) {
7      money = m;                                // 口座の残高を初期化する
8    }
9    // 手続き
10   public void receive(int r) {                // 入金手続き
11     money = money + r;
12   }
13   public void pay(int p) {                    // 引落し手続き
14     money = money - p;
15   }
16 }
```

- **資源**はスレッド間の共有変数 money である．
- **初期化プログラム**はモニタのインスタンス生成時に実行される．
- **手続き**（クリティカルセクション）は，ガードにより自動的に相互排除される．

# モニタによる相互排除の利用（仮想言語）

```
1  class MonAccontMain {
2    static MonAccount account = new MonAccount(0);   // 残高 0 円の口座を作る
3    static void receiveThread() {                    // 入金管理スレッド
4      for ( ; ; ) {                                  // 以下を繰り返す
5        int receipt = receiveMoney();                // ネットワークから入金を受信
6        account.receive(receipt);                    // 口座に入金する
7      }
8    }
9    static void payThread() {                        // 引落し管理スレッド
10     for ( ; ; ) {                                  // 以下を繰り返す
11       int payment = payMoney();                    // ネットワークから支払いを受信
12       account.pay(payment);                        // 口座から引落す
13   }
14   public static void main(String[] args) {
15     入金管理スレッドを起動;
16     引落し管理スレッドを起動;
17   }
18 }
```

- モニタのインスタンスを生成し，それに対して操作する．
- ガードは自動的なので排他忘れが無い．

# セマフォを用いた相互排除問題の解（参考）

```
int       account;                          // スレッド間の共有変数(残高)
Semaphore accSem = 1;                       // 初期値 1 のセマフォ accSem (account のロック用)
void receiveThread() {                      // 入金管理スレッド
  for ( ; ; ) {                             // 入金管理スレッドは以下を繰り返す
    int receipt = receiveMoney();           // ネットワークから入金を受信する
    P( &accSem );                           // account 変数をロックするための P 操作
    account = account + receipt;            // account 変数を変更する (クリティカルセクション)
    V( &accSem );                           // account 変数をロック解除するための V 操作
  }
}
void payThread() {                          // 引落し管理スレッド
  for ( ; ; ) {                             // 引落し管理スレッドは以下を繰り返す
    int payment = payMoney();               // ネットワークから支払い額を受信する
    P( &accSem );                           // account 変数をロックするための P 操作
    account = account - payment;            // account 変数を変更する (クリティカルセクション)
    V( &accSem );                           // account 変数をロック解除するための V 操作
  }
}
```

- P 操作と V 操作の使用はプログラマまかせ.
- 間違うとタイミング異存の発見の難しいバグになる.

# モニタによるキューの実現（仮想言語、前半）

```
1  monitor BoundedBuffer {
2      // 資源(リングバッファ)
3      int N;
4      int[] buf;
5      int first, last, cnt;
6      // 条件変数
7      Condition empty;
8      Condition full;
9      // 初期化
10     BoundedBuffer(int n) {
11         N = n;
12         buf = new int[N];
13         first = last = cnt = 0;
14     }
```

- この例ではリングバッファのデータ構造が資源である.
- 資源はモニタ外部から直接アクセスすることはできない.
- 条件変数 empty はキューが空の時，消費者スレッドが待合せに使用.
- 条件変数 full はキュー満杯時に，生産者者スレッドが待合せに使用.

# モニタによるキューの実現（仮想言語、後半）

```
15      // 手続き
16      public void append(int x) {     // (1)
17          if (cnt==N) full.wait();    // (1)
18          buf[last] = x;              // (3)
19          last = (last + 1) % N;      // (3)
20          cnt++;                      // (3)
21          empty.signal();             // (3)
22      }
23      public int remove() {           // (2)
24          if (cnt==0) empty.wait();   // (2)
25          int x = buf[first];         // (2)
26          first = (first + 1) % N;    // (2)
27          cnt--;                      // (2)
28          full.signal();              // (2)
29          return x;                   // (4)
30      }
31  }
```

- wait() はスレッドを状態変数の待ち行列に入れる.
- signal() は状態変数の待ちスレッドを起こす.
- signal() で起床したスレッドは**ただちに**実行される.

# モニタによる生産者と消費者問題の解（仮想言語）

```
1  class BoundedBufferMain {
2    static BoundedBuffer queue = new BoundedBuffer(10);  // 大きさ 10 のキュー
3    static void producer() {                             // 生産者スレッドが実行
4      while(true) {
5        int x = データを作る ();
6        queue.append(x);                                 // キューにデータを追加
7      }
8    }
9    static void consumer() {                             // 消費者スレッドが実行
10     while(true) {                                      // キューから
11       int x = queue.remove();                          //       データを取り出す
12       データを使用する (x);
13     }
14   }
15   public static void main(String[] args) {             // main から実行を開始
16     生産者スレッドを起動;
17     消費者スレッドを起動;
18   }
19 }
```

- モニタにより定義されたキューのインスタンスを使用した解

# モニタと同等なキューを Java のセマフォで実現（1/4）

```
1  import java.util.concurrent.Semaphore;              // セマフォ型を利用可能にする
2  public class SemBoundedBuffer {
3      private Semaphore guard = new Semaphore(1); // ガード用のセマフォ
4      private Semaphore next = new Semaphore(0);  // signal 時ブロック用セマフォ
5      private int nextCont = 0;                   // signal 時ブロック・スレッド数
6      private class Condition {                   // 内部クラス' 条件変数型' を定義
7          Semaphore sem = new Semaphore(0);       // 条件変数待ち用セマフォ sem
8          int count = 0;                          // 条件変数を待つスレッドの数
9          void await() {                          // 条件変数を待つメソッド
10             count++;                            // この条件変数待ちスレッドの数
11             if (nextCont>0) {                   // 起床後に await() した場合なら
12                 next.release();                 //   signal() したスレッドを起床
13             } else {                            // 起こすスレッドがないなら
14                 guard.release();                //   ガードを外してからブロック
15             }
16             sem.acquireUninterruptibly();       // 条件変数のセマフォで待つ
17             count--;                            // 待ちが完了
18         }
```

- Java のセマフォはカウンティングセマフォ（計数セマフォ）
- P 操作：`acquireUninterruptibly()`，V 操作：`release()`

# モニタと同等なキューを Java のセマフォで実現（2/4）

```
19          void signal() {                                  // 条件変数で待つスレッドを起床
20              if (count>0) {                               // 待っているスレッドがあれば
21                  nextCont++;                              //   signal 途中のスレッド数
22                  sem.release();                           //   待ちスレッドを起こす
23                  next.acquireUninterruptibly();           //   起きたスレッドを先に実行
24                  nextCont--;                              //   signal 完了
25              }
26          }
27      }
28      private void exitProc() {                            // 手続きの出口処理
29          if (nextCont>0) {                                // signal された後なら
30              next.release();                              //   signal したスレッドを起床
31          } else {                                         // そうでなければ
32              guard.release();                             //   ガードを外す
33          }
34      }
```

- 条件変数型を内部クラスとして定義
- `wait()` の代わりに `await()` を定義
- `exitProc()` は**手続き**の最後で呼出す.

# モニタと同等なキューを Java のセマフォで実現（3/4）

```
35      // 資源(リングバッファ)
36      private int N;
37      private int[] buf;
38      private int first, last, cnt;
39      // 条件変数
40      private Condition empty = new Condition();
41      private Condition full  = new Condition();
42      // 初期化
43      public SemBoundedBuffer(int n) {
44          N = n;
45          buf = new int[N];
46          first = last = cnt = 0;
47      }
```

- **資源**はクラス外から見えないように private にする.
- 前半で宣言した**条件変数型**の変数を二つ使用.
- **初期化プログラム**はクラスのコンストラクタで実現.

# モニタと同等なキューを Java のセマフォで実現（4/4）

```
48      // 手続き
49      public void append(int x) {                    // (1)
50          guard.acquireUninterruptibly();            // (1) ガードを取得
51          if (cnt==N) full.await();                  // (1)
52          buf[last] = x;                             // (3)
53          last = (last + 1) % N;                     // (3)
54          cnt++;                                     // (3)
55          empty.signal();                            // (3)
56          exitProc();                                // (3) 手続きの出口処理
57      }
58      public int remove() {                          // (2)
59          guard.acquireUninterruptibly();            // (2) ガードを取得
60          if (cnt==0) empty.await();                 // (2)
61          int x = buf[first];                        // (2)
62          first = (first + 1) % N;                   // (2)
63          cnt--;                                     // (2)
64          full.signal();                             // (2)
65          exitProc();                                // (4) 手続きの出口処理
66          return x;                                  // (4)
67      }
68  }
```

- モニタなら自動的に実行されるものも明示

# Java による生産者と消費者問題の解（前半）

```
1  public class MonBoundedBuffer {
2      // 資源(リングバッファ)
3      private int N;
4      private int[] buf;
5      private int first, last, cnt;
6      // 初期化
7      public MonBoundedBuffer(int n) {
8          N = n;
9          buf = new int[N];
10         first = last = cnt = 0;
11     }
12     // 手続き
13     private void await() {
14         try{wait();}catch(InterruptedException e){}
15     }
```

- 資源には private を明示
- 初期化はコンストラクタにより実現
- Java の wait() は try-catch 構文で使用する必要がある.

# Java による生産者と消費者問題の解（後半）

```
16      public synchronized void append(int x) {     // (1)
17          while (cnt==N) await();                  // (1)
18          buf[last] = x;                           // (3)
19          last = (last + 1) % N;                   // (3)
20          cnt++;                                   // (3)
21          if (cnt==1) notify();                    // (3)
22      }
23      public synchronized int remove() {           // (2)
24          while (cnt==0) await();                  // (2)
25          int x = buf[first];                      // (2)
26          first = (first + 1) % N;                 // (2)
27          cnt--;                                   // (2)
28          if (cnt==N-1) notify();                  // (2)
29          return x;                                // (2)
30      }
31  }
```

- **手続き**は synchronized 修飾したメソッド
- Java の wait() は別の理由でも終了するので while の中で使用
- 条件変数は一つしかないので，利用方法に工夫が必要
- remove() 最後の行の実行順序がモニタと異なる．