経営情報システム学特論１
４．モニター

SS専攻 経営情報システム学講座 客員
石川 冬樹
f-ishikawa@nii.ac.jp

目次

- モニター（LTSA版）
- モニター（Java版）
- セマフォ

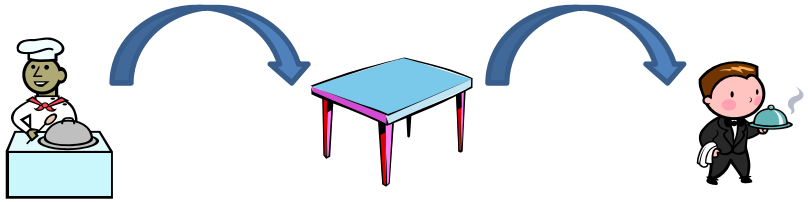
（再）モニター

- モニター：あるオブジェクトを監視することにより，条件が成り立つまで待つといった振る舞いを実現する機構
  - 並行処理を記述するプログラミング言語において，何かしらの形で表現されている
  - 前回のJavaにおけるsynchronizedは，「他のスレッドの実行が終わり自分がロックを確保できるまで待つ」
  - そのプログラムならではの条件で待ちたいことも当然ある（例：料理ができるまで）
  - Javaなどプログラミング言語であれば，待つときはCPUを消費せずに適切にスリープする動作をすべき

（再）例題：生産者と消費者

- 例：大人数への料理を次々と作る人と運ぶ人
  - 「作る人」は，置き場に料理を置いていく
    ※ 置き場がいっぱいになったら待ちになる
  - 「運ぶ人」は，置き場から料理を持って行く
    ※ 置き場が空になったら待ちになる

※ それぞれ複数人いる場合も考える



（再）例題： 生産者と消費者

今ならLTSA版を簡単に書ける？

- 置き場 RESOURCES[i:0..N]
  - get/putを受け付けて数を増減
  - 0のときのget，Nのときのputは ERRORに遷移
    （単に未定義にしておいてもERROR扱いになる）
- 生産者 PRODUCER，消費者 COMSUMER
  - それぞれput，getを適切に発行
- これらプロセスの合成からなるシステム
  - まずは生産者と消費者1プロセスずつ
  - 消費者が2プロセスの場合も試してみる

生産者と消費者（LTSA版１）

- 簡単な不適切版（Safety Checkに通らない）

```
const N = 2

RESOURCE = RESOURCE[0],
RESOURCE[u:0..N] = (
  put -> RESOURCE[u+1]
  | get -> RESOURCE[u-1]
).

PRODUCER = ( put -> PRODUCER ) +{get,put}.

CONSUMER = ( get -> CONSUMER ) +{get,put}.

||SYSTEM = ( {p1,c1,c2}::RESOURCE || p1:PRODUCER
                    || c1:CONSUMER || c2:CONSUMER ).
```

生産者と消費者（LTSA版２）

- 簡単すぎる修正版：生産者か消費者が複数の場合Safety Checkを通らない（理由はわかるは

```
…

RESOURCE = RESOURCE[0],
RESOURCE[u:0..N] = (
  read[u] -> RESOURCE[u]
  | put -> RESOURCE[u+1]
  | get -> RESOURCE[u-1]
).

PRODUCER = ( read[v:0..N-1] -> put -> PRODUCER ) +{read[0..N],get,put}.

CONSUMER = ( read[v:1..N] -> get -> CONSUMER ) +{read[0..N],get,put}.

…
```

生産者と消費者（LTSA版３）

- 再修正版：生産者または消費者が単数でもデッドロック！（なぜ？）

```
…

PRODUCER = ( lock -> read[v:0..N-1] -> put -> unlock -> PRODUCER )
                    +{read[0..N],get,put}.

CONSUMER = ( lock -> read[v:1..N] -> get -> unlock -> CONSUMER )
                    +{read[0..N],get,put}.

LOCK = ( lock -> unlock -> LOCK ).

||SYSTEM = ( {p1,c1,c2}::RESOURCE || p1:PRODUCER || c1:CONSUMER ||
c2:CONSUMER || {p1,c1,c2}::LOCK ).

…
```

演習：生産者と消費者（LTSA版）

- ここまでの再修正版の反例を見るなどして，なぜうまくいかないか，どう修正すべきか検討してみよ

- 次スライドは解答

演習解答：生産者と消費者（LTSA版）

- ロックを確保してから，現在の値に応じて待ちに入ることがある
  - 消費者の場合，現在の値が0のときreadでブロック

```
CONSUMER = ( lock -> read[v:1..N] -> get -> unlock -> CONSUMER )
                +{read[0..N],get,put}.
```

  - 生産者がputをするまで待つことになるが，ロックを持ったまま待っているので，生産者はputできない（デッドロック）
  - ➡進めなかった場合にはロックを手放せばよい

```
CONSUMER = ( lock -> (
  read[v:1..N] -> get -> unlock -> CONSUMER
  | read[0] -> unlock -> CONSUMER
)) +{read[0..N],get,put}.
```

目次

- モニター（LTSA版）
- モニター（Java版）
- セマフォ

Java版

- LTSA版：概念的なモデル
  - 言語内に「条件が成り立つまで・ラベル同期できるまで待つ」という振る舞いが含まれている
    - when(i>0) や read[1..N]など
- Java版：効率等意識した動かし方の指示
  - if文で条件を確認して，必要なら「待つ」という条件分岐を書くことになる
  - ただし，「待つ」ときには，CPUを使わずにスリープすべき
  - ポーリングによる無駄なCPU利用を避けるためには，条件が成り立ったら「起こしてもらう」のがよい

Java版

- 以降のJavaコードでは簡単のため，「置き場」の上限は考えていない
  - LTSAだとどうしても有限
  - Javaでも厳密には有限だが普通は気にならないくらい十分に大きい

生産者と消費者（Java版１）

- 配布JavaコードResource1
  - 下記はエラー処理等を省略

```
synchronized void produce(){
  num++;
}

synchronized void consume(){
  num--;
}
```

Javaでのモニター機構

- Objectクラスで実装されている（つまりすべてのクラスで使える）メソッド
  - wait： 該当オブジェクトに関し自分が保持しているロックを一時解放して待つ
  - notify： 該当オブジェクトに関し待ち状態にあるスレッドを1つ起こす
  - notifyAll：該当オブジェクトに関し待ち状態にあるスレッドをすべて起こす

生産者と消費者（Java版２）

- 配布JavaコードResource2
  - うまくいく？

```
synchronized void produce(){
  num++;
  notify();
}

synchronized void consume(){
  if(!(num > 0)){
    wait();
  }
  num--;
}
```

留意事項

- LTSA版３での問題は起きていない？
  - ロックを持ったままwaitを呼んでいるが？
- 数頁前の再掲
  - wait： 該当オブジェクトに関し自分が保持しているロックを一時解放して待つ

生産者と消費者（Java版３）

- 配布JavaコードResource3
  - notifyAllに変えてみると何が起きるか？

```
synchronized void produce(){
  num++;
  notifyAll();
}

synchronized void consume(){
  if(!(num > 0)){
    wait();
  }
  num--;
}
```

Javaでのモニター機構

- 一般的には下記の使い方をする

同期が必要

```
public synchronized void act()
    throws InterruptedException{
  while (!cond){
    wait();
  }
  // modify the monitor data
  notifyAll(); // or notify();
}
```

ifではなくwhile
（起こされた後も再チェック）

目次

- モニター（LTSA版）
- モニター（Java版）
- セマフォ

セマフォ

- 先のResourceのようなデータ変数は，*セマフォ（Semaphore）*として広く知られている
  - 個数だけをカウント
  - セマフォをインクリメントするV操作とデクリメントするP操作
    （P操作はセマフォが0のときにはブロックする）

※ ダイクストラによるもの

適切な？Java版

- 配布Javaコード Resource4
  - 下記はエラー処理等を省略

```
private Semaphore
   sem = new Semaphore(0);

synchronized void produce(){
   sem.release()
}

synchronized void consume(){
   sem.acquire();
}
```

議論

- 先のセマフォの例で，synchronizedを付けるとデッドロックする
  - （再掲）
    wait： 該当オブジェクトに関し自分が保持しているロックを一時解放して待つ
  - 今までの例では付けてもよくて，今回の例では付けてはならない理由を説明せよ
    （配布コードでは範囲を絞って付けている）

生産者と消費者の実用

- 通常は，先の問題のようにテーブルが無限に料理を置けるとはしない
  - 上限まで埋まっている場合，生産者も待たされる
- さらに複数消費者はFIFOになるようにすべき
- 有限長のキューを考えることが多い
  （BoundedQueue）

まとめ

- モニター
  - 資源を監視し，ある条件が成り立つまで待つといった振る舞いを実現する
  - JavaではwaitとnotifyAll()メソッドにより実現される
  - 資源の個数を管理するセマフォが非常に典型的な概念である
  - wait時には確保済みの資源ロックを解放するようになっているが，どのオブジェクトに関するロックかを意識しないとデッドロックを引き起こせる
- 次回： デッドロックをより踏み込んで扱う