経営情報システム学特論１
３．相互排他

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

- 共有変数の更新
- LTSAでのモデル化と検証
- 生産者と消費者

並行システムの基本例題（再）

- グローバル変数などを更新する簡単な関数

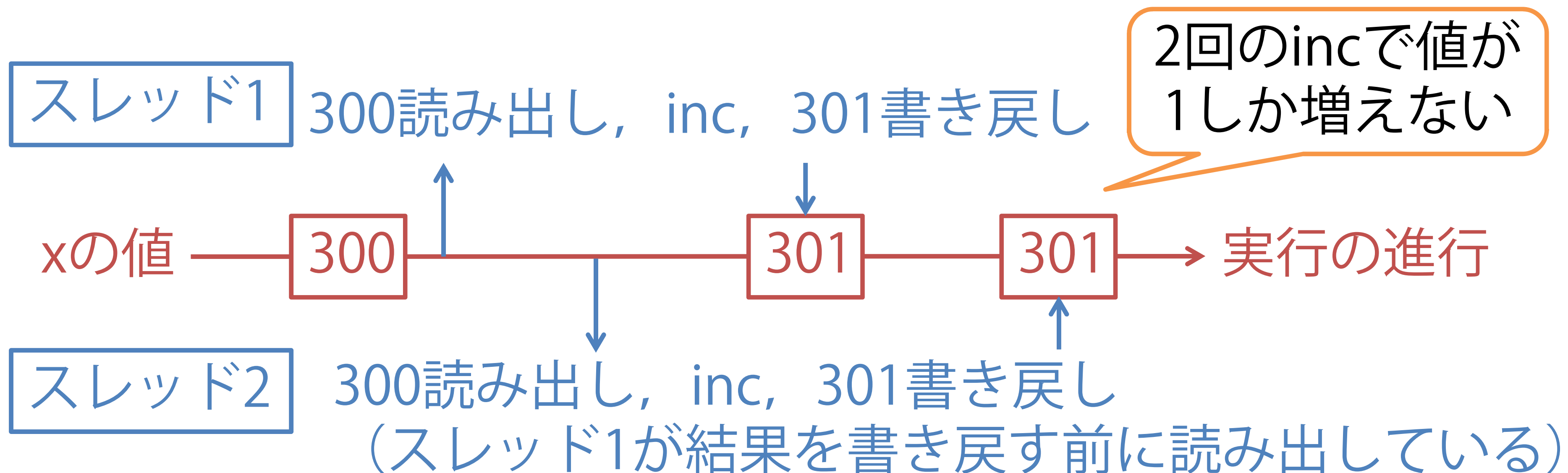
```
void increment(){
  x = x + 1;
}
```

- 仕様もなんてことない
  「実行後のxの値は，実行前より1大きい」
  （オーバーフローしない前提がおけるならば）
- これをマルチスレッド化すると・・・

並行システムの基本例題（再）

- テスト（付録 IncrementTest.java）
  - 変数xの初期値を 0 とする
  - 前スライドのincrementを1000回実行するスレッドを2個作成し，並列に動作させてみる

➡ 結果は2000より小さい「ことがある」



もう少しだけ意味のありそうな設定

- 東門と西門からの入場者をカウント
  - 閉館時に全員が出たかを確認

相互排他

- *相互排他（Mutual Exclusion）*

  あるいは*排他制御*

  - あるプロセスに独占的に資源を利用させ，その間は他のプロセスが利用できないようにする

    （共有資源にアクセスするクリティカルセクションに同時に入らないようにする）

Javaでの相互排他

- 対象オブジェクトに対してロックを確保した1つのスレッドのみが実行可能とする

```
void increment(){
  synchronized(this) {
    x = x + 1;
  }
}
```

```
synchronized void increment(){
  x = x + 1;
}
```

Javaでの相互排他

- synchronizedは，あるオブジェクトに対してロックを確保する
  - 先ほどの例では sychronized(this) と書いていた
  - メソッドをsynchronizedメソッドにする場合，ロックの対象はそのオブジェクト（this）になる

目次

- 共有変数の更新
- LTSAでのモデル化と検証
- 生産者と消費者

LTSAでの共有変数モデル化

- 変数の読み書きの表現

```
const N = 2
range T = 0..N

VAR      = VAR[0],
VAR[u:T] = ( read[u] -> VAR[u]
           | write[v:T] -> VAR[v]
           | write[N+1] -> VAR[0] ).
```

LTSAでの共有変数モデル化

- イベントを数えインクリメントを行うプロセス

```
const N = 2
range T = 0..N

VAR = VAR[0],
VAR[u:T] = ( read[u] -> VAR[u]
         | write[v:T] -> VAR[v]
         | write[N+1] -> VAR[0] ).

INCREMENT = ( event -> read[x:T] -> write[x+1] -> INCREMENT )
            +{read[T],write[T]}.

||SYSTEM = ( VAR || INCREMENT ).
```

アルファベット拡張（後述）

LTSA補足： Alphabet

- プロセスに含まれるラベルの集合をAlphabetという
- 複数プロセス間で共通するラベルは共有アクションと見なされる
  - たまたま特定のラベルを使うかどうかで，共有アクションになるかが決まってしまう
  - ⇨ 前スライドでは，アルファベット拡張により，とにかくすべて共有アクションと見なすようにしている
    （write[1] は同期して特定タイミングで起きるが，write[0] は好き勝手に起きうる，といったことを避ける）

演習： LTSAでの共有変数モデル化

- 復習： INCREMENTプロセスが2つとなるように書き換えてみよ
（いずれもVARをread/writeするように）
  - そろそろ図示では確認できない規模

- 次スライドは解答

演習解答：LTSAでの共有変数モデル化

```
const N = 2
range T = 0..N

VAR = VAR[0],
VAR[u:T] = ( read[u] -> VAR[u]
           | write[v:T] -> VAR[v]
           | write[N+1] -> VAR[0] ).

INCREMENT = ( event -> read[x:T] -> write[x+1] -> INCREMENT )
            +{read[T],write[T]}.

||SYSTEM = ( {t1,t2}::VAR || t1:INCREMENT || t2:INCREMENT ).
```

LTSAでの検証

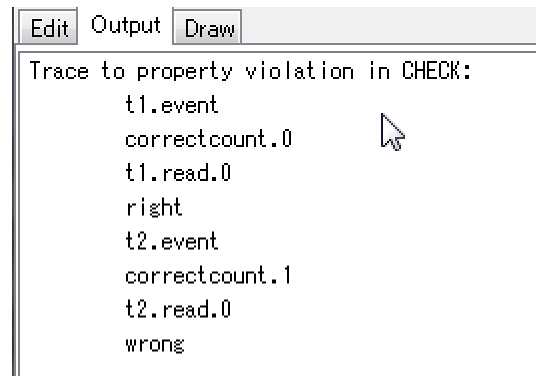
- 例：正解を（神様視点で）取得，保持し，変数からの読み取り値と常に比較をする「見張り」を並行に走らせる

```
CHECK = CHECK[0],
CHECK[u:T] =
 ( {t1.event,t2.event} -> check[u] ->
  ( t1.read[v:T] ->
   ( when (u!=v) wrong -> ERROR
   | when (u==v) right -> CHECK[u+1] )
  | t2.read[v:T] ->
   ( when (u!=v) wrong -> ERROR
   | when (u==v) right -> CHECK[u+1] )
  )
 ),
CHECK[N+1] = CHECK[0].
||SYSTEM = ( {t1,t2}::VAR || t1:INCREMENT || t2:INCREMENT || CHECK ).
```

ERRORは言語上のキーワード

LTSAの使い方（３）

- Checkメニューから Safety
  - ERRORに到達することがないか検証，もしある場合は反例が表示される



モデル検査の基礎： 検証する性質（再）

Reachability *到達可能性*	ある条件の下ある状況に到達しうる 「初期画面に戻る操作の列が常にある」
Safety *安全性*	ある条件の下ある状況に到達することがない 「踏切が空いた状態で電車が通過することはない」
Liveness *活性*	ある条件の下ある状況にいつか必ず到達する 「登録をするといつか必ず確認メールが届く」
Deadlock-freeness	デッドロックが起きない
Fairness *公平性*	ある条件の下ある状況が無限回起きる（ない） 「ユーザが無数にリクエストを送れば無数に返事が来る（他のユーザ等にブロックされ続けることはない）」

演習： ロックのモデル化

- Javaにおけるsynchronizedによる制御に相当する制御をモデルに含め，Safety Checkをパスするようにモデルを変更してみよ

- 次スライドは解答

演習解答： ロックのモデル化

```
…

INCREMENT = ( event -> get ->
              read[x:T] -> write[x+1] ->
              put ->  INCREMENT )
             +{read[T],write[T]}.

LOCK = ( get -> put -> LOCK ).

…
```

考察

- ロックを確保するモデルにおいて，INCREMENT内の任意のタイミングでSTOPする可能性を考慮するとどういう不適切な状況が発生するか？

注：JavaのThreadクラスにおいては，suspend/stopメソッドがdeprecatedとなっている

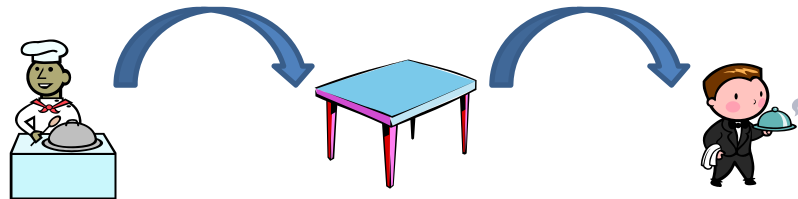
目次

- 共有変数の更新
- LTSAでのモデル化と検証
- <u>生産者と消費者</u>

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



演習： 生産者と消費者（次週に向け）

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

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

まとめ

- 相互排他
  - あるプロセスに独占的に資源を利用させ，その間は他のプロセスが利用できないようにする
  - ⇨ 不整合を避けるが，「待ち」のオーバーヘッドがあるため最低限としたい（と言う難しさ）
  - Javaではsynchronizedキーワードによりロック確保・解放を直接書かずに行うことができる
  - LTSAでは定義された不整合が発生しないことをSafety（安全性）として検証することができる

- 次回： モニター機構を扱う