

# SPINで弱いメモリ順序のメモリモデルでの プログラムの実行をモデル検査するための ライブラリの改良

松元稿如（高知工科大学）

鵜川始陽（高知工科大学）

安部達也（千葉工業大学人工知能・  
ソフトウェア技術研究センター）

# 概要：背景

- モデル検査器SPINはマルチスレッドプログラムの検査に使用されている
- マルチスレッドプログラムの検査ではアウトオブオーダー実行を考慮する必要がある
- SPINはインオーダー実行しか検査できない

# 概要：貢献

- アウトオブオーダー実行を  
検査できるSPIN用ライブラ  
リを開発した

```
#define PROCSIZE 2
#define VARSIZE 2
#define BUFFSIZE 1
#include "tso.h"
#define s0 0
#define s1 1

proctype A() {
    int a;
    WRITE(s0, 1);
    a = READ(s1);
}

proctype B() {
    int b;
    WRITE(s1, 1);
    b = READ(s0);
}
```

# 目次

- 背景と目的
- 開発したライブラリの使い方
- ライブラリの実装
  - アウトオブオーダー実行
  - 検査する性質を表す式
- 性能評価
- 関連研究
- まとめ

# マルチスレッドプログラムの実行

各スレッドの命令がインターリーブ実行される

共有変数	スレッドA	実行例
s0	s0 = 1;	スレッドA s0 = 1;
s1	a = s1;	スレッドB s1 = 1;
ローカル変数	スレッドB	スレッドA a = s1;
a	s1 = 1;	スレッドB b = s0;
b	b = s0;	

成り立って欲しい性質：

a, bのどちらかは実行完了までに1になる

# マルチスレッドプログラムの実行

各スレッドの命令がインターリーブ実行される

共有変数

s0

s1

ローカル変数

a

b

スレッドA

```
s0 = 1;
```

```
a = s1;
```

スレッドB

```
s1 = 1;
```

```
b = s0;
```

実行例

スレッドA

```
s0 = 1;
```

スレッドB

```
s1 = 1;
```

スレッドA

```
a = s1;
```

スレッドB

```
b = s0;
```

成り立って欲しい性質：

全6パターンの実行

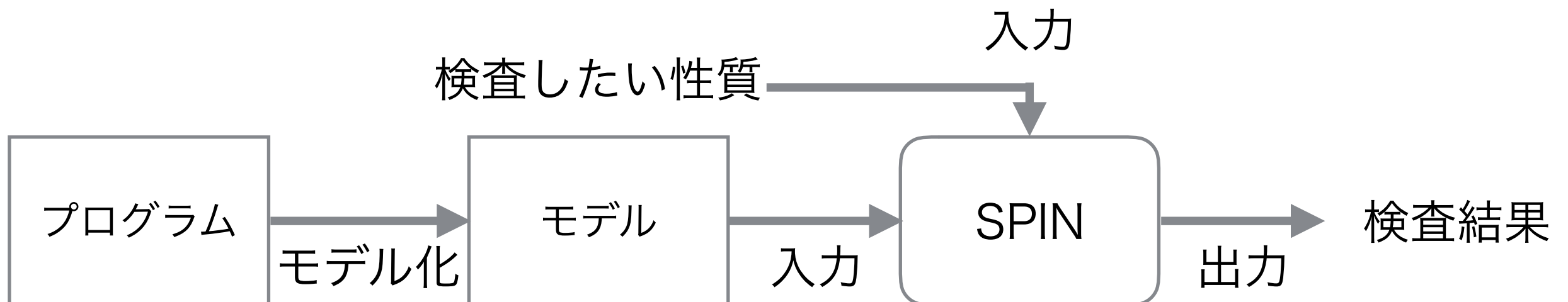
a, bのどちらかは実行完了までに1になる

# SPIN[Holzmann, '91]

- マルチスレッドプログラムのモデル検査に使用される

モデル検査：モデルの全実行パターンを検査する

モデル：プログラムをモデル記述用の言語Promelaで書き換えたもの

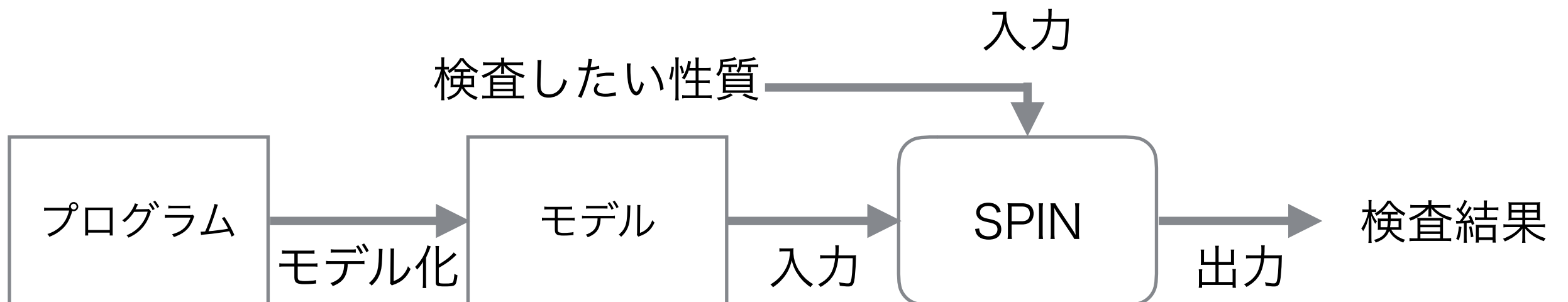


# SPIN[Holzmann, '91]

- マルチスレッドプログラムのモデル検査に使用される

モデル検査：モデルの全実行パターンを検査する

モデル：プログラムをモデル記述用の言語Promelaで書き換えたもの ➡ プロメラ





# Promelaモデル

共有変数

s0  
s1

ローカル変数

a  
b

スレッドA

s0 = 1;  
a = s1;

スレッドB

s1 = 1;  
b = s0;



```
int s0;  
int s1;  
  
proctype A() {  
    int a;  
    s0 = 1;  
    a = s1;  
}  
  
proctype B() {  
    int b;  
    s1 = 1;  
    b = s0;  
}
```



プロセス

- proctypeでスレッドをモデル化

# アウトオブオーダー実行

- CPUが最適化のために命令の順序を入れ替えて実行すること
  - メモリアクセスの順序がリオーダーリングされる場合がある
  - 命令を実行したスレッドはプログラム通りの実行結果を観測する
  - 別のスレッドからは入れ替わった実行結果を観測できる場合がある

# アウトオブオーダー実行の影響

	スレッドA	スレッドB
各スレッドの実行順序	<pre>s0 = 1; a = s1;</pre>	<pre>s1 = 1; b = s0;</pre>
スレッドBからの見え方	<pre>a = s1; s0 = 1;</pre>	<pre>s1 = 1; b = s0;</pre>
性質が成り立たない実行例	<pre>a = s1; s1 = 1; b = s0; s0 = 1;</pre>	

成り立って欲しい性質：

a, bのどちらかは実行完了までに1になる **とは限らない**

# メモリモデル

- アウトオブオーダー実行される場合のあるメモリアクセスの順序を定義したもの
- CPUのアーキテクチャ毎に異なる

# メモリモデルに従った実行の検査

- アウトオブオーダー実行によって実行結果が変化する場合があります
- 入れ替わり得る順序はメモリモデルで定義されている



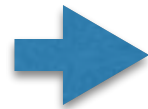
- メモリモデルに従った実行を検査する必要がある

# SPINの問題点

- SPINはメモリモデルに従った実行を検査しない
- 検査しようとする、メモリアクセスの順序のリオーダリングを含めてモデル化する必要がある
  - 複雑なプロメラになり、間違ったプロメラを作ってしまう  
やすい

# メモリモデルに従った実行を考慮したプロメラ

```
int s0;  
int s1;  
  
proctype A() {  
    int a;  
    s0 = 1;  
    a = s1;  
}  
  
proctype B() {  
    int b;  
    s1 = 1;  
    b = s0;  
}
```



```
int s0;  
int s1;  
  
proctype A() {  
    int a;  
    if  
        ::true ->  
            s0 = 1;  
            a = s1;  
        ::true ->  
            a = s1;  
            s0 = 1;  
    fi;  
}  
  
proctype B() {  
    int b;  
    if  
        ::true ->  
            s1 = 1;  
            b = s0;  
        ::true ->  
            b = s0;  
            s1 = 1;  
    fi;  
}
```

# 目的

- SPINでメモリモデルに従った実行を検査する
- ユーザが直接Promelaを記述することを支援する



メモリモデルに従ったメモリアクセスを提供するライブラリを開発する



# ライブラリが提供するもの

- 共有変数のプロメラ
  - 共有変数は整数を使って識別する
- 共有変数にアクセスするマクロ
  - `WRITE (s, v)`
    - 共有変数`s`に`v`を書き込む
  - `READ (s)`
    - 共有変数`s`の値を返す

# ライブラリの使用例

```
int s0;
int s1;

proctype A() {
    int a;
    s0 = 1;
    a = s1;
}

proctype B() {
    int b;
    s1 = 1;
    b = s0;
}
```



```
#define PROCSIZE 2
#define VARSIZE 2
#define BUFFSIZE 1
#include "tso.h"
#define s0 0
#define s1 1
```

```
proctype A() {
    int a;
    WRITE(s0, 1);
    a = READ(s1);
}
```

```
proctype B() {
    int b;
    WRITE(s1, 1);
    b = READ(s0);
}
```

プロメラのパラメタ

ライブラリ

共有変数の対応付け

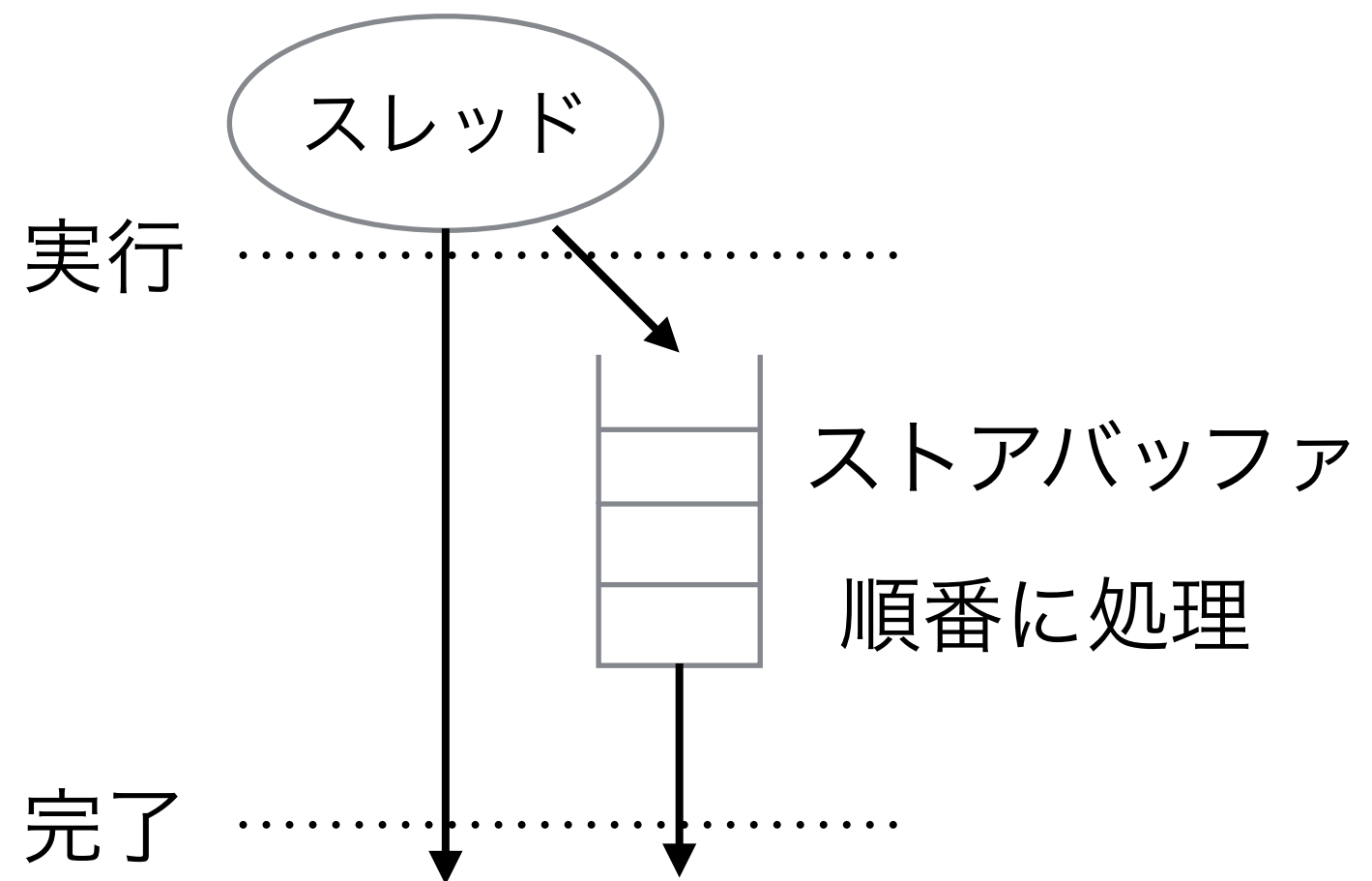
- インクルードするライブラリによってマクロの中身が異なる

# ライブラリの対応状況

- トータルストアオーダリング(TSO)
  - WRITE→READが入れ替わる場合がある
- パーシャルストアオーダリング(PSO)
  - WRITE→READが入れ替わる場合がある
  - WRITE→WRITEが入れ替わる場合がある

# CPUによる高速化

- メモリアクセスの速度はCPUの速度より遅い
- TSO、PSOではストアバッファを利用して高速化している



# CPUによる高速化をモデル化

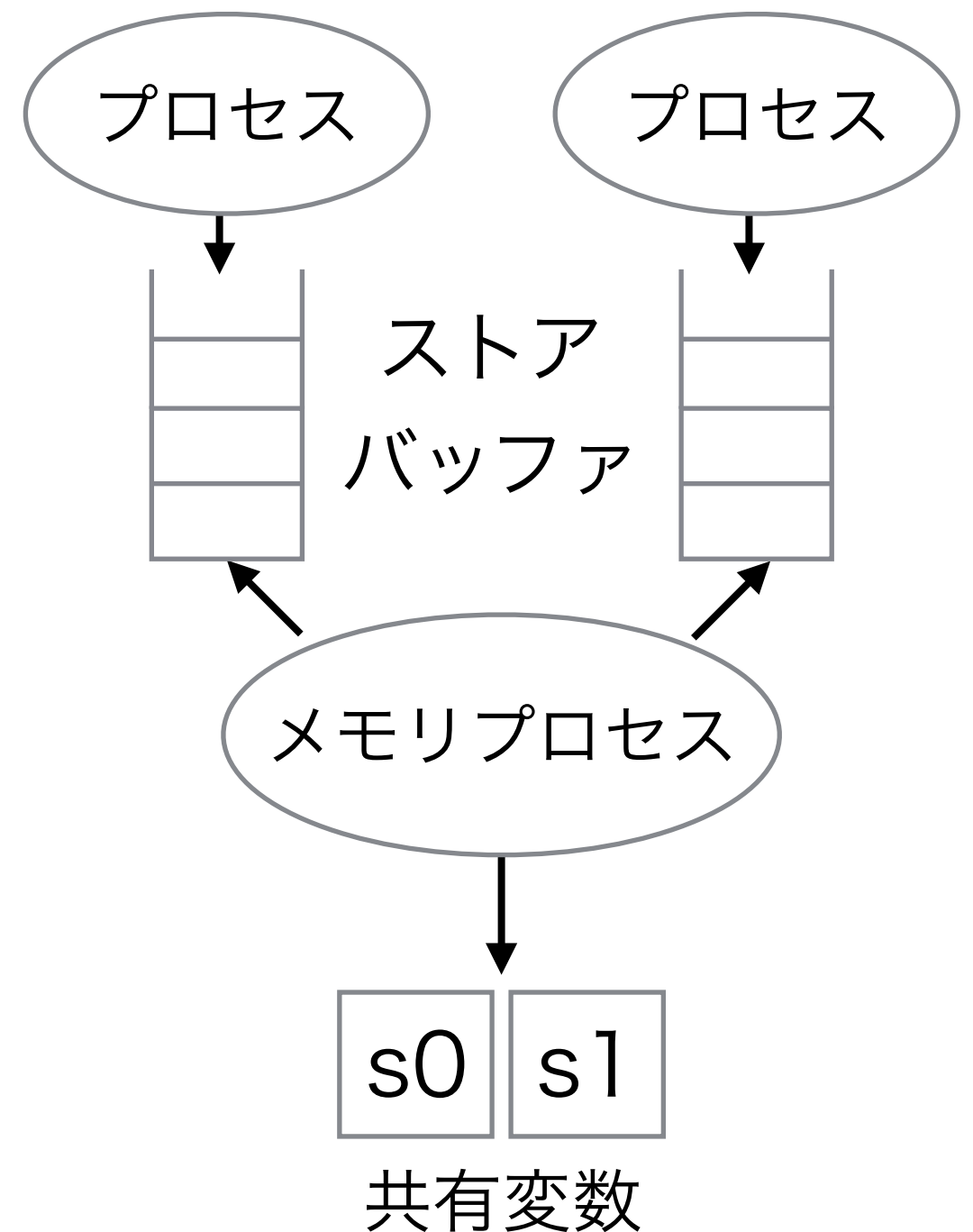
ストアバッファ

- チャンネルでモデル化

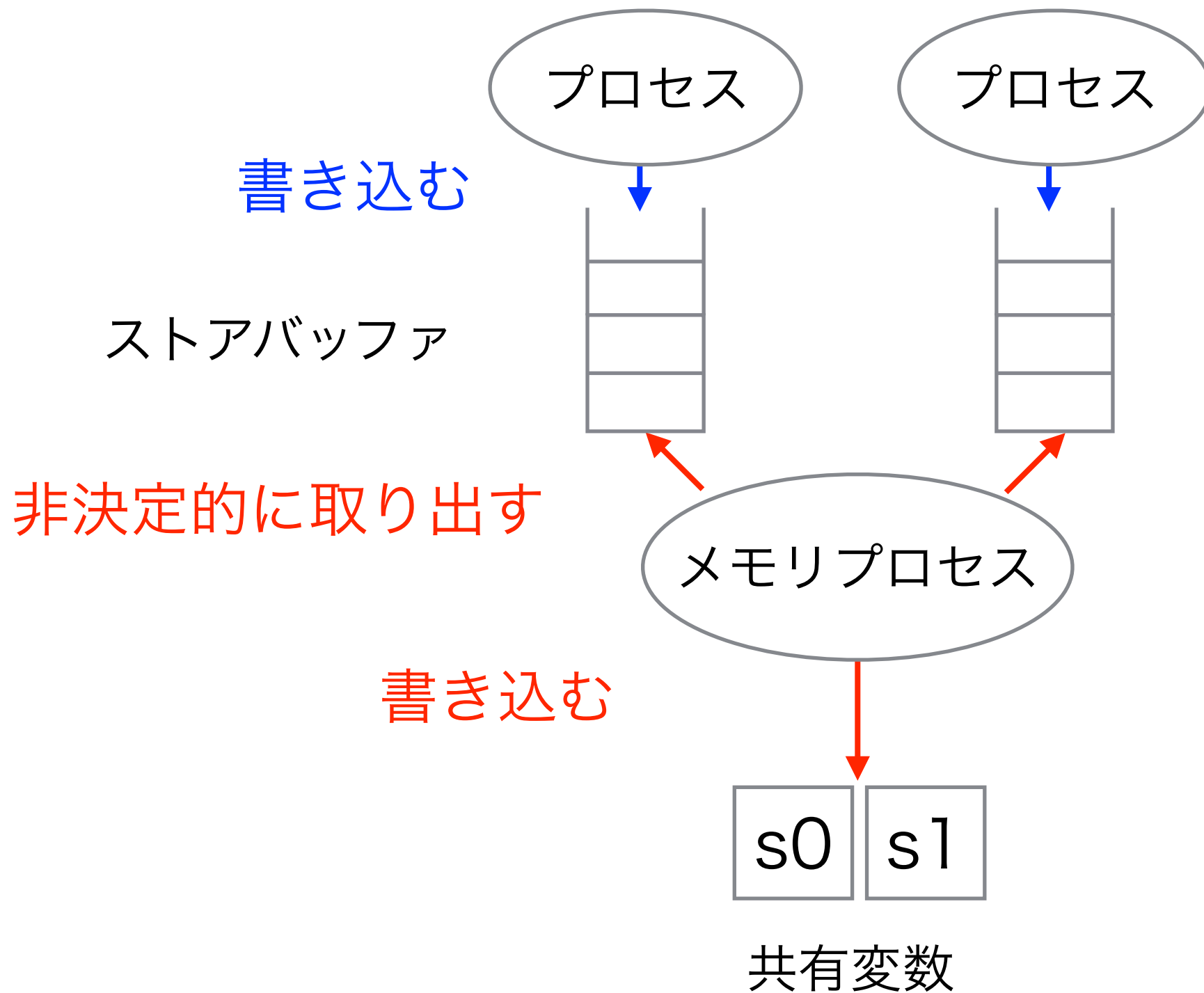
チャンネル：FIFO形式のキュー

メモリプロセス

- ストアバッファの中身を反映する



# TSOのプロメラ



# WRITEの実装(TSO)

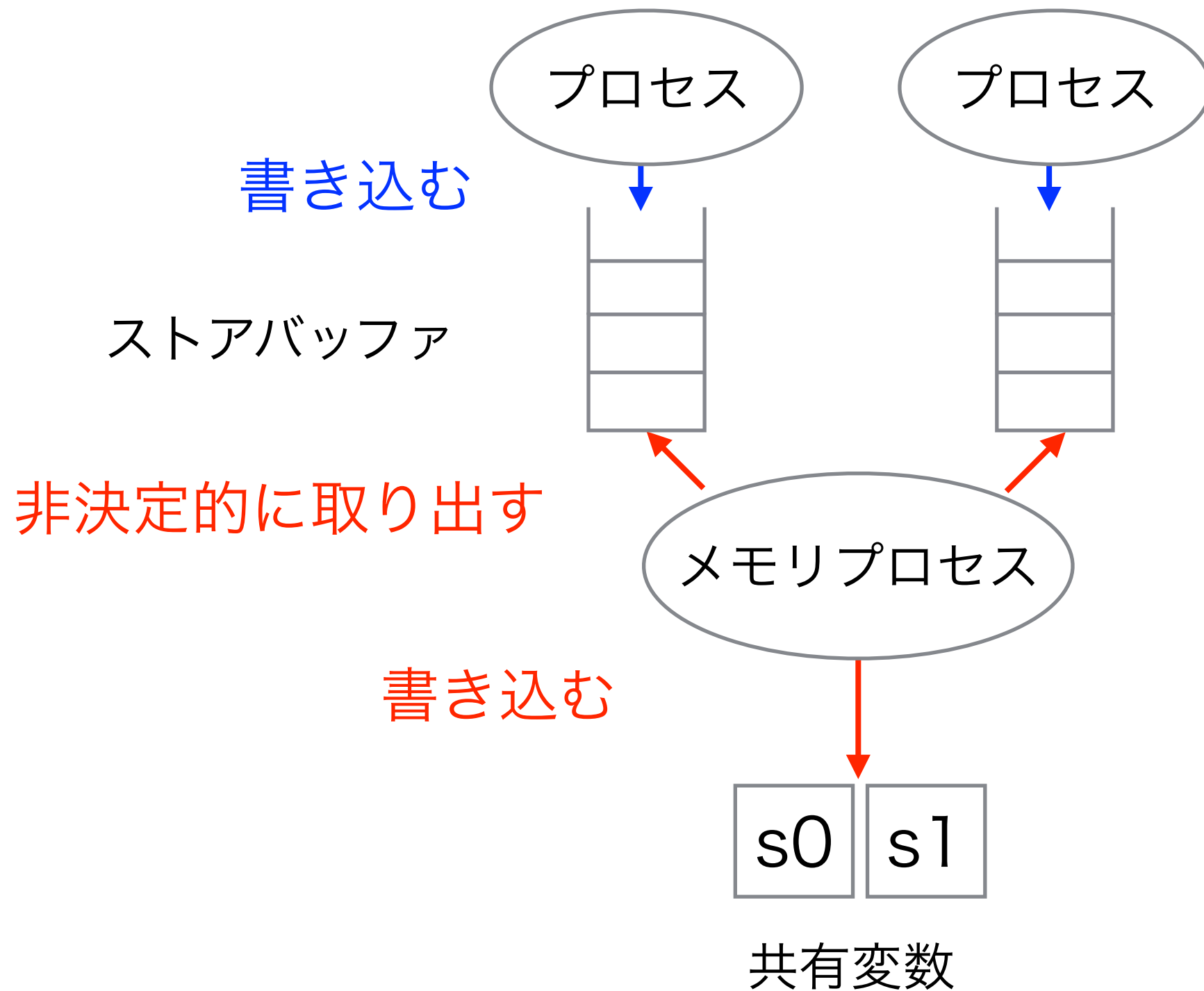
- $v$ に直接式が展開される
- READの値はライブラリが管理する変数の値に依存する
- 実行中にREADの値が変わってしまうかもしれない

```
#define WRITE(s, v) \
```

```
int tmp = v; \
store_buffer[_pid]!s, tmp; \
copy[_pid * VARSIZE + (s)] = tmp; \
counter[_pid * VARSIZE + (s)]++;
```

\_pid: プロセスID

# TSOのプロメラ



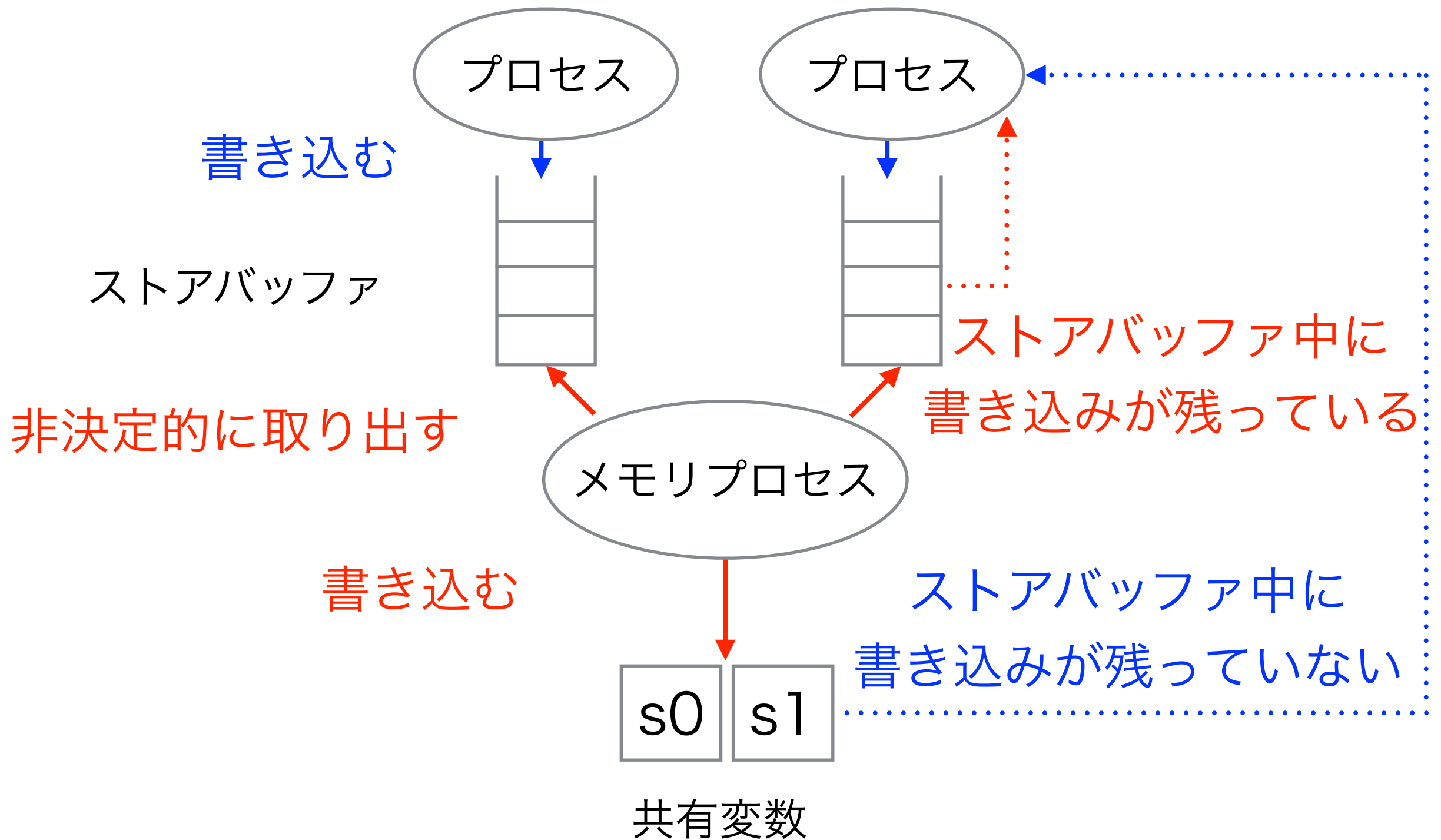


# メモリプロセスの実装(TSO)

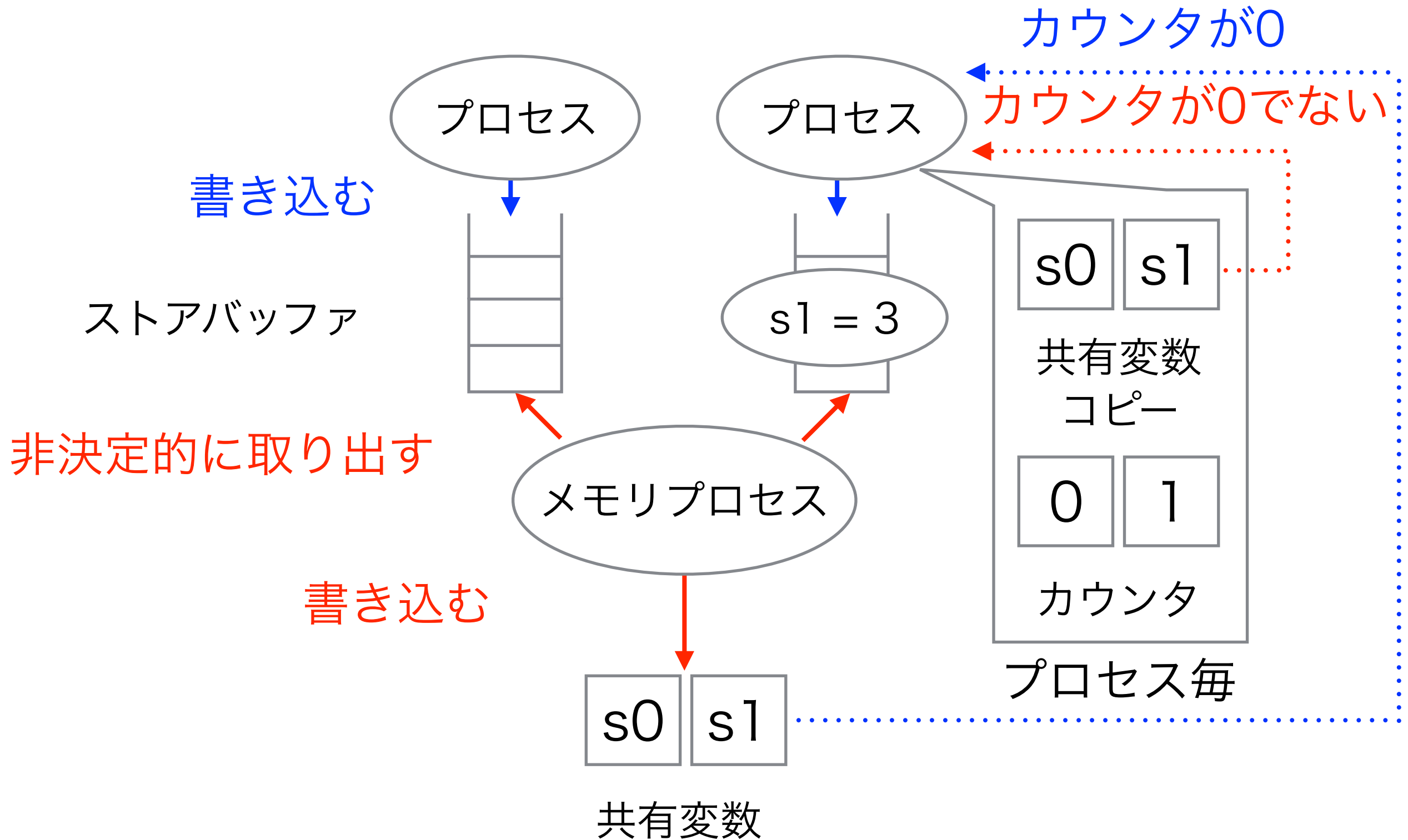
- ①：ストアバッファの中身を共有変数に反映
- ②：次のストアバッファへ移る

```
do
  :: (len(store_buffer[i + 2]) > 0) -> } ①
    COMMIT_WRITE(i + 2); break;
  :: (i < PROCSIZE - 1 &&
    r > len(store_buffer[i + 2])) -> } ②
    r = r - len(store_buffer[i + 2]);
    i++;
od;
```

# TSOのプロメラ



# TSOのプロメラ



# WRITEの実装(TSO)

- $v$ に直接式が展開される
- READの値はライブラリが管理する変数の値に依存する
- 実行中にREADの値が変わってしまうかもしれない

```
#define WRITE(s, v) \
```

```
int tmp = v; \
store_buffer[_pid]!s, tmp; \
copy[_pid * VARSIZE + (s)] = tmp; \
counter[_pid * VARSIZE + (s)]++;
```

\_pid: プロセスID

# READの実装(TSO)

- 三項演算子

(条件式 -> 真の処理 : 偽の処理)

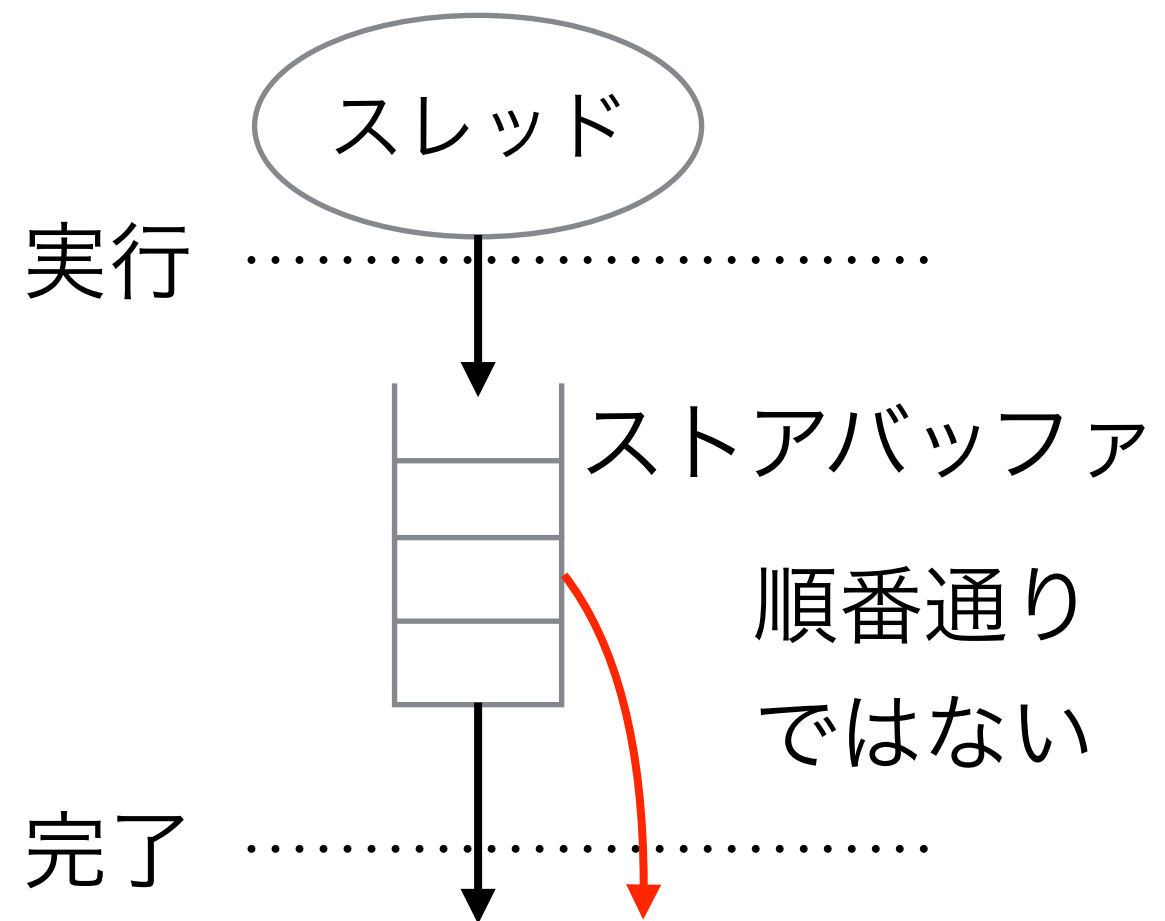
```
#define READ(s) \ 対応するカウンタの値が0?
```

```
(counter[_pid * VARSIZE + (s)] == 0 -> \  
    shared_memory[s] : \  
    copy[_pid * VARSIZE + (s)])
```

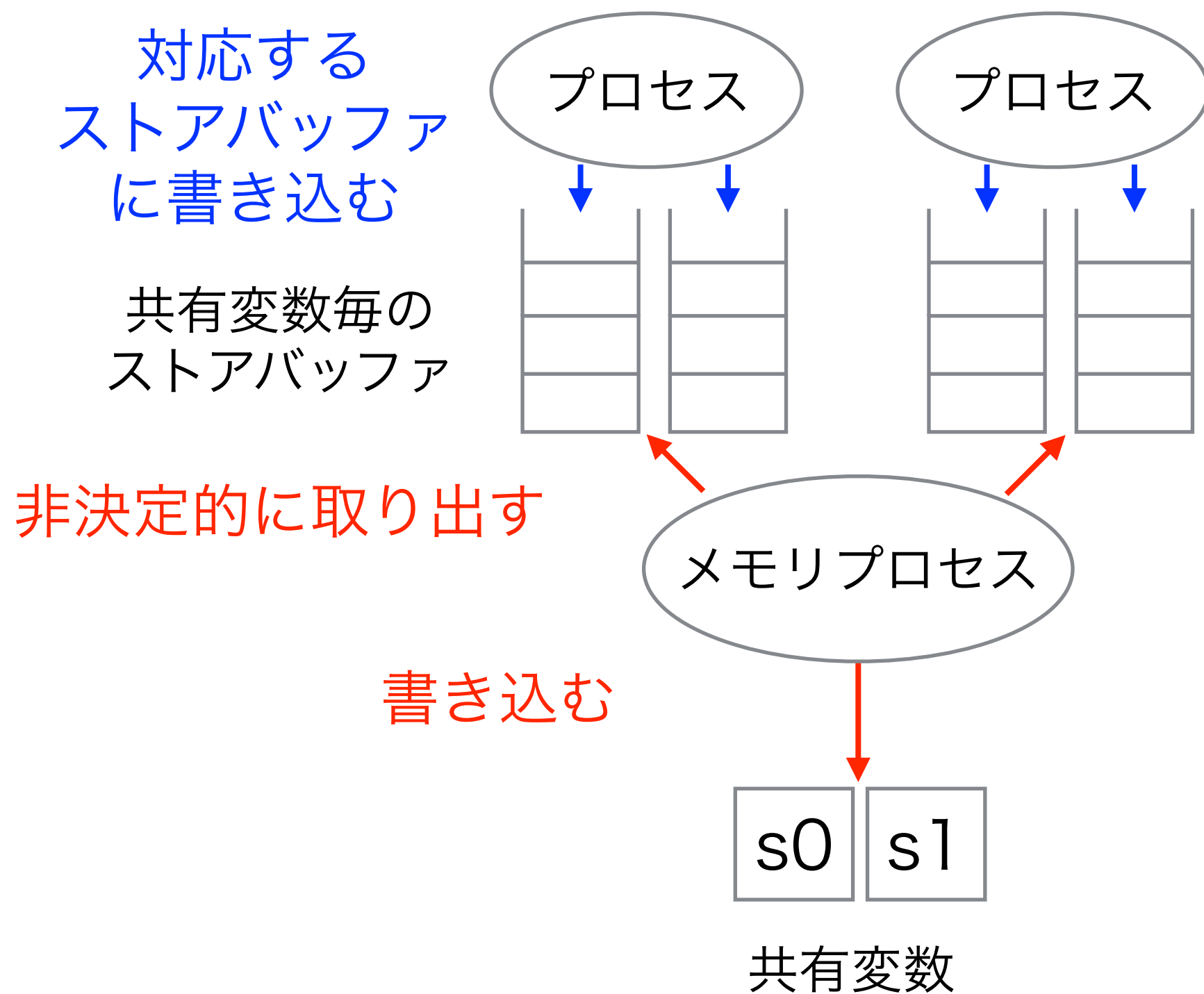
\_pid : プロセスID

# CPUによる高速化：PSO

- WRITE→READ、  
WRITE→WRITEが入れ替わる
- ストアバッファの先頭以外から取り出して完了させることがある
- 同じ共有変数への書き込みは入れ替わらない



# PSOのプロメラ



# 検査したい性質の入力方法

- プロメラ中にassert文を記述する
  - 記述した位置で論理式が満たされるか検査する
- プロメラとは別にLTL式を入力する
  - 実行トレースに関する性質を検査する

## LTL式

- 実行パスの性質を記述できる



# assert文の記述方法

s0に1を書き込んだ後はs0の値は1


- ライブラリを使用しない場合

```
assert (s0 == 1);
```

- ライブラリを使用する場合

```
assert (READ(s0) == 1);
```

スレッドA



```
s0 = 1;  
a = s1;
```

スレッドB

```
s1 = 1;  
b = s0;
```

# LTl式の記述方法

$s0, s1$ のどちらかともいつかは1になる

- ライブラリを使用しない場合

$\langle \rangle (s0 == 1 \ \&\& \ s1 == 1)$

- ライブラリを使用する場合

$\langle \rangle (\underline{SVAR(B, s0)} == 1 \ \&\& \ \underline{SVAR(A, s1)} == 1)$

スレッドBから観測できる  $s0$   
(プロセス)

スレッドAから観測できる  $s1$   
(プロセス)

スレッドA

```
s0 = 1;  
a = s1;
```

スレッドB

```
s1 = 1;  
b = s0;
```

# SVARの実装

- READとの違い
  - 実行したプロセスのプロセスIDに依存しない

`p:_pid` : プロセス`p`のプロセスID

```
#define SVAR(p, s) \
```

```
(counter[p:_pid * VARSIZE + (s)] == 0 -> \
    shared_memory[s] : \
    copy[p:_pid * VARSIZE + (s)])
```

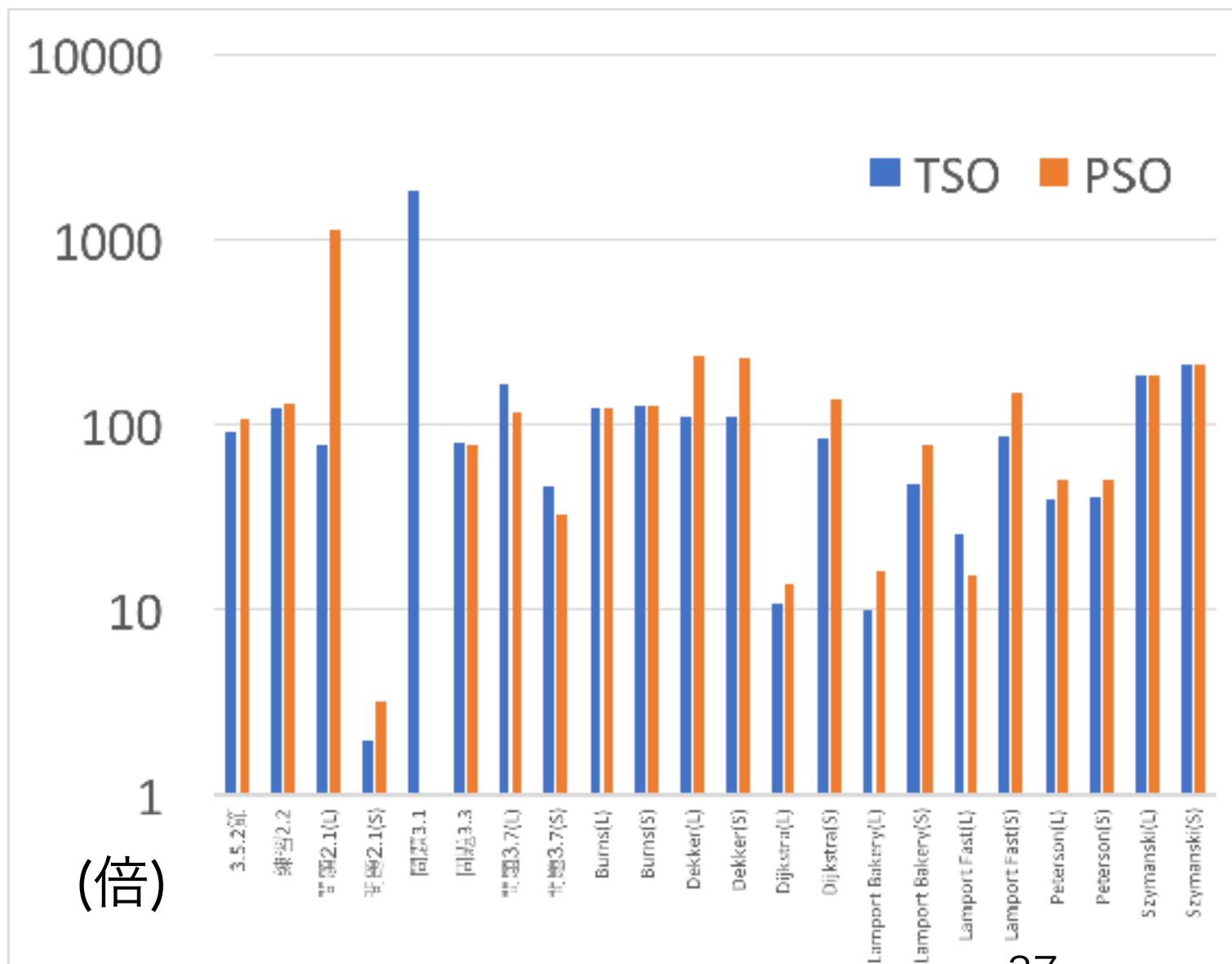
`_pid` : プロセスID

# 性能評価

- 使用したプロメラ
  - モデル検査の教科書のLTL式を使用するプロメラ[産業技術総合研究所システム検証研究センター, '10]
  - メモリモデルを考慮した2プロセスについての相互排除アルゴリズムのプロメラ[Linden, '13]
- 評価の観点
  - メモリ使用量
  - 実行時間
  - 反例を検出するか、検出せずに検査を完了するまで

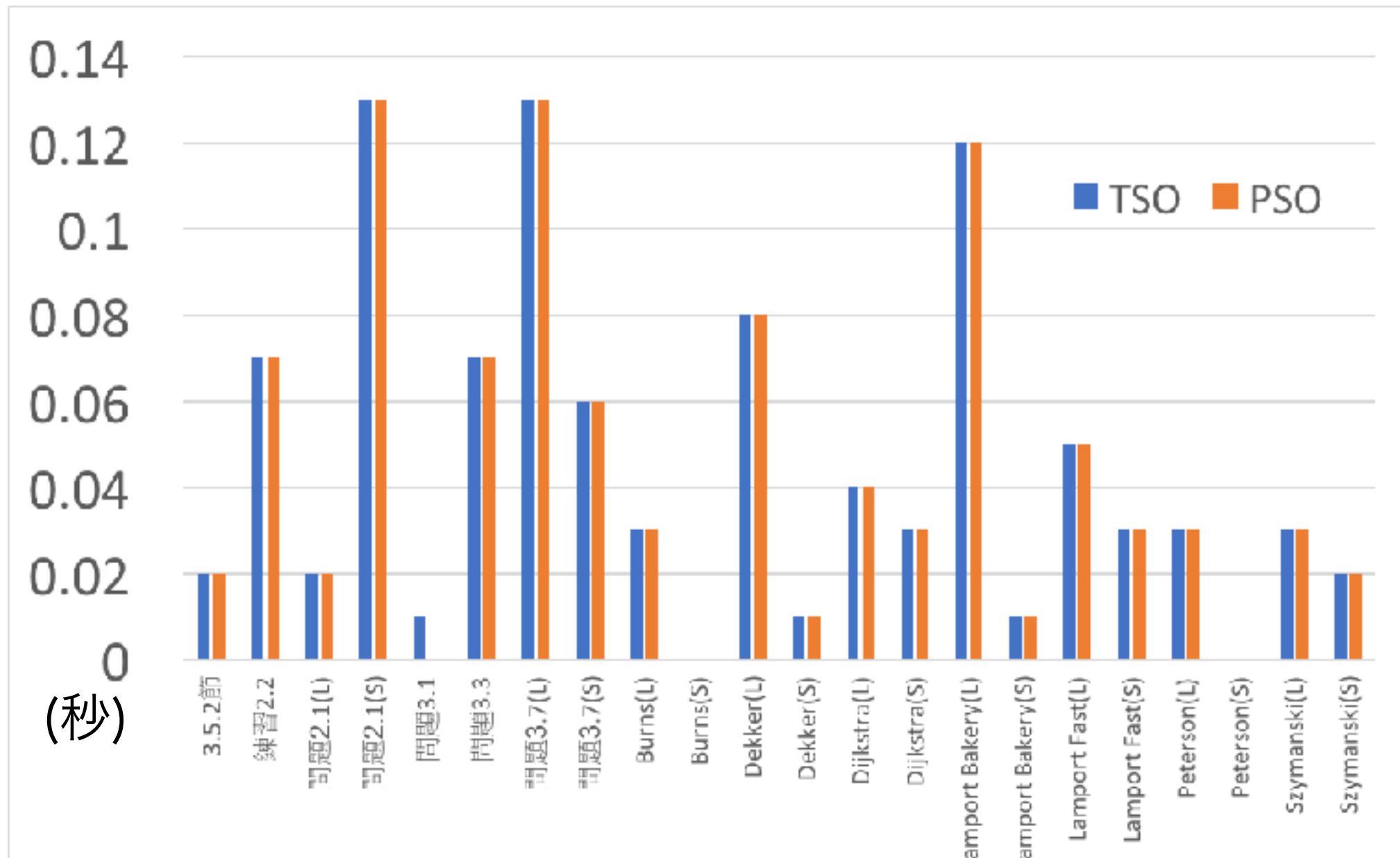
# 性能評価：メモリ使用量

各ライブラリを使用したプロメラ/ライブラリを使用しないプロメラ



- 状態数が爆発して検査しきれなかったプロメラ
- 共有変数が多いと状態数が爆発する特徴を有する
- 極端に共有変数が多い例以外は検査できた

# 性能評価：実行時間



- メモリが足りれば実行時間が問題になることはない

# 関連研究

- C/C++で記述したモデルをメモリモデルに従った実行を検査できるPromelaモデルに変換する手法[Wehrheim et al., '15, '16]
  - 変換後の可読性の低いPromelaモデルを読む必要がある
- メモリモデルに従った実行を検査できるPromelaライブラリ[Wehrheim et al., '15, '16]
  - 共有変数を参照する式を記述できない
- メモリモデルに従った動作をするメモリアクセスAPIを使って、C言語風のモデルを書く[Tomasco et al., '16]
  - LTL式による検査をサポートしていない

# まとめ

- マルチスレッドプログラムの検査ではメモリモデルに従った実行も検査する必要がある
- メモリモデルに従った実行を検査するためのSPIN用ライブラリを開発した
- ライブラリを使用することで、少ない手間で可読性の高いPromelaモデルによる検査を行える