並列分散コンピューティング
(5)並行プログラミング
Cスレッド(3)

大瀧保広

今日の内容

- 排他制御のバリエーション

- Reader-Writer問題
- 同期
- 生産者/消費者 問題

- ロック関連の雑多な話題

排他制御のバリエーション

排他制御のバリエーション

- 2プロセスのロックの例だけをみていると
  排他制御の問題は簡単に見えるかもしれないが、
  実は かなり難しい。

- 排他制御には様々なバリエーションがある。
  例えば：
  - 許可できる同時アクセスが1より大きいケース
  - 書き込みは排他制御しなければならないが、
    読み出しは同時に行っても良いケース
  - ある条件が成立したときにのみアクセスを許可するケース

- 以下ではこれらをモデル化した問題ををみていく。

Reader/Writer問題

- 複数のスレッドが共有するデータがある。
- スレッドは2種類
  - Reader – データを読むだけ。書き込みは行わない。
  - Writer – データの更新 (読み+書き)を行うことができる。
- 条件
  - 複数のReaderは同時にデータを読み出すことができる。
  - 同時にデータにアクセスできるWriterは1つだけ。
  - Writerがアクセスしている時にはReaderはアクセス不可。

	Reader	Writer
Reader	OK	NG
Writer	NG	NG

Reader/Writer問題（単純ロックでやってみる）

■Writer

```
while (true) {
        pthread_mutex_lock(w);
         // データの更新
        pthread_mutex_unlock(w);
}
```

■Reader

```
while (true) {
                pthread_mutex_lock(w);
                // データの読み出し
                pthread_mutex_unlock(w);
}
```

	Reader	Writer
Reader	NG	NG
Writer	NG	NG

すべてのスレッドが排他的に
アクセスすることになる。
Reader同士も排他的。
効率が悪い。

Reader/Writer問題（ロック+counterにしてみる）

■Writer

```
while (true) {
        pthread_mutex_lock(w);
         // データの更新
        pthread_mutex_unlock(w);
}
```

■Reader

```
while (true) {
            readcount++;
            if（readcount==1) pthread_mutex_lock(w);
            //データの読み出し
            readcount--;
            if (readcount==0) pthread_mutex_unlock(w);
}
```

最初にアクセスを始める
Reader がロック。
最後の Reader が アンロック。

サンプルプログラム（ReaderWriter）

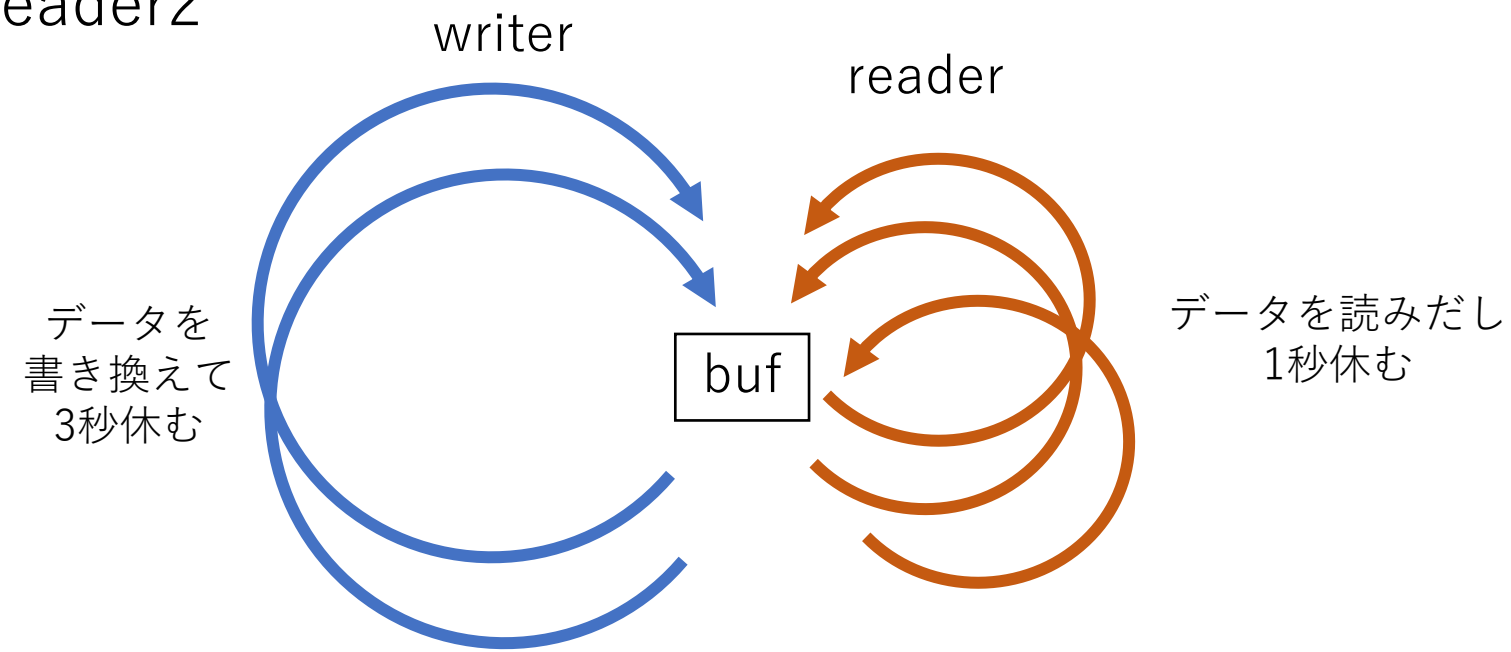
- **PDC/C-Thread2/ReaderWriter/**

- プログラムの説明
  - reader1.c
    writer 2つ ：3秒ごとにデータを書き換える
    reader 4つ ：読み出したデータを表示する
    ロックのみ

  - reader2.c
    上に同じ：ロック ＋ readcount

実行してみよう

- コンパイルと実行
  - make
  - ./reader1
  - ./reader2



サンプルプログラム（ReaderWriter）

reader2.cでは以下のようなことが起きる。

■readerがアクセスしているのにreadcount=0
→ロック外れ
→writerが書き込みに来てしまう

■readerがひとつもアクセスしていないのに readcount>0
→ロックかかったまま
→writerがいつまでも書き込めない

サンプルプログラム（ReaderWriter）

■原因
変数readcount がreader同士の共有資源なのに、
排他制御を行なっていない。

■readcountへのアクセスを排他制御するための ロックを
新たに追加し、プログラムを修正せよ。
振る舞いは どのように変化するか？

セマフォ（Semaphore）

- セマフォ：
  排他制御をより一般化したアクセス制御のための機構

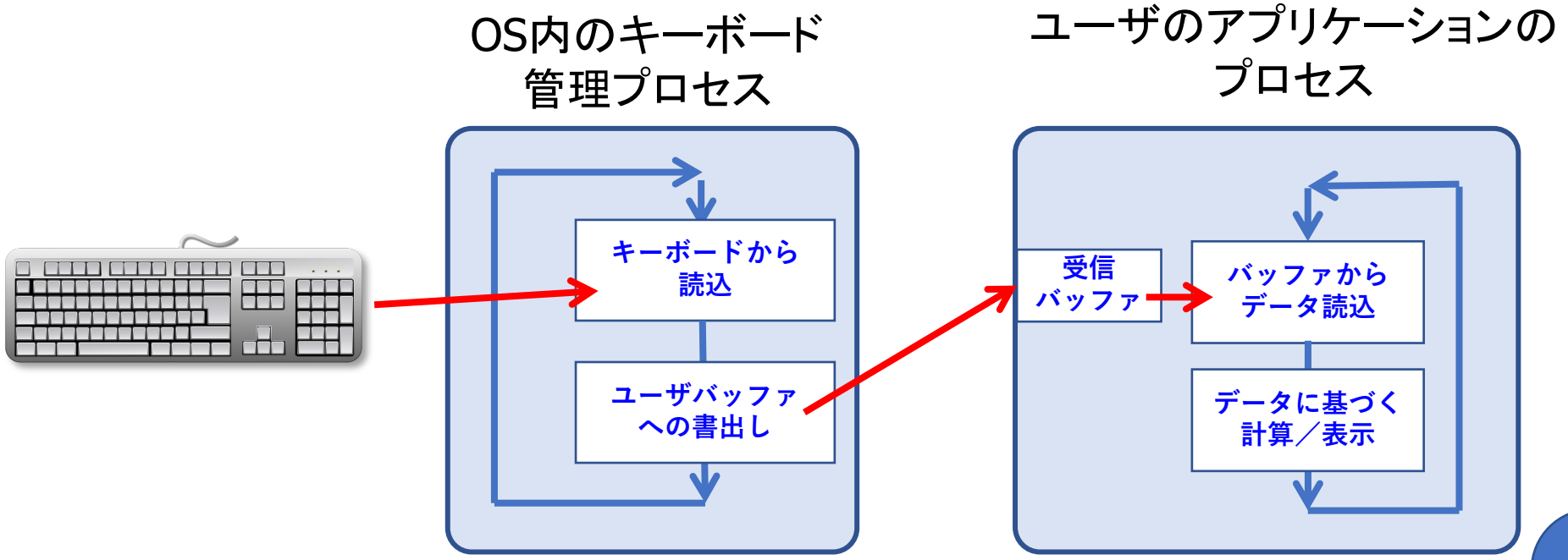
- カウンティング セマフォ
- バイナリ セマフォ(0,1のみ)
  ＝ロック（mutex）

- カウンティングセマフォは、自分で頑張らなくても
  pthread_mutexの属性を変更することで実現できる。
  （前回資料を参照）

同期

並行処理で発生する問題

- 通信によって協調動作をする場合でも問題は発生する。
- 共有資源となるのは通信データを授受するところ



並行処理で発生する問題

キーボード
管理プロセス

バッファ

アプリのプロセス

キーボードから読込

ユーザバッファへの書出し

キーボードから読込

ユーザバッファへの書出し

:

バッファからデータ読込

計算/表示

バッファからデータ読込

計算/表示

:

バッファへ書き出す前に
キーボードから
次のデータを読むと
前のデータが消える

**ここの順序が
守られることが必要**

順序が守られないと：
- データが読み出される前に上書き。
- 同じデータを2回 読み込む。

やってみよう

前のReader-Writer問題では、
Writer が書き込んだデータを Readerが読み出している。

この「データの受け渡しの問題」は、
ReaderWriter問題とは違うのだろうか？

さっきのreader/writer問題のプログラム reader1.c を
reader1個、writer1個にして実行してみよう。

期待している動きと、どこが違うだろうか？

並行処理で発生する問題

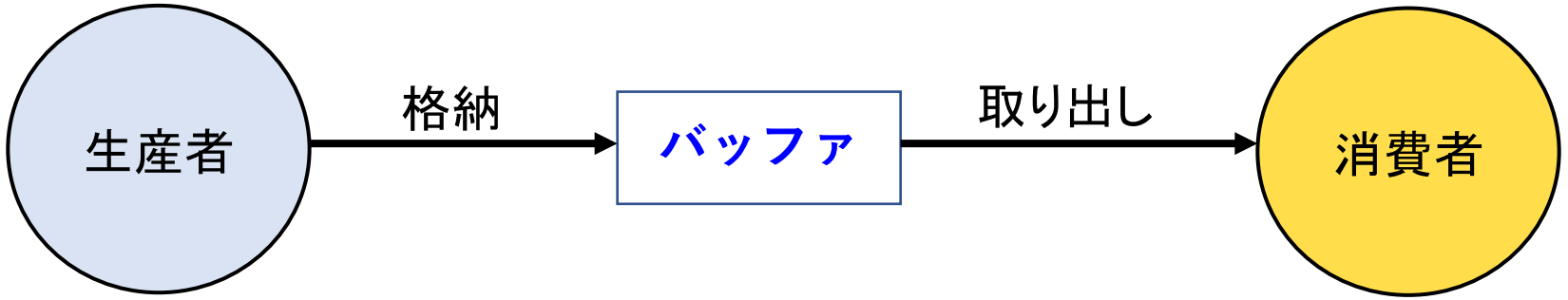
- 異なるプロセス（スレッド）上の処理の間の順序関係が正しくないと、望む結果にならない。

- ところが
  異なるプロセス上での実行の状況（位置）は
  お互いに分からない。
  これではタイミングを合わせることができない。

- そこで、
  アクセスのタイミングを制御する仕掛け
  （「同期」を取るための仕掛け）
  が必要になる。

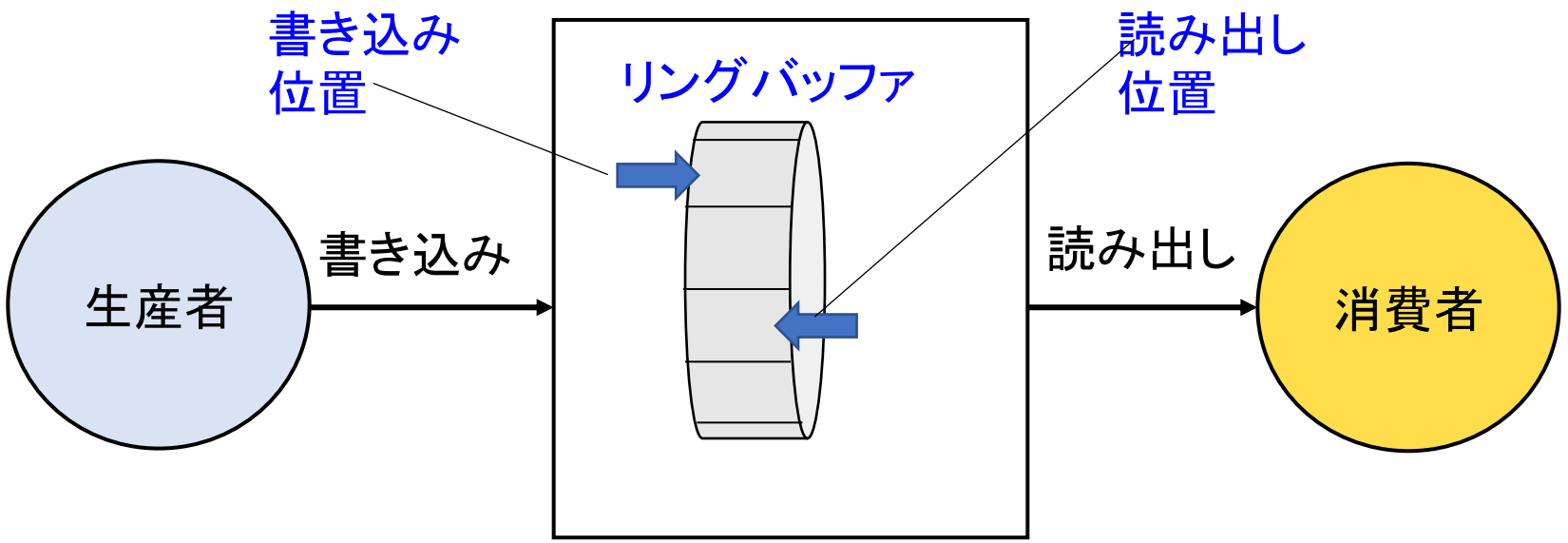
生産者／消費者 問題（データの受け渡し）

- 登場人物
  - サイズが有限の１個のバッファ
  - ２種類のスレッド
    - 一方のスレッドを生産者、もう一方のスレッドを消費者と呼ぶ。
- 動作
  - 生産者はバッファにデータを入れる。
  - 消費者はバッファからデータを取り出す （そして処理する）。
- 制約条件
  - 生産者は、バッファに空きができるまでデータを入れることができない。
  - 消費者は、バッファが空だとデータを取り出すことができない。



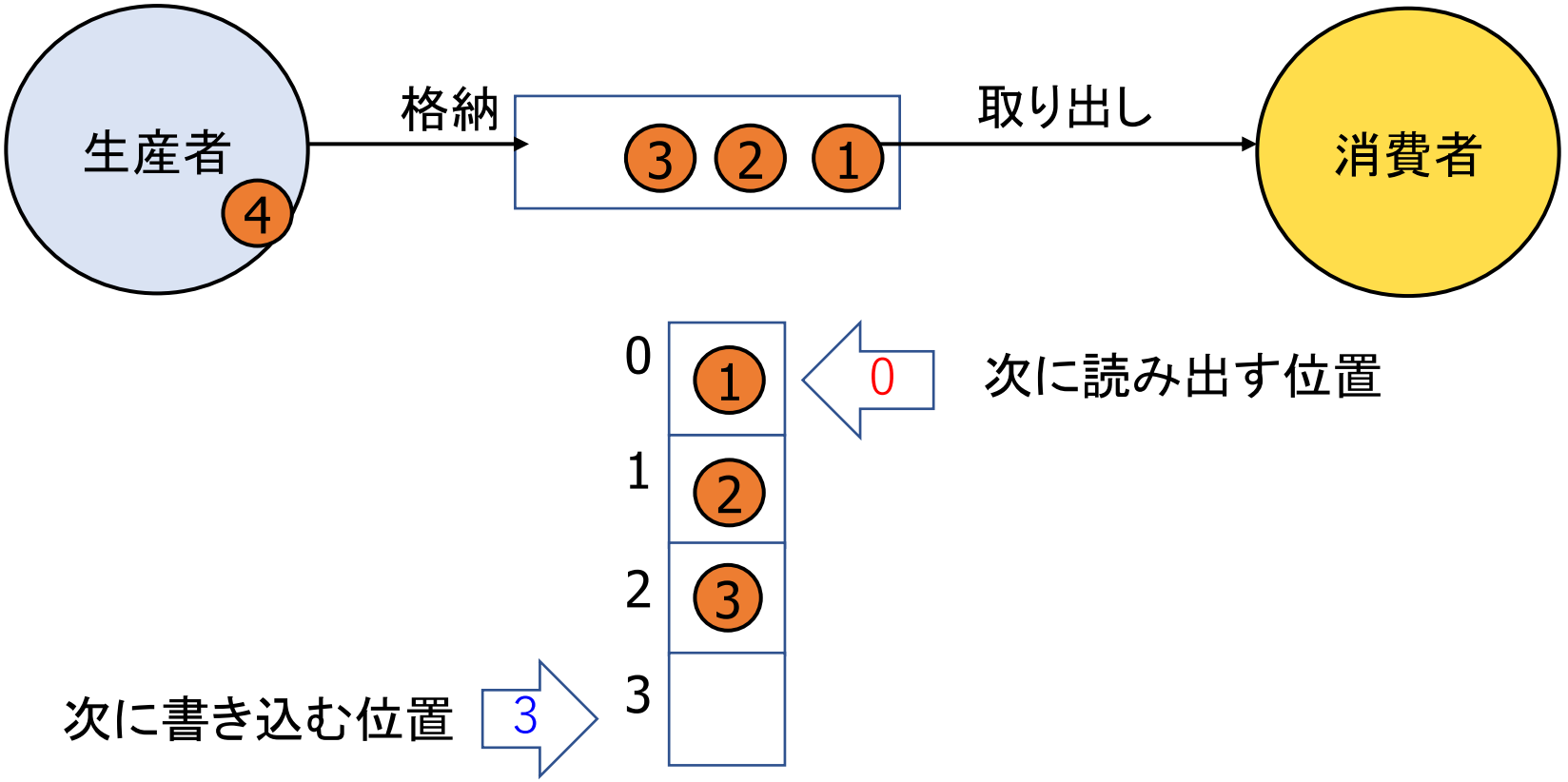
生産者／消費者 問題

- バッファのサイズは1ではなく、複数（有限）入るようにする。
- 実装方法はいろいろあるが、
  ここではキュー（リングバッファ）として実装する。



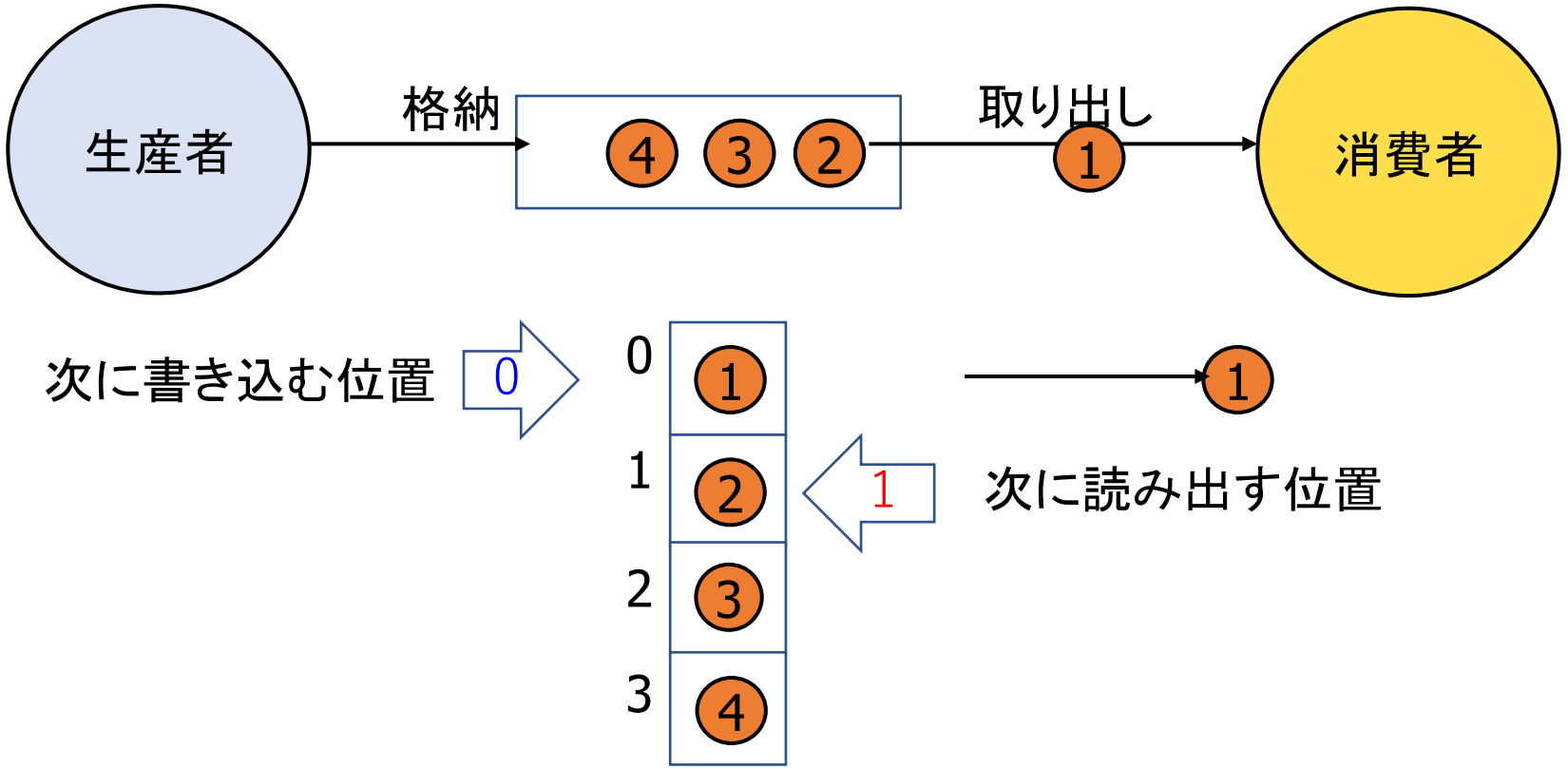
バッファの実装

- 動作
  - 生産者はバッファにデータを入れる。
  - 消費者がバッファからデータを取り出す。



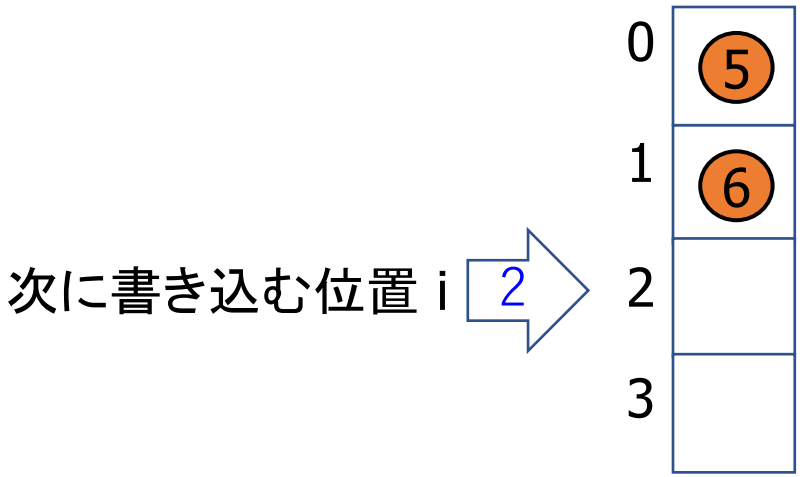
バッファの実装

動作
- 生産者はバッファにデータを入れる。
- 消費者がバッファからデータを取り出す。



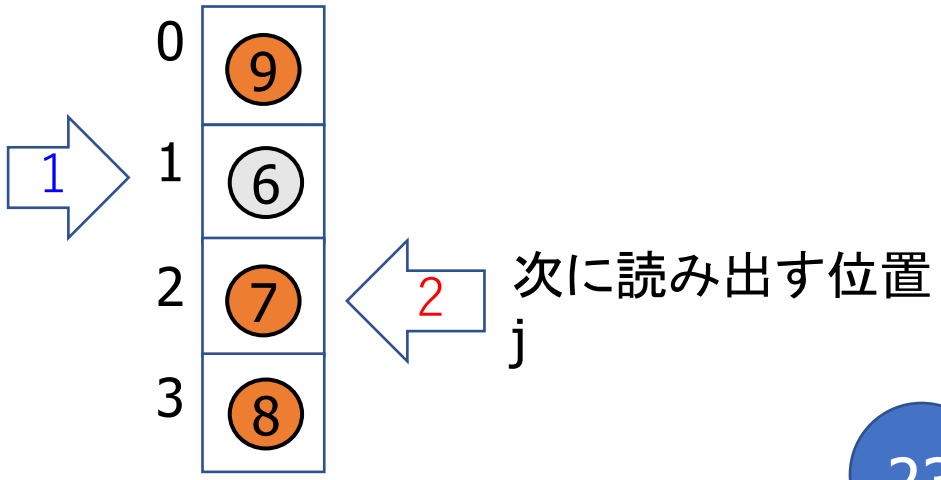
生産者(Producer) のイメージ

```
while (Items_number == buffer size)
        ;   //満杯なので、空くのを待つ

Buffer[i]=next_produced_item;
i=(i+1)%Buffer_size;
Items_number++;
```



消費者(Consumer)のイメージ

```
while (Items_number == 0)
        ; // 空なのでデータが入るのを待つ

Consumed_item= buffer[j];   ⑦
j = (j + 1) % Buffer_size;
Items_number--;
```



生産者/消費者 問題

- Items_number++ は実際には以下のように実行される。
  1. register1=Items_number;
  2. register1=register1 +1;
  3. Items_number=register1;

- Items_number-- も同様。

- Items_numberは生産者スレッドと消費者スレッドの共有資源であるため、排他制御する必要がある。

サンプルプログラム(consumer)

- **PDC/C-Thread2/consumer/**
  - 生産者1個、消費者1個、バッファサイズ4の生産者/消費者問題。
- プログラムの説明
  - consumer1.c：
    バッファに対するmutexだけ。
  - consumer2.c：
    バッファの満杯／空を判定する処理を加えただけ。

サンプルプログラム(consumer1.c)

■生産者スレッド

```
void *producer( void *x ) /* 生産者 */
{
    struct buffer *b;
    int i, v;
    b=(struct buffer *)x;
    for( i = 0 ; i<10 ; i++ ) {
        v = i ;
        put( b, v );   /* バッファ b に v を書き込む */
    }
    pthread_exit(NULL);
}
```

サンプルプログラム(consumer1.c)

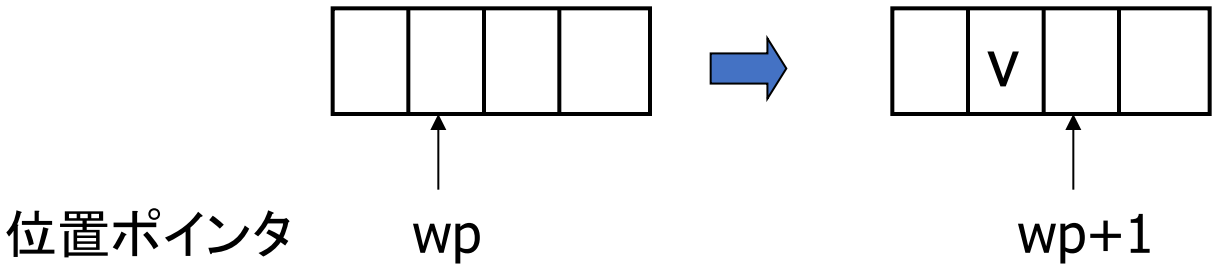
■消費者スレッド

```
void *consumer( void *x )  /* 消費者 */
{
   struct buffer *b;
   b=(struct buffer *)x;
   for( i = 0 ; i<10 ; i++ ) {
      v=get( b );   /* バッファ b から読み出す */
    }
    pthread_exit(NULL);
}
```

サンプルプログラム(consumer1.c)

■バッファにデータを書き込む操作

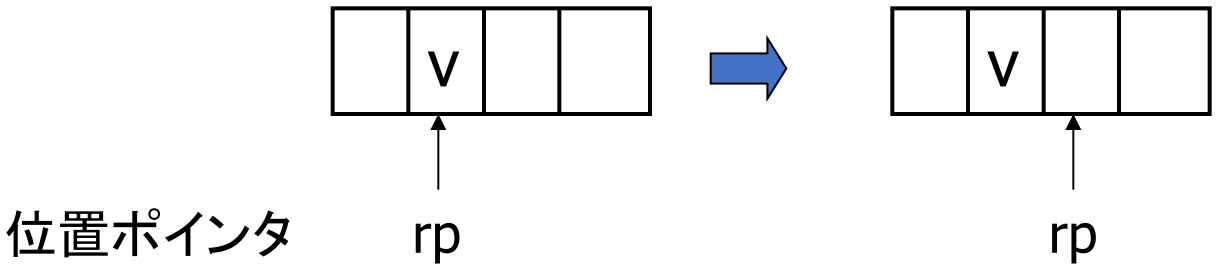
```
void put( struct buffer *b, int v )
{
  pthread_mutex_lock( &b->mutex );
  b->data[ b->wp ] = v ;
  b->wp = (b->wp +1) % BUFFER_SIZE; //書き込み位置を進める
  pthread_mutex_unlock( &b->mutex );
}
```



サンプルプログラム(consumer1.c)

■バッファから値を読み出す操作

```
int get( struct circular_buffer *b )
{
   pthread_mutex_lock( &b->mutex );
  位置rpの配列dataの値を読み出して、rpを1進める;
  pthread_mutex_unlock( &b->mutex );
  return( v );
}
```



実行してみよう

- make
- ./consumer1

- bufferに対する排他制御だけでは
  readerが読み出す前に
  writerが上書きするのを防止できていない。

consumer2.c

- バッファに格納されているitem数を把握して、
  書き込みが読み出しをoverrunしないようにしよう！
  - リングバッファに現在格納されているitem数を管理する。
  - 関数putはitem数とBuffer_sizeから
    空きがあるかどうかを判断する。
  - 関数getはitem数からデータがあるかどうか判断する。

サンプルプログラム(consumer2.c)

■バッファにデータを書き込む操作

```
void put( struct buffer *b, int v )
{
    pthread_mutex_lock( &b->mutex );
    バッファが満杯なら待つ;
    b->data[ b->wp ] = v ;
    b->wp = (b->wp +1) % BUFFER_SIZE; //書き込み位置を進める
     pthread_mutex_unlock( &b->mutex );
 }
```

サンプルプログラム(consumer2.c)

■バッファから値を読み出す操作

```
int get( struct circular_buffer *b )
{
   pthread_mutex_lock( &b->mutex );
   バッファが空ならば待つ;
   位置rpの配列dataの値vを読み出して、rpを1進める;
   pthread_mutex_unlock( &b->mutex );
   return( v );
}
```

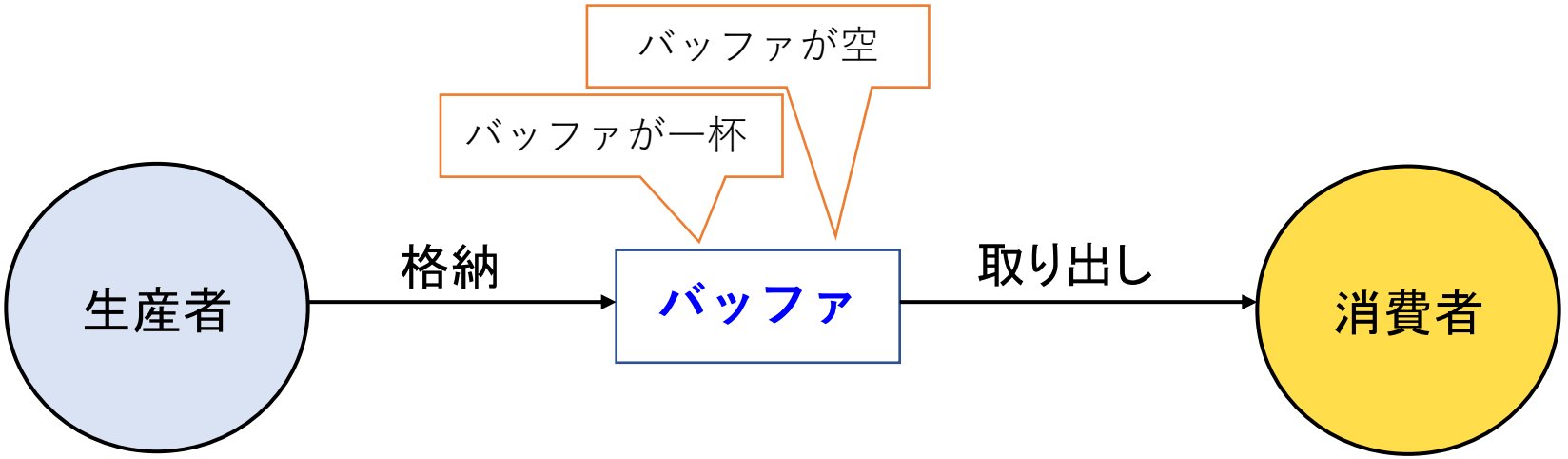
実行してみよう

- ./consumer2

- 望み通りの動きだったか？
- 何が起きたのか わかりますか？

生産者／消費者 問題

- 単純な排他制御ではない→ロックでは解決できない。
- **条件変数**を用いて制御する。
  - ある条件が成立しているかどうかを示す変数。
    - バッファが一杯の状態か？
    - バッファが空の状態か？



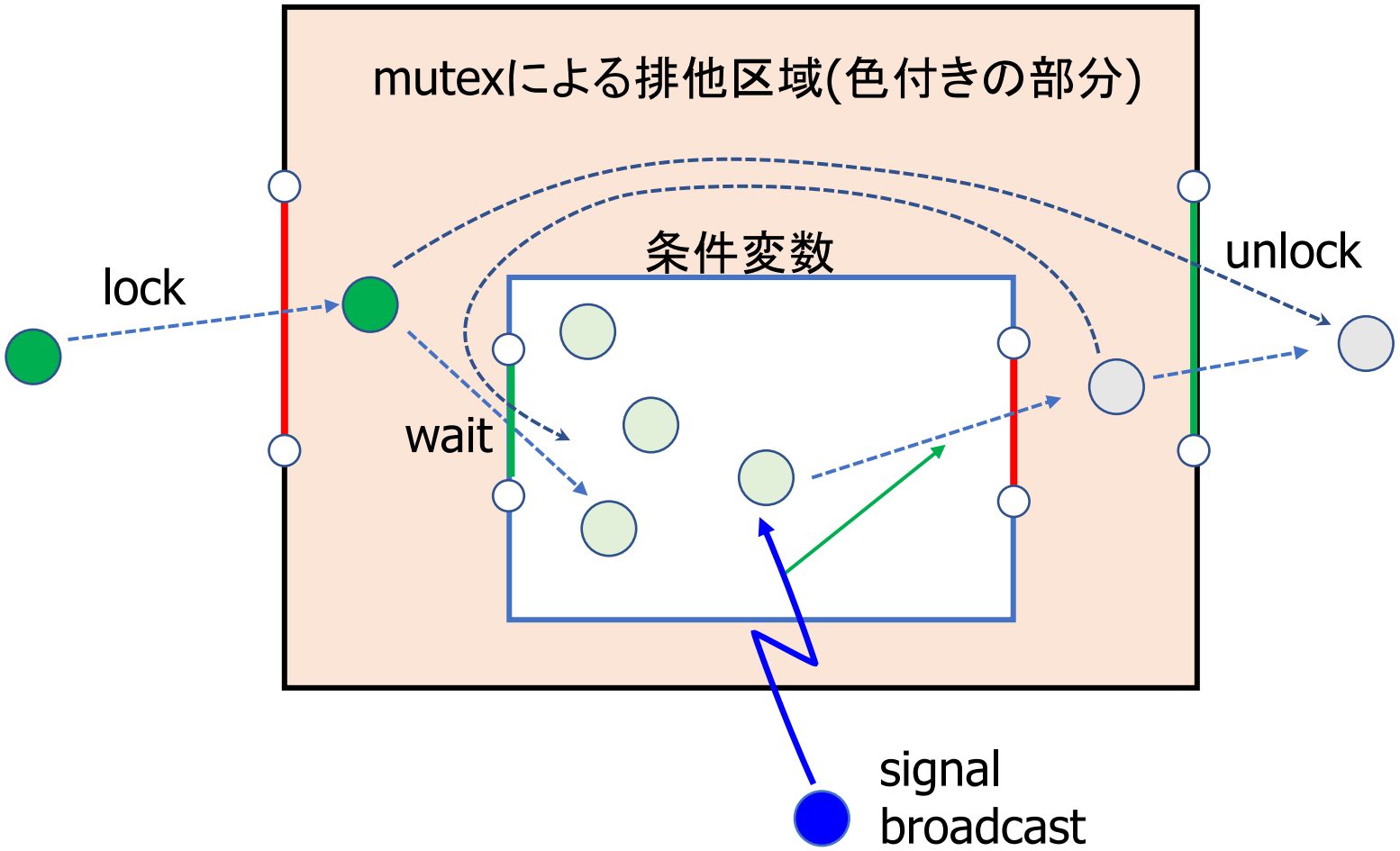
■**条件変数**の宣言

```
pthread_cond_t  cv;
```

■条件変数の操作関数

操作	関数
条件変数 cv の初期化	pthread_cond_init(&cv, NULL)
条件変数によるブロック	pthread_cond_wait(&cv, &mutex)
ひとつのスレッドのブロック解除	pthread_cond_signal(&cv)
時刻指定のブロック	pthread_cond_timewait(&cv)
全スレッドのブロック解除	pthread_cond_broadcast(&cv)
条件変数の削除	pthread_cond_destroy(&cv)

mutex（ロック）と条件変数の関係



サンプルプログラム(consumer3)

■**consumer/consumer3.c：**

バッファの満杯／空を判定する処理を条件変数で制御

■**条件変数**

- ■**not_full**　　**：満杯ではない　という条件**
- ■**not_empty**　　**：空ではない　という条件**

サンプルプログラム(consumer3.c)

■バッファにデータを書き込む操作

```
void put( struct buffer *b, int v )
{
    pthread_mutex_lock( &b->mutex );
    バッファが満杯なら待つ(条件変数を指定して待機状態へ);
    b->data[ b->wp ] = v ;
    b->wp = (b->wp +1) % BUFFER_SIZE; //書き込み位置を進める
    読み出すデータがある状態になったことをsignalで通知する。
     pthread_mutex_unlock( &b->mutex );
 }
```

サンプルプログラム(consumer3.c)

■バッファから値を読み出す操作

```
int get( struct circular_buffer *b )
{
   pthread_mutex_lock( &b->mutex );
    バッファが空ならば待つ(条件変数を指定して待機状態へ);
   位置rpの配列dataの値vを読み出して、rpを1進める;
   バッファに空きができたことをsignalで通知する;
   pthread_mutex_unlock( &b->mutex );
   return( v );
}
```

サンプルプログラム(consumer3.c)

- コンパイル／実行してみよ。
  バッファへの書き込み／読み出しは、納得のいく動きか？

- consumer3.c 内のprintf();の行のコメントを外し、
  より詳細な動きを表示するように修正してみよ。
  動きは納得できるか？

排他制御は難しい！

Lock considered Harmful.（「ロックは害悪である」）

- ロックを用いたすべてのプログラムは、バグがあるか、もしくはまだバグが見つかっていないかのどちらかである -- “Java Concurrency in Practice”
- ロックによる並行プログラミングは、アセンブリでプログラムするに等しい -- “出典不明”

ロックによる排他制御の問題点

- ロックが もたらすバグ・不具合一覧
  - アンロックし忘れ
  - デッドロック
  - ロックの不足（Race Condition）
  - 過剰なロック（全然並列にならないぞ）
  - モジュラリティの欠如

ロックによる排他制御の問題点

- **ロックし忘れ**
  - ロックしたmutexをアンロックし忘れる
  - リソースに永久にアクセスできなくなる
  - 対策：手動でのロック・アンロックは“決して行なってはいけない”（えっ!）

    →**ライブラリやミドルウェアに任せろ、素人は手を出すな。**

- **デッドロック**
  - 複数のロックを複数のスレッドが同時に獲得しようとする際に発生
  - スレッドが永久に停止する
  - 対策：必ず同じ順番でロックを取るようにする
    - すべてのロックに順序を付ける。
      これから獲得しようとするロックを予め全て列挙し、
      みんなが その順番でロックすればよい。→**実際にできるか？**

ロックによる排他制御の問題点

- **ロックの不足**
  - クリティカルセクションでロックを忘れる
  - 対策：忘れないように気をつける。
- **過剰なロック**（並行に動作しなくなる）
  - 心配性のため必要のないところでロックをかけてしまう
  - 粒度の大きなロックをかけてしまう
  - 対策：本質的に困難。
    粒度の小さいロックを実現する方法をみつけること自体が研究テーマになるレベル。
- **モジュラリティの欠如**
  - ロックを用いたプログラムモジュールは互いに組み合わせることができない
  - 対策：原理的に不可能

ロックを使わない並行プログラミング技術

- **Message Passing**
  - スレッドがそれぞれメッセージキューを持ち、スレッド間の通信はメッセージのやり取りに限る
  - プログラミングモデルに制限がかかる
- **Software Transactional Memory（STM）**
  - メモリ操作をトランザクションとして記述する
  - 実装としては、楽観的並行性制御で実行し、競合がおきたらロールバックするのが一般的
- **Concurrent Revisions**
  - 最近(2010-)提唱された並行性制御のための手法
  - バージョン管理のアナロジーで 共有リソースを扱う
  - スレッドごとに領域が割り当てられる。Joinの際に決定的(deterministic)にコンフリクトを解消する

今日のまとめ

- 排他制御のバリエーション
  - Reader-Writer問題
- 同期の実現
  - 生産者/消費者 問題
- ロックを自分で書こうなんて思ってはダメ