並列分散コンピューティング
(4)並行プログラミング
Cスレッド(2)-ロック

大瀧保広

今日の内容

- 排他制御
- ロック
- Dekkerのアルゴリズム

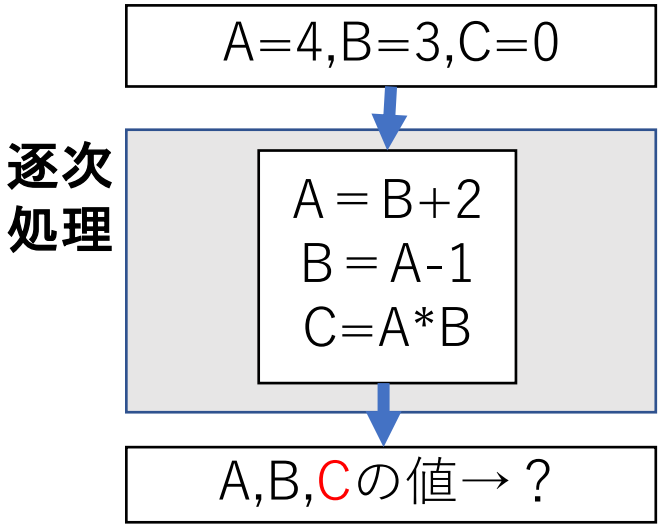
- C言語によるスレッドプログラミング
  - ロック
  - デッドロックの例

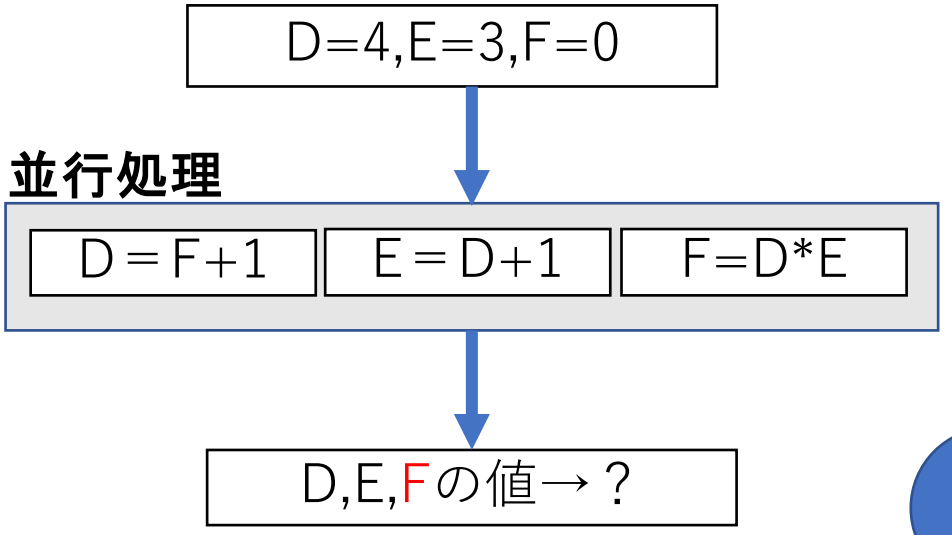
- （中間レポート課題のソースコードの説明）

排他制御

復習：並行処理の決定性

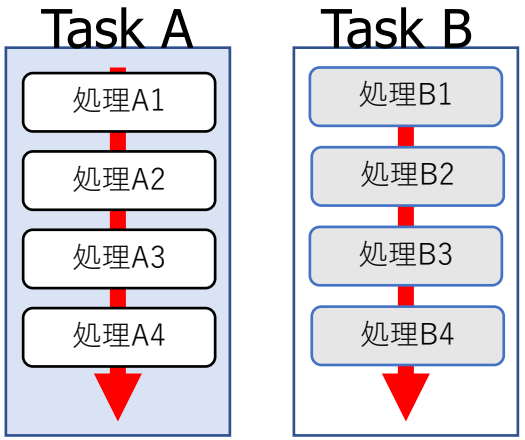
- 処理が「決定的(deterministic)」であるとは：
  同じ入力が与えられたときに、いつも同じ結果になること。
  - 逐次処理は、決定的である。
  - 並行処理では、それぞれの処理が決定的であったとしても、一般に、全体としての結果は一意に定まらない（非決定的）。



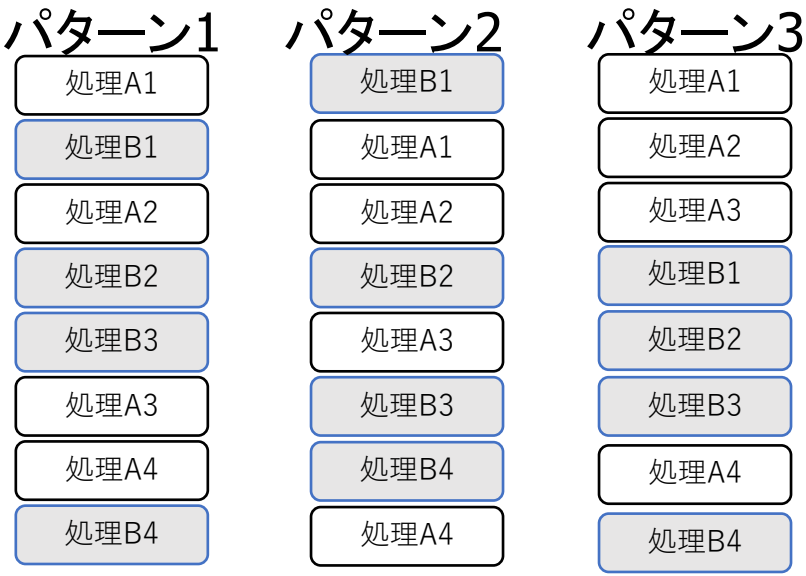


競合状態(Race Condition)

- 複数のタスクが同一の資源（＝共有資源）にアクセスするとき、処理結果が非決定的になる状況が発生する。
  これを競合状態という。(正確には Data Race Condition)
- 競合状態は、真の並列か擬似並列かに かかわらず発生する。



擬似並列の場合の実際の処理の様子



余談：

厳密には非決定性と不整合は別の話である。
真に避けなければならないのは、不整合の方。

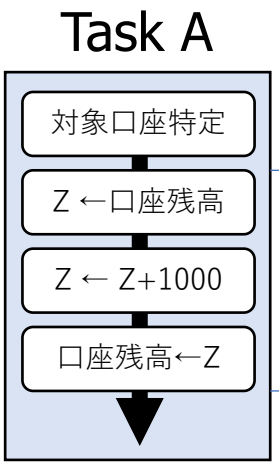
競合状態が発生するプログラムのデバッグはとても難しい。

- デバッグの基本は、再現条件を確定させること。
- 実行するたびに結果が変わるため、バグの再現条件がなかなか確定できない。
- デバッグ用のコード(printf など)を追加することで、競合が発生しない（しにくい）状態になることもある。

→//「なぜかわからないが、この行を削除すると動かなくなる」という謎のコメント

復習：クリティカルセクション

- 正しい処理結果を得るためには、
  口座残高を読み出してから書き戻すまでの間に、
  他のタスクによる口座残高の値の変更が行われてはならない。
- このような
  他のタスクから共有資源をアクセスされると困る
  処理区間
  をクリティカル セクション(Critical Section)と呼ぶ。



クリティカル セクション
この処理の間において、
他のタスクに口座残高への
アクセス(書き込み)を許すと、
不整合が生じる恐れがある。

TaskBにも同様に
クリティカル
セクションが
ある。

クリティカルセクション と 排他制御

■非決定性をなくすには、
　クリティカルセクションにおいて、
　共有資源にアクセスしているタスクが自分だけである
ように制御する仕組みが必要である。

■これを排他制御、相互排他（**Mut**ual **Ex**clusion）などと呼ぶ。

排他制御手法に求められる性質

排他制御を行う方法には 様々な手法がある。
以下の条件を満たすことが求められる。

■**即時性**
クリティカルセクションの実行に競合するプロセスが他にない場合，プロセスはクリティカルセクションの実行を即座に許可されること。

■**デッドロック防止**
競合するプロセスがある場合，許可されるまで永久に待たされてはいけない。

■**公平性**
どのプロセスも，他のプロセスがクリティカルセクションを実行することを妨げられない。

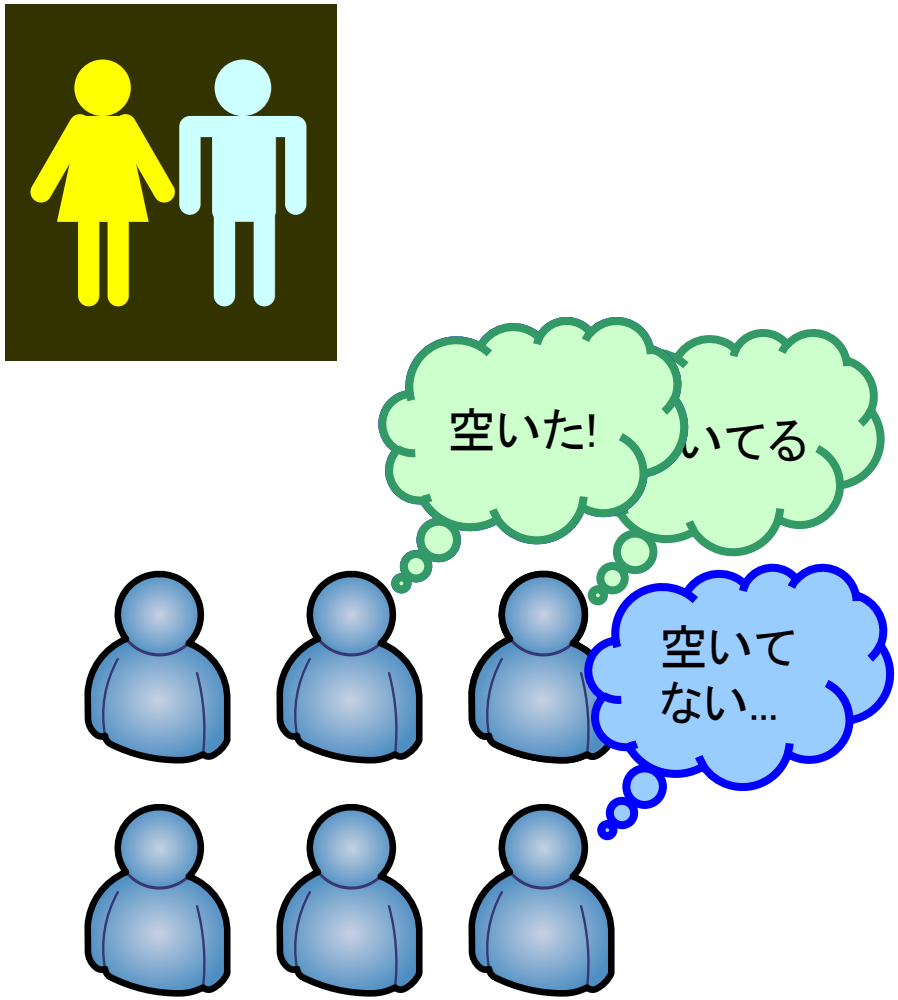
ロックによる排他制御

■排他制御の仕組みの中で
直感的に理解しやすいのはロックである。

■ロックを用いた排他制御では、
共有資源にアクセスしたいタスクは、クリティカルセクションの処理を行うにあたり、（直感的には）以下の手順に従う。

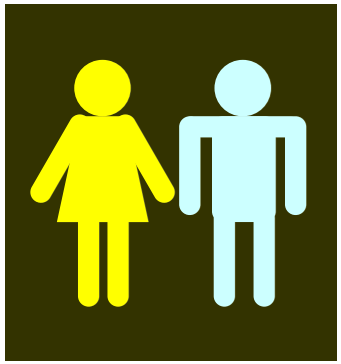
1. 共有資源がロックされているか調べる。
2. ロックされていたら1へ(wait)。
   ロックされていなければ、ロックをかけ、3へ。
3. クリティカルセクションの処理を行う。
   （=共有資源にアクセスする）
4. ロックを解除する。

この方法で大丈夫か？

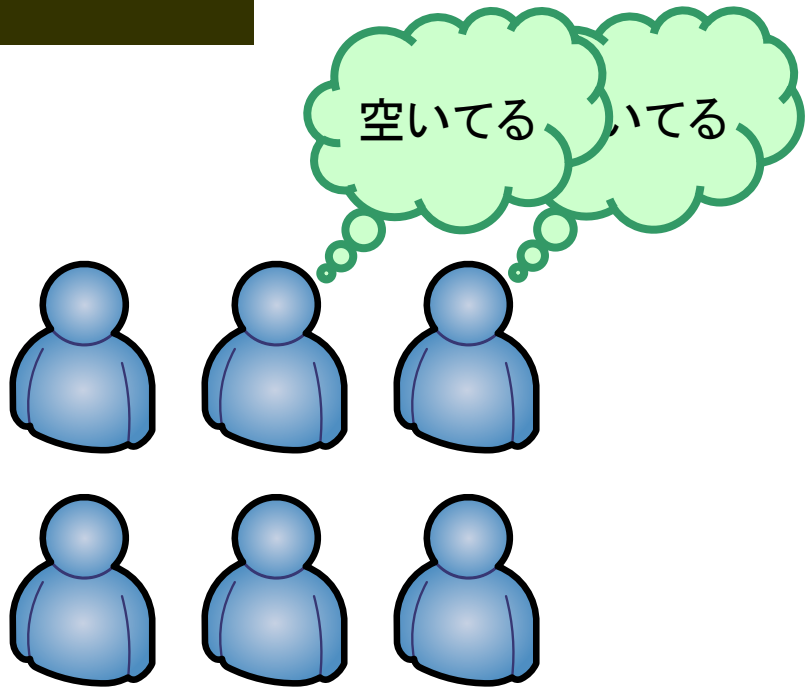


- 新幹線のトイレで考える
  - クリティカルセクション（トイレ）に入ろうとするプロセス（乗客）は，フラグ（インジケータ）を確認し，入るかどうかを決定
  - クリティカルセクションに入ったらフラグを下げる（インジケータが点く）

この方法で大丈夫か？



- うまくいかない場合
  - フラグを見て確認
  - 入る



ハードウェアによるロック

「ロックの状態をチェックする処理」と「ロックをかける処理」の間に他のプロセスが動くことを許すと危ない。

- Test-and-Set (TAS)命令
  メモリ上のある番地の値のチェックと セットを アトミック に行う命令。
  - アトミック（Atomic）とは、
    処理が途中で分割されないことが保証されていることをいう。
  - 以前はCPUのひとつの命令として存在していた。
    シングルプロセッサの時代には、これで排他制御が実現できた。
  - マルチコアCPUの排他制御はより複雑。

Dekkerのアルゴリズム

ソフトウェアによる排他制御の基本形

- 2プロセスの排他制御を行うことが可能。
  それぞれプロセスA,Bとする。

2つのフラグを利用する：

- Interest (true または false)
  - プロセスA,Bが、クリティカルセクションに興味があるか否かを示す(興味があればtrue)
- Priority (A または B)
  - プロセスA,Bがクリティカルセクションに同時に興味を持った場合，どちらを優先するかを決定する

Dekkerのアルゴリズム

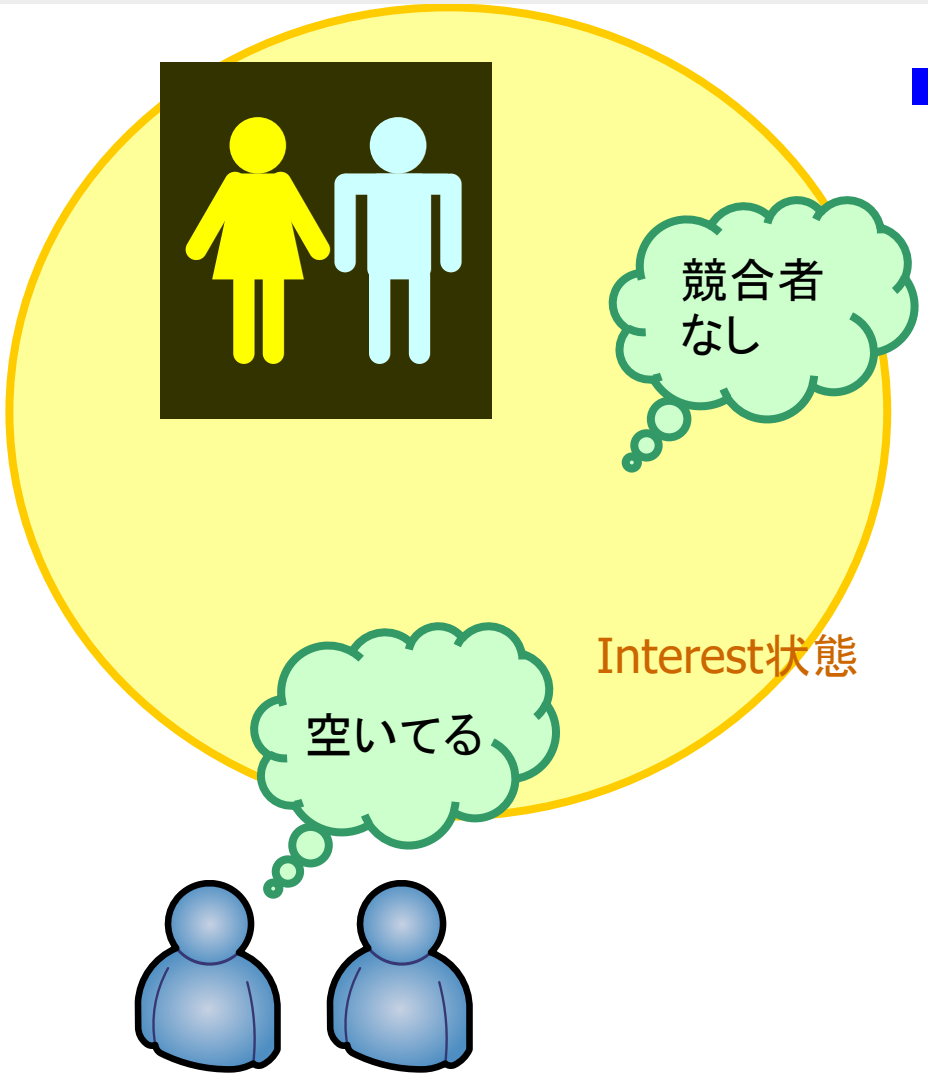
A

```
Interest[A] = TRUE;
while( Interest[B] ){
  if( Priority == B ){
    Interest[A] = FALSE;
    while( Priority == B ){};
    Interest[A] = TRUE;
  }
}

        :
クリティカルセクション
        :

Priority = B;
Interest[A] = FALSE;
```

B

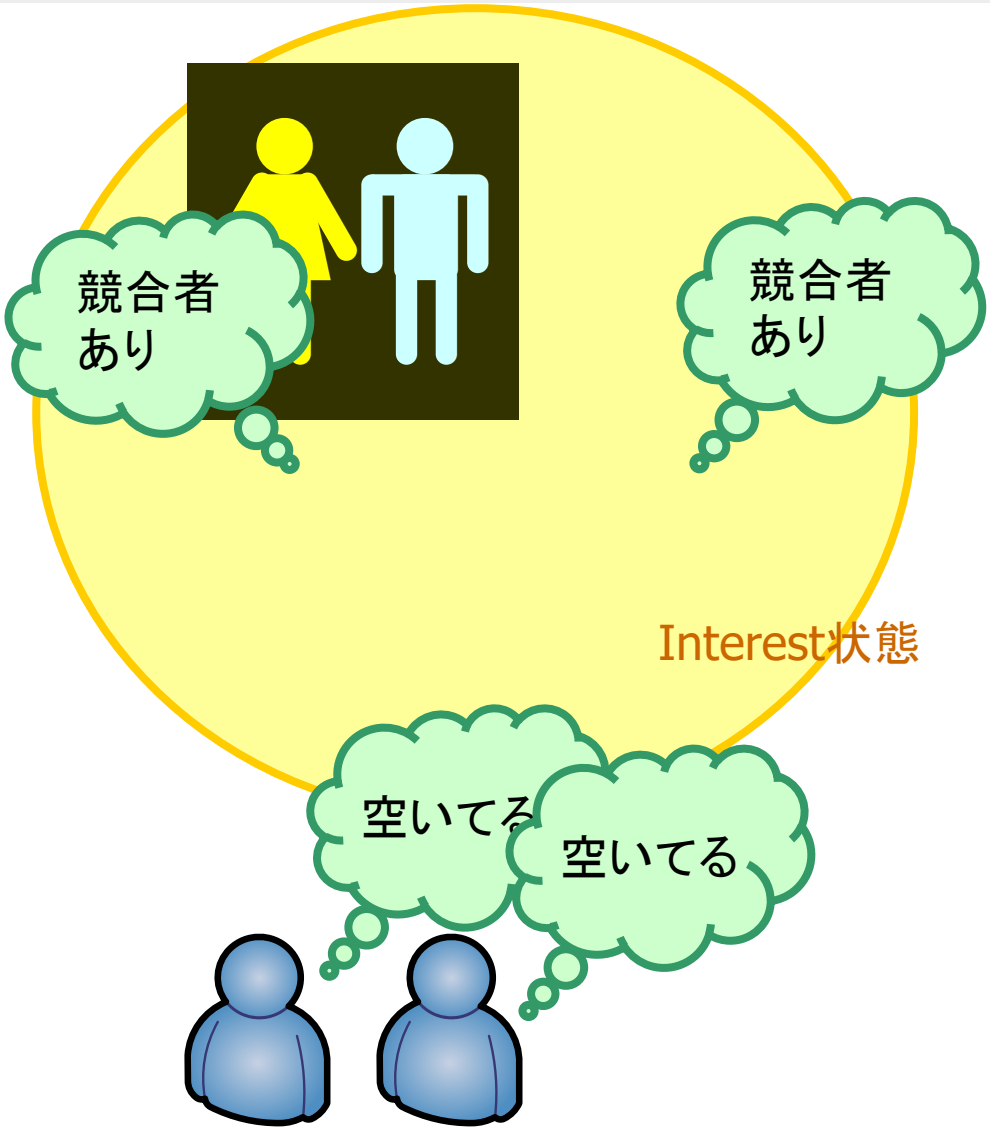
```
Interest[B] = TRUE;
while( Interest[A] ){
  if( Priority == A ){
    Interest[B] = FALSE;
    while( Priority == A ){};
    Interest[B] = TRUE;
  }
}

        :
クリティカルセクション
        :

Priority = A;
Interest[B] = FALSE;
```

Dekkerのアルゴリズム



- Interest
  - クリティカルセクションに入る前に，まず「クリティカルセクションに入りたい」ことを宣言する
  - 競合者がいなければ入ることができる。

Dekkerのアルゴリズム



- Priority
  - 競合者がいた場合
  - Priorityが示す優先度で
    どちらが入るか決定
  - 自分に優先度が回ってくる
    までInterest状態を
    解除して待つ

Dekkerのアルゴリズム

- ポイント
  - 入る前に手を挙げる
  - 優先権により競合を解決する

- 問題点
  - ユーザプログラムに依存する
    ちゃんと各プロセスが約束を守ってくれないと破綻
  - ビジーウェイト（busy wait）の問題
    - 一方がクリティカルセクションを実行中,
      待っている方は優先権をひたすらチェックし続ける。
      →CPUリソースの無駄

POSIX Threadライブラリを利用した排他制御プログラム

実際のコードではどう書くのか

POSIX Treadライブラリ

- 「OSがもっているスレッド機能」を利用するための
  ライブラリ（libpthread）が提供されている。
- スレッド利用時に必要となる排他制御の機能なども
  提供される。

- ここでは pthread_mutex の基本的な使い方を理解する。

サンプルプログラム（排他制御）

■**PDC/C-Thread/mutex/**

■プログラムの説明
- ■nomutex.c：2つのスレッドが、
  共有資源であるグローバル変数 shared_resourceに対して、
  それぞれ 1を100万回 加算する。排他制御なし。
- ■mutex.c：上記と同じ。ただし排他制御あり

■コンパイルと実行
- ■make
- ■./nomutex

pthread_mutexを用いたスレッドの排他制御

- C言語でスレッドを使う場合、排他制御をするために
  ロックとして使用する変数は、通常の変数として宣言してはダメ。
- 排他制御型（pthread_mutex_t 型）の変数を使用し、
  必ず専用の関数を使って操作する。

```
pthread_mutex_t mutex;  //排他制御用変数の宣言
```

```
pthread_mutex_init ( &mutex, NULL ); //初期化

pthread_mutex_lock ( &mutex ); //ロック（ロックできたら戻ってくる）
    // クリティカルセクション
pthread_mutex_unlock ( &mutex ); //ロック解除
```

サンプルプログラム(mutex.c)

■lock/unlockでクリティカルセクションを挟む。

```
pthread_mutex_t mutex0;
pthread_mutex_init( &mutex0, NULL ); //初期化

void thread_A() { /* スレッドAの処理, 1を100万回加える */
  int i, x ;
  for( i = 0 ; i<1000000 ; i++ )
    {
      pthread_mutex_lock( &mutex0 );
        shared_resource++;
      pthread_mutex_unlock( &mutex0 );
    }
}
```

void thread_B() も同様

サンプルプログラム

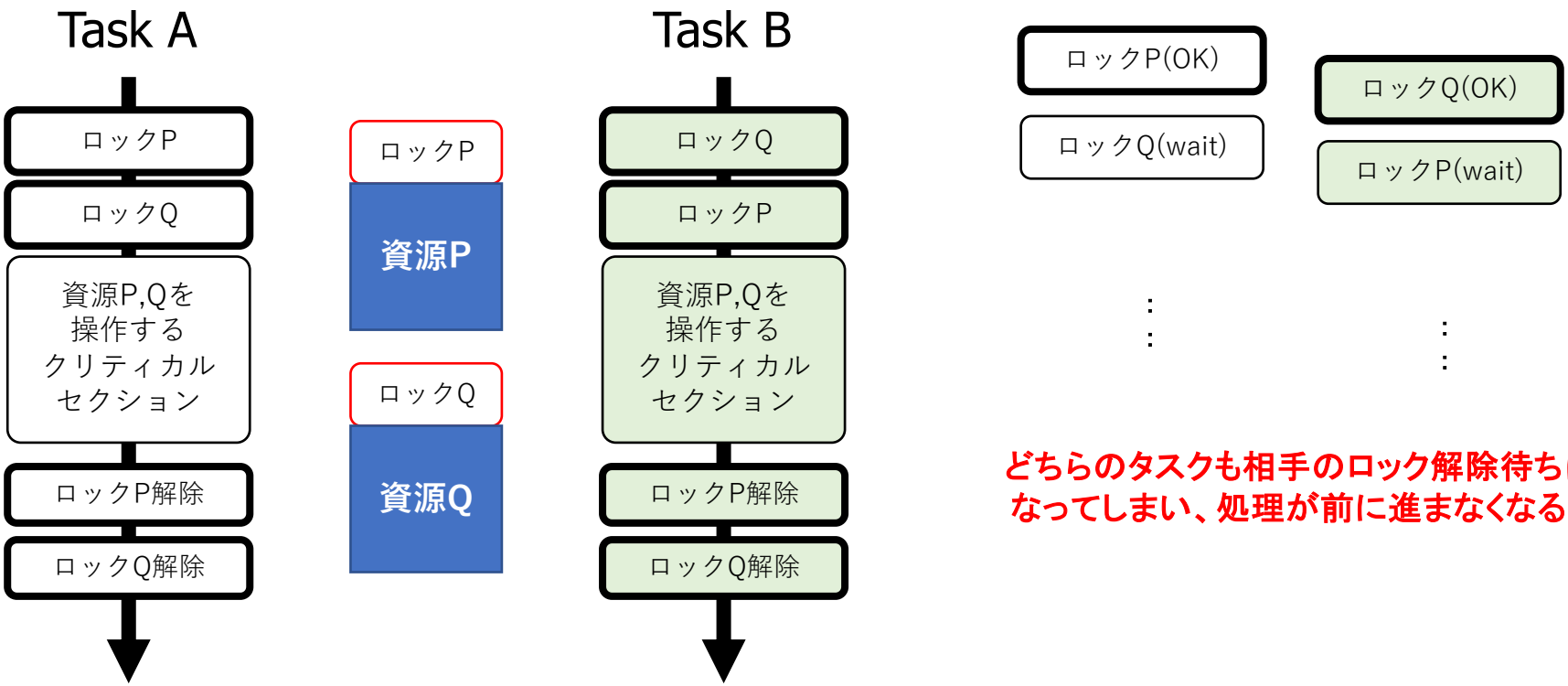
- **PDC/C-Thread/mutex/**

- コンパイルと実行
  - make
  - ./mutex

- 自分で確認しましょう。
  - 値が正しく計算されていること
  - for文を外側をlock/unlockで挟んだらどうなるか考えてみよう。

デッドロック（Dead Lock）

2つ以上のプロセスが 2つ以上のロックを操作する時、
デッドロックが発生する恐れがある。



**どちらのタスクも相手のロック解除待ちに
なってしまい、処理が前に進まなくなる。**

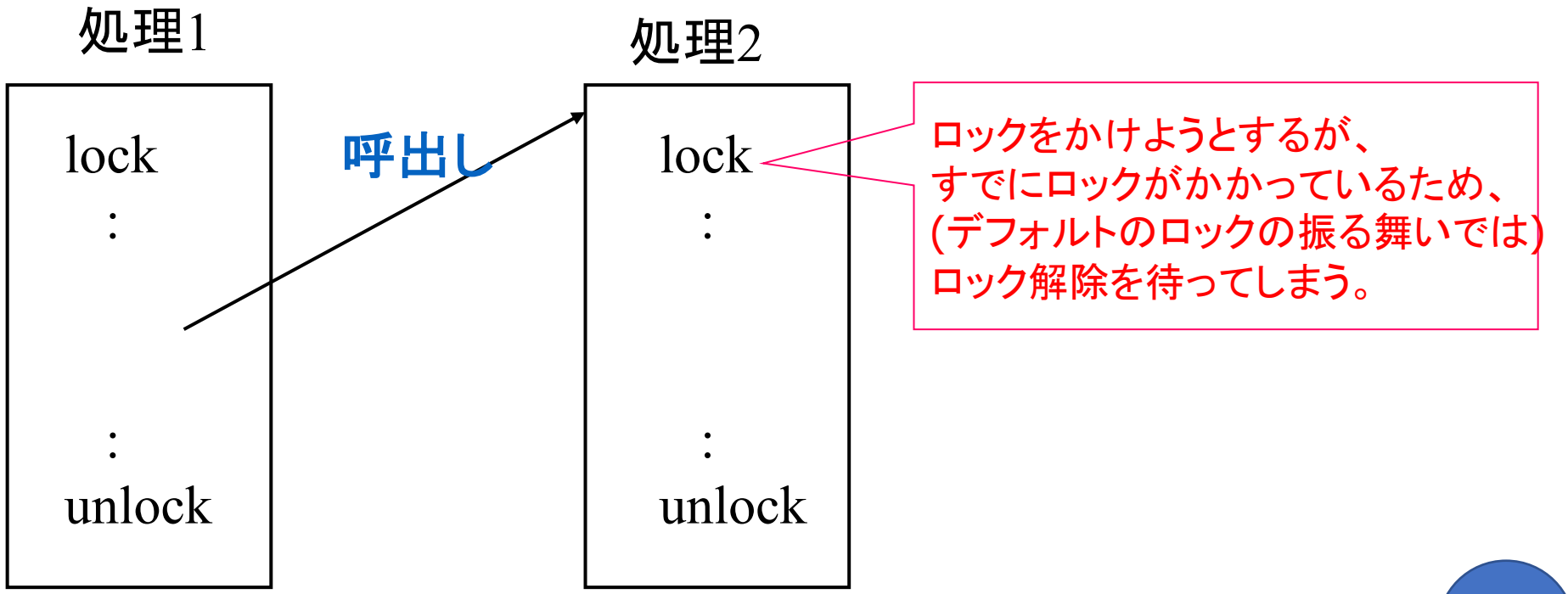
サンプルプログラム(deadlock)

- PDC/C-Thread/deadlock/

- プログラムの説明
  - deadlock1.c：
    2つのロックを互いに掛け合うプログラム
  - deadlock2.c：
    デッドロックを起こす利息計算を行うプログラム
  - recursive.c：
    再帰的ロックをゆるすように deadlock2.cを
    変更したプログラム(ちょっと特殊)

同一スレッドの中でもデッドロックは起きる

- デッドロックは、同じスレッド内で同じロックを踏んでしまうことでも発生する。



サンプルプログラム（deadlock2.c）

- ロックをかけた状態で呼び出した下請けの手続きが、同じロックをかけようとしてデッドロックになる。
  （ロックをかけようとしたら、呼び出し元がすでに同じロックをかけてた。）

```
deposit( int n ) /* 預金の処理 */
{
   pthread_mutex_lock( &mutex1 );
   shared_resource += n ; /* 元金にn円を加算 */
   pthread_mutex_unlock( &mutex1 );
}
```

```
add_interest() /* 利息を加える処理 */
{
   pthread_mutex_lock( &mutex1 );
   risoku = shared_resource * RATE ; /* 利息の計算 */
   deposit( risoku ); /* 預金に利息を加えたい.... */
   pthread_mutex_unlock( &mutex1 );
}
```

再帰的ロックを許す設定

- ロックの初期化（デフォルトの振る舞い）

```
pthread_mutex_init( &mutex, NULL);
```

- ロックの振る舞いを指定して初期化する。
  - ロックされた状態でさらにロックできる。
  - 内部でロックされた回数をカウントしている。
  - ロックと同じ回数アンロックすると、全解除状態になる。

```
pthread_mutexattr_t attr;
pthread_mutexattr_init( &attr );
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE_NP);
pthread_mutex_init( &mutex, &attr );
```